VMware vCenter API Performance Best Practices

Performance Study - May 10, 2023

Table of Contents

1. Executive Summary

Many customers and partners need to use vSphere APIs to monitor inventory, track events, collect statistics, or issue tasks. There are several documents about how to use these APIs, however, the APIs can still be challenging to use. In this paper, we discuss common issues we've come across, and look at best practices to get better performance from these interfaces. Some issues we discuss here include how to:

- Avoid memory leaks by destroying resources after use and monitoring only the required information.
- Reduce the latency overhead of connecting to vCenter using connection pooling or reusing sessions.
- Improve the performance of statistics and event queries by properly defining time ranges.

2. Introduction

In the following sections of this paper, we discuss how to:

- Use long-lived connections with `ConnectionPool`, rather than creating new connections every time, to reduce the connection overhead.
- Understand the effects of group memberships on vCenter memory when the user is part of many groups, and the need for persistent sessions.
- Fetch the smallest subset of required data, rather than fetching all of it.
- Use time ranges (start time and end time) when querying for statistics or events to reduce the database overhead for such queries.
- Use `cancelWaitForUpdates` to notify vCenter when the user has finished consuming updates.
- Destroy the View and Property Filter objects after use.

Note: All code snippets in this blog are respective to pyVmomi; similar alternatives are available in the other SDKs such as the Automation SDKs.

3. Connection Pooling

A *connection* refers to the TCP connection established from the client to the vCenter.

One source of overhead in most monitoring scripts is the repeated establishment of new connections from the client to the vCenter for various operations; instead, we recommend using a *connection pool*, which lets you reuse existing connections to vCenter, rather than creating new ones. By reusing connections, the client application avoids a costly handshake to vCenter. When the number of requests exceeds the number of connections, you could add more connections to the connection pool or wait until a connection is available.

Reusing the existing connections effectively keeps a check on the total number of connections being established to vCenter. Due to idle connection timeouts, if there are no live connections in the pool, you must create new connections and add them to the connection pool before using them.

For example, let's assume you want to issue 20 concurrent provisioning requests, and they maintain 5 connections in the connection pool. Since only 5 requests will be run at a time per the available connections, the other 15 requests will wait until they get a connection. As the requests complete execution, connections are returned to the connection pool, which can then be picked up by the waiting requests.

If your setup has multiple vCenters that the client needs to interact with, then we recommend maintaining connection pools dedicated to each vCenter.

Most available Automation SDKs have connection pooling logic built in, which can be tweaked on the client side as needed. For example, the Python SDK (pyVmomi), has a poolSize parameter in the SoapAdapter.py file, which refers to the HTTP Connection Pool size. The default pyVmomi HTTP Connection Pool size is 5, with the idle connection timeout set to 900 seconds. Similarly, Java has the `setMaxTotal` parameter to set the pool size in HttpClient.java; the default is 600.

4. Views

Views are used to observe the vCenter Inventory, to select or gain access to a set of objects, or to continuously monitor for changes. Views simplify the task of retrieving data from the vCenter. For example, if you want to monitor all the VMs that enter and exit the vCenter Inventory, you could use an `InventoryView`. Views are automatically updated as new objects enter or exit the vCenter Inventory. Views can be used with the `PropertyCollector` to detect incremental changes to certain properties of the object (see the code samples below). A View exists until you destory it or until the end of its session.

There are three types of Views:

- **ListView** - monitors a specific set of objects
- **ContainerView** - used to monitor hosts, data center, resource pools, and so on
- **InventoryView** - used to monitor the Inventory objects within a folder

4.1. Destroy Views

When creating a View object to monitor Inventory, you should destroy the View after use to avoid memory accumulation in the vCenter.

Another pattern for using a View is to create it once and keep it for the duration of your client. For example, if you have a backup job that needs to know which VMs reside on which host, you can create a View when your client starts up, and you can delete it when you are ready to terminate the client.

The provided interfaces have built-in methods to destroy the Views, which must be invoked after a View has served its purpose or is no longer of use.

When using pyVmomi to destroy a View, you need to invoke its `Destroy()` method. The equivalent while using the Java SDK is `destroyView(<ContainerView>)`. An example is shown on the next page.

