



## vCD Sample Script

---

### <Introduction Required>

```
"""
.. module:: vCDclient
    :platform: Linux, Windows
    :synopsis: Reference module script to demonstrate the interaction
between VMware vCD
                and Cisco DCNM via VMware vCloud AMQP notification,
REST APIs and DCNM
                REST APIs.

.. moduleauthor:: Cisco DCNM team

.. note:: The configuration parameters need to be specified in
:file:`vCDclient-ini.conf` file
                before running this script.
"""
import sys, ConfigParser, time
import urllib2
import contextlib
import base64
import json
import requests
try:
    import xml.etree.cElementTree as et
except ImportError:
    import xml.etree.ElementTree as et
# for AMQP
import pika
```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

import logging
from logging import StreamHandler, FileHandler
logger = logging.getLogger('vCDclient')
class AMQPClient():
    """ This AMQP client class listens to vCD's AMQP notification
    and interacts
        with VMware vCloud Director (vCD) and vShield Manager (vSM) for
        further tenant
        and network information. It also communicates with DCNM to
        populate network data.
    """
    def __init__(self, params, client_vcds, client_dcnm):
        """Create a new instance of AMQP client.
        :param dict params: AMQP configuration parameters, e.g.
        AMQP server ip, port,
            user name, password, name of
        AMQP exchange and queue for vCD notification.
        :param list client_vcds: vCD client instances.
        :param object client_dcnm: DCNM instance.
        :raises: ValueError
        """
        # extract from input params
        self._server_ip = params.get('ip')
        self._port = int(params.get('port'))
        self._user = params.get('user')
        self._pwd = params.get('password')
        # exchange, queue name for receiving vCD events
        self._vcd_exchange_name = params.get('vcdexchangenname')
        self._vcd_queue_name = params.get('vcdqueueenname')
        self._client_vcds = client_vcds
        self._client_vcd = None
        self._client_dcnm = client_dcnm
        if (not self._server_ip) or (not self._vcd_exchange_name)
        or (not self._vcd_queue_name):
            raise ValueError, '[AMQPClient] Input IP, vCD
            exchange name or vCD queue name parameter is not specified'
            logger.info('[AMQPClient] AMQP server: %s, exchange
            name: %s, queue name: %s.' % (self._server_ip, self._vcd_exchange_name,
            self._vcd_queue_name))
            logger_pika = logging.getLogger('pika')

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        logger_pika.setLevel(logging.CRITICAL)
    def _cb_vcd_msg(self, ch, method, properties, body):
        """ Callback function to process vCD
organization/VDC/network
        creation/update/deletion AMQP messages being received.
        It also communicates with vCD and vSM to extract detailed
info
        and passed them to DCNM via DCNM REST APIs.
        :param pika.channel.Channel ch: The channel instance.
        :param method method: The method
        """
        if 'true.' not in method.routing_key:
            # send acknowledgement
            ch.basic_ack(delivery_tag = method.delivery_tag)
            return
        if 'network' in method.routing_key:
            self._process_org_vdc_network_msg(ch, method,
properties, body)
            # send acknowledgement
            ch.basic_ack(delivery_tag = method.delivery_tag)
            return
        key_org_create = 'com.vmware.vcloud.event.org.create'
        # no need to process vCD org modify event, as the only
field - org name is not editable in vCD
        key_org_delete = 'com.vmware.vcloud.event.org.delete'
        key_org_vdc_create = 'com.vmware.vcloud.event.vdc.create'
        key_org_vdc_update = 'com.vmware.vcloud.event.vdc.modify'
        key_org_vdc_delete = 'com.vmware.vcloud.event.vdc.delete'
        if (key_org_create in method.routing_key) or
(key_org_delete in method.routing_key):
            tenant_name = self._parse_vcd_org_event(body)
            if tenant_name:
                # add tenant entry
                if (key_org_create in method.routing_key):
self._client_dcnm.create_org(tenant_name)
                    else:
self._client_dcnm.delete_org(tenant_name)
            elif method.routing_key.endswith(key_org_vdc_create) \