Destroy ContainerView after its use - view.Destroy()

```
def some_function():
    container_view = []
    service_instance = []
    try:
        '''Connect to vCenter and get the ServiceInstance'''
        service_instance = connectToVC()
        global global_content
        global_content = service_instance.RetrieveContent()
        '''Create container view'''
        container_view = createContainerView(global_content, vim.VirtualMachine, False)
        '''Business logic goes here'''
        ... ...
    except Exception as e:
        log.error("<error_log>")
        return None
    finally:
        '''Destroy container view'''
        if container_view:
            destroyContainerView(container_view)
        '''Disconnect from vCenter'''
        if service_instance:
            disconnectFromVC(connect, service_instance)

def connectToVC():
    context = ssl.create_default_context()
    return connect.SmartConnect(host=<vCenter_IP>, user=<vCenter_username>,
pwd=<vCenter_password>, port=<vCenter_connection_port>, sslContext=context)

def disconnectFromVC(connect, service_instance):
    return connect.Disconnect(service_instance)

def createContainerView(global_content, type):
    return global_content.viewManager.CreateContainerView(global_content.rootFolder,
[type], True)

def destroyContainerView(view):
    return view.Destroy()
```

4.2. Container View

While creating a `ContainerView`, we recommend you avoid setting the optional `recursive` field to `True`.

- When `recursive` is `False`, the list of objects contains only immediate children.
- When `recursive` is `True`, the server populates the list by following references beyond the immediate children.

To keep track of the number of container views created, look at the profiler logs. An example follows.

createContainerView

```
# grep -rn "createContainerView" /var/log/vmware/vpxd/vpxd-profiler.log | grep
"numSamples" | tail -1
106802:-->
/ActivationStats/Task/Actv='vim.view.ViewManager.createContainerView'/TotalTime/numSam
ples 506
```

Similarly, to check on the number of destroyed Views, look at the `View.destroy` string in the profiler log.

5. Property Collector

Use the PropertyCollector to retrieve or monitor changes to a set of objects in the vCenter.

- The `RetrievePropertiesEx` method provides one-time property retrieval. Use this, for example, to retrieve all the existing VMs in a vCenter.
- The `WaitForUpdatesEx` method provides incremental change detection and supports both polling and notification. Use this, for example, to monitor the configuration changes to the VMs in a vCenter.

Use Property Filters to narrow down the focus area from the list of objects by specifying a few important properties of those objects.

- When a new session is created, a default `PropertyCollector` is created for that session.
- A session cannot share its `PropertyCollector` filters with other sessions.
- Two different clients can share the same session and so can share the same filters, but we don't recommend this.
- When a session ends, the associated `PropertyCollector` filters are automatically destroyed.

5.1. Calling CancelWaitForUpdates

When you want the user to poll for updates, create the necessary filters and call `waitForUpdatesEx` repeadedly. `waitForUpdatesEx` then detects the changes over a specified period of time. `cancelWaitForUpdates` must be called after consuming the required data from `waitForUpdatesEx`.

- `cancelWaitForUpdates` is a definitive way to let the vCenter know that the user is done consuming the necessary updates.
- If `cancelWaitForUpdates` is not called, memory will eventually accumulate on the vCenter.
- In a scenario where you don't want the user to stop consuming updates just yet, call `cancelWaitForUpdates`.

- `cancelWaitForUpdates` will attempt to cancel the outstanding calls to `waitForUpdatesEx` in the current session.

WaitForUpdatesEx and CancelWaitForUpdates - pc.CancelWaitForUpdates()

```
def GetProperties(service_instance, container_view, vm_properties, vm_spec_type):
    # Build a view and get basic properties for all Virtual Machines)
    pc = serviceInstance.content.propertyCollector
    try:
        traversal_spec = vmodl.query.PropertyCollector.TraversalSpec(name='tSpecName',
path='view', skip=False, type=container_view.__class__)
        property_spec = vmodl.query.PropertyCollector.PropertySpec(all=False,
pathSet=vm_properties, type=vm_spec_type)
        object_spec = vmodl.query.PropertyCollector.ObjectSpec(obj=container_view,
selectSet=[traversal_spec], skip=False)
        filter_spec = vmodl.query.PropertyCollector.FilterSpec(objectSet=[object_spec],
propSet=[property_spec], reportMissingObjectsInResults=False)

        pc_filter = pc.CreateFilter(filter_spec, True)
        wait_options = vmodl.query.PropertyCollector.WaitOptions()
        wait_options.maxWaitSeconds = 90
        version = ''
        iterations = 100

        while True:
            if iterations is not None:
                if iterations <= 1:
                    pc.CancelWaitForUpdates()
                    pc_filter.Destroy()
                    break
            result = pc.WaitForUpdatesEx(version, wait_options)
            if result is None:
                continue
            ''' If results contain data, process the results '''
            for filter_set in result.filterSet:
                ... <Do something with the filtered results> ...
            ''' Prepare for next iteration '''
            version = result.version
            if iterations is not None:
                iterations -= 1
    except Exception as e:
        <Error_logs go here>
        return None
```