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

                                or
method.routing_key.endswith(key_org_vdc_update) \
                                or
method.routing_key.endswith(key_org_vdc_delete):
                                (tenant_name, vdc_name) =
self._parse_vcd_org_vdc_event(body)
                                if tenant_name and vdc_name:
                                    if
method.routing_key.endswith(key_org_vdc_create):

self._client_dcnm.create_update_partition(tenant_name, vdc_name, True)
                                    elif
method.routing_key.endswith(key_org_vdc_update):

self._client_dcnm.create_update_partition(tenant_name, vdc_name,
False)

                                    else:

self._client_dcnm.delete_partition(tenant_name, vdc_name)
                                    # send acknowledgement
                                    ch.basic_ack(delivery_tag = method.delivery_tag)
def _process_org_vdc_network_msg(self, ch, method, properties,
body):
    """ Process vCD vDC network creation/update/deletion
AMQP message
    being received.
    It also communicates with vCD and vSM to extract detailed
info
    and pass them to DCNM via DCNM REST APIs.
:param pika.channel.Channel ch: The channel
    """
    if 'true.' not in method.routing_key:
        return

    key_vdc_network_create_complete =
'com.vmware.vcloud.event.task.complete.networkCreateOrgVdcNetwork'
    key_network_delete_complete =
'com.vmware.vcloud.event.task.complete.networkDelete'
    key_network_update_complete =
'com.vmware.vcloud.event.task.complete.networkUpdateNetwork'
    key_vapp_network_deploy =
'com.vmware.vcloud.event.network.deploy'
    key_vapp_network_undeploy =
'com.vmware.vcloud.event.network.undeploy'

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        if (key_vdc_network_create_complete not in
method.routing_key) \
                                and (key_vdc_network_create_complete not in
method.routing_key) \
                                and (key_network_delete_complete not in
method.routing_key) \
                                and (key_network_update_complete not in
method.routing_key) \
                                and (key_vapp_network_deploy not in
method.routing_key) \
                                and (key_vapp_network_undeploy not in
method.routing_key):
                return
        is_vapp_network = False
        is_create_network = True
        self._parse_vcd_event(body)
        if (key_vapp_network_deploy in method.routing_key) \
                or (key_vapp_network_undeploy in
method.routing_key):
                is_vapp_network = True
        if (key_network_update_complete in method.routing_key):
                is_create_network = False
        if (key_network_delete_complete in method.routing_key) \
                or (key_vapp_network_undeploy in
method.routing_key):
                network_info =
self._client_vcd.process_network_delete_message(body,
is_vapp_network)
                self._client_dcnm.delete_network(network_info)
        else:
                network_info =
self._client_vcd.process_network_create_update_message(body,
is_vapp_network)
        self._client_dcnm.create_update_network(network_info,
is_create_network)
    def _parse_vcd_event(self, msg):
        """Parse vCD event to find the input vCD instance with
matched IP address.
        :param str msg: The received vCD AMQP notification.
        :returns: object -- The matched vCD instance.
        """

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        if not msg:
            return
        # entity resolver
        root = et.fromstring(msg)
        # find entityResolver
node_resolver = root.find('//*[rel="entityResolver"]')
url_resolver = ''
        if node_resolver is not None:
            url_resolver = node_resolver.attrib['href']
        vcd_ip = url_resolver.split('/')[2]
        for client_vcd in self._client_vcds:
            if vcd_ip == client_vcd._vcd_ip:
                logger.debug('[AMQPClient] vCD IP: %s'
% vcd_ip)
                    self._client_vcd = client_vcd
                    break

def _parse_vcd_org_event(self, msg):
    """Parse vCD organization event to extract the organization
name.

:param str msg: The vCD organization AMQP notification.
:returns: str -- The organization name.
    """
    if not msg:
        return
    org_name = None
    # entity resolver
    root = et.fromstring(msg)
    # find org element
    node_org =
root.find('//*[rel="entity"][@type="vcloud:org"]')
    if node_org is not None:
        org_name = node_org.attrib['name']
    return org_name

def _parse_vcd_org_vdc_event(self, msg):
    """Parse vCD VDC event to extract the organization and
vDC name.