5.2. Fetching the smallest set of data

When using a `PropertyCollector`, be sure to specify filters to fetch only the required data. For example, if you just need the name and memory size of a VM, then create a filter that retrieves the name and memory size. If instead, you create a filter for the entire configuration, you will retrieve far more data than you need. These large response sizes cause temporary memory accumulation in the vCenter and cause high latency for the response.

There are a number of factors that can affect the performance of a `PropertyCollector`; some examples include:

- The number of objects being fetched
- The sizes of these objects
- The number of properties in each object
- The size of each property
- The frequency of changes to these properties

Here is an example of where you'd want to fetch only the names of all the existing VMs in a vCenter.

Fetch only VM Names

```
def fetchVmNames(service_instance, container_view):
    '''Property collector instance'''
    pc = service_instance.content.propertyCollector

    '''Object spec'''
    object_spec = vmodl.query.PropertyCollector.ObjectSpec()
    object_spec.obj = containerView
    object_spec.skip = False

    '''Traversal spec'''
    traverse_spec = vmodl.query.PropertyCollector.TraversalSpec()
    traverse_spec1.name = 'traverseEntries'
    traverse_spec.path = 'view'
    traverse_spec.skip = False
    traverse_spec.type = container_view.__class__
    object_spec.selectSet = [traverse_spec]

    '''Property spec'''
  property_spec = vmodl.query.PropertyCollector.PropertySpec()
    property_spec.all = False
    property_spec.type = vim.VirtualMachine
    property_spec.pathSet = ["name"]

    '''Filter spec'''
    filter_spec = vmodl.query.PropertyCollector.FilterSpec()
    filter_spec.propSet = [property_spec]
    filter_spec.objectSet = [object_spec]

    '''Retrieve properties'''
    ret_options = vmodl.query.PropertyCollector.RetrieveOptions(maxObjects=10)
    result = pc.RetrievePropertiesEx([filter_spec], ret_options)
    if(result):
        ... <Do something with the resulting set of VM Names> ...
```

In the above example, notice that we fetch only the VM Names (`property_spec.pathSet = ["name"]`), instead of fetching the entire VM object and then parsing it to get the VM name from each VM object.

We recommend that you don't retrieve more than 10 objects a time: `(PropertyCollector.RetrieveOptions(maxObjects=10)`.

- Retrieving a higher number of objects at once leads to slower response times in most cases and sometimes leads to timeouts and exceptions.
- `ContinueRetrievePropertiesEx` can then be used to fetch the subsequent set of objects.

As an example, in our lab environment, we observed the following improvements upon using appropriate filters:

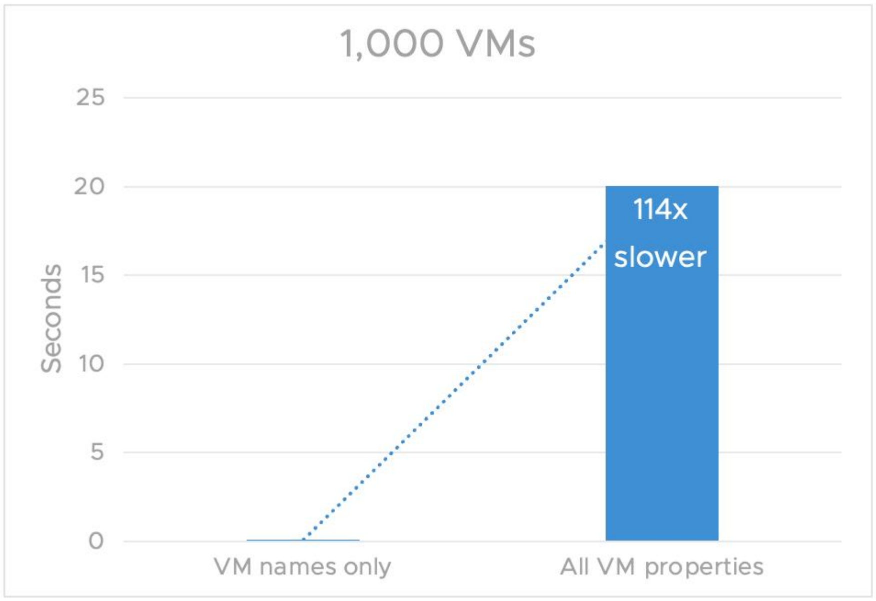

Figure 1. For 1,000 VMs, fetching all VM properties is 114 times slower than fetching VM names only.

5.3. Destroy Property Filter

We don't recommend destorying property filters after the user has consumed the required data because closing a session or destroying a `PropertyCollector` also removes all property filters associated with it, so you don't need to explicitly destroy them. If for some reason they aren't destroyed after use, numerous property filters could overload the vCenter.

Property collector filter - pc_filter.Destroy()

```
def fetchVmNames(service_instance, container_view):
    '''Property collector instance'''
    pc = service_instance.content.propertyCollector

    '''Object spec'''
    object_spec = vmodl.query.PropertyCollector.ObjectSpec()
    object_spec.obj = containerView
    object_spec.skip = False

    '''Traversal spec'''
    traverse_spec = vmodl.query.PropertyCollector.TraversalSpec()
    traverse_spec.name = 'traverseEntries'
    traverse_spec.path = 'view'
    traverse_spec.skip = False
    traverse_spec.type = container_view.__class__
    object_spec.selectSet = [traverse_spec]

    '''Property spec'''
    property_spec = vmodl.query.PropertyCollector.PropertySpec()
    property_spec.all = False
    property_spec.type = vim.VirtualMachine
    property_spec.pathSet = ["name"]

    '''Filter spec'''
    filter_spec = vmodl.query.PropertyCollector.FilterSpec()
    filter_spec.propSet = [property_spec]
    filter_spec.objectSet = [object_spec]

    '''Property Collector Filter'''
    pc_filter = pc.CreateFilter(filter_spec, True)
    ... <Poll to check for changes in properties of a resource> ...
    pc_filter.Destroy()
```

6. Querying by time range

While querying events or statistics from the vCenter, the response sizes can be quite large and query times can be slow if time ranges are not specified.

Querying events and stats are the heaving lifting operations in a vCenter, with massive amounts of data under each. These data help the clients understand the status, health, and performance of their systems.

- Processing all the available events or stats in the system is extremely time-consuming and cumbersome for the client.
- Since the data associated with these are huge, we recommend that the clients specify the time interval of interest to get minimal data set.
- Due to the large set of data associated with events and sats, if you want to fetch them for an entire day, we recommend you fetch them for a small interval (for example, once every hour) for the duration of that day, instead of using just one query to fetch all the data for that day. This way, you get a smaller set of events and stats for every hour, while also reducing the load on the vCenter to consolidate all this data.

6.1. Events

As part of the vCenter monitoring system, events are used to record significant state changes to the objects in vCenter. There are many events generated in the system every second: user events, host events, cluster events, VM events, and more.

- We recommend you use the `EventFilterSpec.ByTime` in pyVmomi to specify the time interval of interest to the client.
- If no interval is specified, then it brings out all the events in the database irrespective of when they occurred, and this usually takes a long time.

Here, we look at a code snippet where the function fetches the events that occurred within a specified time range: in this case, 30min.

Time ranged Events query - EventFilterSpec.ByTime()

```
def GetEventsForLast30min(service_instance):
    ''' Property collector '''
    pc = service_instance.content.propertyCollector

    try:
        ''' Time Filter for Events '''
        time_filter = vim.event.EventFilterSpec.ByTime()
        current_time = datetime.datetime.now()
        time_filter.beginTime = current_time - datetime.timedelta(minutes=30)
        time_filter.endTime = current_time

        ''' Event Filter Spec and Event Manager '''
        event_type_list = []
        filter_spec = vim.event.EventFilterSpec(eventTypeId=event_type_list,
time=time_filter)
        eventManager = serviceInstance.content.eventManager

        ''' Collect all the Events '''
        event_collector = eventManager.CreateCollectorForEvents(filter_spec)
        ### Events collected are not ordered by the Event creation_time.

        ''' Iterate through the collected Events, and read Next set of Events if
available '''
        page_size = 1000
        while True:
            ''' Number of loop iteration depends on the number of events that exist in
the specified time range '''
            events_in_page = event_collector.ReadNextEvents(page_size)
            num_event_in_page = len(events_in_page)
            if num_event_in_page == 0:
                break
            ... <Do something with the Events found in events_in_page> ...

    except Exception as e:
        <Error_logs go here>
        return None
```

6.2. Statistics

While retrieving statistical information of a target entity, take into consideration the following pointers to achieve better performance for your request.

Below are some suggestions to follow when creating a query spec:

1. Prefer to use the CSV format over the normal format when creating a query.
2. Specify the server side `startTime` and `endTime` to help reduce the search area for the query. If not specified, the result includes all the stats available in the database.

3. Avoid retrieving stats more frequently than they are refreshed. For example, when you retrieve 20-second interval data, the data will not change until the next 20-second data collection event.

Here, we look at a code snippet where the function fetches the stats within a specified time range: in this case, 5min.

Time ranged Stats query - PerformanceManager.QuerySpec()

```
def GetStatsForLast5min(entities):
    pc = si.content.propertyCollector
    perf_manager = si.content.perfManager

    try:
        list_of_query_specs = []
        query_spec = vim.PerformanceManager.QuerySpec()
        query_spec.format = "csv"

        ''' Define the time range for which you want the stats '''
        end_time = datetime.datetime.today()
        start_time = end_time - datetime.timedelta(seconds=300)
        query_spec.endTime = end_time
        query_spec.startTime = start_time
        query_spec.intervalId = 20
        for entity in entities:
            ''' For e.g., entities=cluster.host, and entity would be each host in that
cluster '''
            query_spec.entity = entity
            list_of_query_specs.append(query_spec)

        metrics_of_entities = perf_manager.QueryPerf(list_of_query_specs)

        ''' Go through all entities of query '''
        for entity in range(len(metrics_of_entities)):
            metrics_per_entity = metrics_of_entities[entity].value
            ''' Get an array consisting of performance counter information for the
specified counterIds, for each queried metric per entity. '''
            queried_counter_ids_per_entity = []
            for metric in metrics_per_entity:
                queried_counter_ids_per_entity.append(metric.id.counterId)

            perf_counter_info_list = perf_manager.QueryPerfCounter(queried_counter_ids
_per_entity)

            ''' Go through all the queried metrics for each entity '''
            for counter in range(len(metrics_per_entity)):
                ... <Do something with the queried metrics for each entity, and their
respective perf counters> ...

    except Exception as e:
        <Error_logs go here>
        return None
```

7. Sessions

When querying data from vCenter, we suggest using long-lived sessions rather than creating a new session for each retrieval. A session is created upon user login, and each session has a session ID. vCenter maintains state with each session. There are several advantages to using a single session:

- Users don't pay the cost of session creation when retrieving data, which also equates to vCenter having to deal with fewer sessions overall.
- Reduced chances of encountering the limit of 2000 sessions to vCenter or 1000 sessions to vAPI endpoint. Avoid 2048 RPCs overall to vCenter (more).
- A user is less likely to cause inadvertent memory leaks by forgetting to destroy per-session states. For example, one common problem is creating a `containerView` and forgetting to destroy it. Repeatedly doing this can lead to memory leaks.

Likewise, there are valid use cases where users may go for multiple sessions. Two examples of this are backup jobs triggered by multiple clients or the requirement to maintain stateless connections to vCenter.

7.1. Group memberships

Group memberships determine if the user is authorized to perform the requested actions against enterprise resources such as file servers or databases.

- It's possible for a single user to be part of multiple such groups with varying permissions.
- Clients generally use a large number of group memberships to grant permissions to different enterprise resources.
  - If a user is a part of many such groups, then the corresponding SAML token can bloat in size because the tokens hold all the group names that the user is part of.
  - If this user has many such active sessions at vCenter, then vCenter must hold that many bloated tokens in memory.
  - We recommend such users reuse their sessions instead of creating new ones so that the vCenter can deal with a minimal set of tokens.

7.2. Persistent sessions

We advise you to use distinct user identities for distinct jobs—for example, vRA user and vROps user—rather than a using a generic service user. Having distinct job-specific users helps you better understand the load being put on the system.

In addition:

- Consecutively logging in and out creates new sessions every time for that user. At vCenter, this results in objects being rapidly created and deleted, which needlessly impacts the vCenter's CPU.
- Reusing a session reduces the stress on vCenter to maintain the extra sessions that would have been created otherwise. Also, it helps the client to avoid certain roadblocks—for example, reaching the maximum session per user limit or the limits imposed by some of the vCenter service providers (more).
- It is still possible to use a single session for a given user identity and create many instances of the property collector in that session. This could simplify the client code because only one session needs to be kept alive.
- If a session is reused by multiple clients and the properties of interest for a client are different from those specified for the default property collector, we advise you create a new property collector for this client. This prevents polling/notification of unnecessary properties by this client.

8. vCenter automation: Using backoffs

The vCenter REST API interface is exposed via the Automation SDKs, also known as vAPI.

To make clients more robust when encountering errors, you can use backoffs. Having an effective backoff strategy helps balance the load of incoming requests from the client. For example, to prevent DDOS attacks, vapi-endpoint enforces certain throttles such as capping the maximum number of its incoming connections at 550. Exceeding this limit leads to a `Service unavailable` error with a 503 HTTP error code (more). In such cases, we recommend using backoff strategies. One way to implement them is by using a retry policy like RetryPolicy's `onInvocationError`, which is available through the Automation SDKs, or to have a local backoff strategy implementation on the client (for example, retry with incremental delays). We also recommend you reuse the user sessions whenever possible, as mentioned in the previous section, "7.2. Persistent sessions."

9. Resources

- Managed Object - View(vim.view.View)
  https://vdc-download.vmware.com/vmwb-repository/dcr-public/fe08899f-1eec-4d8d-b3bc-a6664c168c2c/7fdf97a1-4c0d-4be0-9d43-2ceebbc174d9/doc/vim.view.View.html
- Data Object - PerfQuerySpec(vim.PerformanceManager.QuerySpec)
  https://vdc-download.vmware.com/vmwb-repository/dcr-public/eda658cb-b729-480e-99bc-d3c961055a38/dc769ba5-3cfa-44b1-a5f9-ad807521af19/doc/vim.PerformanceManager.QuerySpec.html
- VMware vSphere Web Services SDK Programming Guide
  https://developer.vmware.com/docs/6611/vmware-vsphere-web-services-sdk-programming-guide
- Managed Object - ContainerView(vim.view.ContainerView)
  https://vdc-download.vmware.com/vmwb-repository/dcr-public/fe08899f-1eec-4d8d-b3bc-a6664c168c2c/7fdf97a1-4c0d-4be0-9d43-2ceebbc174d9/doc/vim.view.ContainerView.html
- Managed Object - ViewManager(vim.view.ViewManager)
  https://vdc-download.vmware.com/vmwb-repository/dcr-public/fe08899f-1eec-4d8d-b3bc-a6664c168c2c/7fdf97a1-4c0d-4be0-9d43-2ceebbc174d9/doc/vim.view.ViewManager.html
- vSphere Automation API Reference
  https://developer.vmware.com/apis/vsphere-automation/latest/
- Interface RetryPolicy
  https://vmware.github.io/vsphere-automation-sdk-java/vsphere/6.5.0/vsphereautomation-client-sdk/com/vmware/vapi/bindings/client/RetryPolicy.html
- Data Object - RetrieveOptions(vmodl.query.PropertyCollector.RetrieveOptions)
  https://vdc-repo.vmware.com/vmwb-repository/dcr-public/1ef6c336-7bef-477d-b9bb-caa1767d7e30/82521f49-9d9a-42b7-b19b-9e6cd9b30db1/vmodl.query.PropertyCollector.RetrieveOptions.html
- Endpoint Limits for Concurrent REST Requests from vCenter APIs
  https://blogs.vmware.com/performance/2022/07/endpoint-limits-for-concurrent-rest-requests-from-vcenter-apis.html
- Using vCenter to Deploy and Monitor Multi-Cloud Workloads at 16:50
  https://www.youtube.com/watch?v=takxz2sSkMo

About the authors

Amith M works as a performance engineer for the vCenter Performance team out of Bengaluru. He has a master's degree in computer science from the National Institute of Technology Calicut. He has worked extensively as a storage developer in the software-defined file storage domain.

Acknowledgments

The author thanks Ravi Soundararajan and Venu Uppalapati for their contributions to this paper.