:param str msg: The vCD VDC AMQP notification.
:returns: tuple (str, str) -- (organization name, VDC name)
    """

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        if not msg:
            return
        org_name = None
        vdc_name = None
        # entity resolver
        root = et.fromstring(msg)
        # find vdc element
        node_vdc =
root.find('.//*[@rel="entity"][@type="vcloud:vdc]')
        if node_vdc is not None:
            vdc_name = node_vdc.attrib['name']
            node_orgs =
root.findall('.//*[@rel="up"][@type="vcloud:org]')
            for node_org in node_orgs:
                org_name = node_org.attrib['name']
                if org_name != 'System':
                    break;

        return (org_name, vdc_name)
def process_amqp_msgs(self):
    """Process AMQP queue messages.
    It connects to AMQP server and calls callbacks to process
VMware events,
    i.e. routing key containing '.vmware.', once they arrive
in the queue.
    """
    # specify the key of interest
    key = '#.vmware.#'
    self._conn = None
    consume_channel = None
    try:
        credentials = None
        if self._user:
            credentials =
pika.PlainCredentials(self._user, self._pwd)
            # create connection, channel
            self._conn =
pika.BlockingConnection(pika.ConnectionParameters(host =
self._server_ip, port = self._port, credentials = credentials))
            # create channels for consuming
            consume_channel = self._conn.channel()

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        # declare vCD exchange
        vcd_exchange =
consume_channel.exchange_declare(exchange = self._vcd_exchange_name,
type = 'topic', durable = True, auto_delete = False)

        result = consume_channel.queue_declare(queue =
self._vcd_queue_name, durable = True, auto_delete = False)

        consume_channel.queue_bind(exchange =
self._vcd_exchange_name, queue = self._vcd_queue_name, routing_key =
key)

        # for info only
        msg_count = result.method.message_count

        logger.info('[AMQPClient] The exchange %r queue
%r has totally %d messages. ' % (self._vcd_exchange_name,
self._vcd_queue_name, msg_count))

        print ' [*] About to retrieve messages. Press
Ctl-C to exit'

        # consume messages
        consume_channel.basic_consume(self._cb_vcd_msg,
queue = self._vcd_queue_name)

        consume_channel.start_consuming()
    except KeyboardInterrupt:
        print '\n Received Ctl-C.'
    finally:
        # don't call cancel or close due to pika's error
        #
        #         consume_channel.cancel()
        #         consume_channel.close()

        if self._conn:
            self._conn.close()

class VCDWSClient():

    """ This vCD Web Service client class interacts with vCD and
vSM for detailed tenant
    and network information.
    """

    def __init__(self, params_vcd, params_vsm):
        """Create a new instance of vCD client.
        :param dict params_vcd: vCD configuration parameters,
e.g. vCD ip and user name.
        :param dict params_vsm: vSM configuration parameters,
e.g. vSM ip and user name.

```



**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        :raises: ValueError
        :param dict params_vcd:
vCD configuration parameters, e.g. vCD ip and user name.
        """
        self._req_header = {'Accept': 'application/*+xml'}
        self._url_login = None
        self._version_num = 1.5
        # vCD namespace
        self._NS_VCD = 'http://www.vmware.com/vcloud/'
        # format {vcd_network_entity_id: segment_id}
        self._network_mapping = {}
        # hard-coded tenant mapping
        self._tenant_mapping = {}
        # url timeout: 10 seconds
        self._TIMEOUT_URL_OPEN = 10
        # extract from input params
        self._vcd_ip = params_vcd.get('ip')
        # vCD user format: userName@org
        self._vcd_user = '%s@system' % (params_vcd.get('user'))
        self._vcd_pwd = params_vcd.get('password')
        self._vsm_ip = params_vsm.get('ip')
        self._vsm_user = params_vsm.get('user')
        self._vsm_pwd = params_vsm.get('password')
        if (not self._vcd_ip) or (not self._vcd_user) or (not
self._vcd_pwd):
            raise ValueError, '[VCDWSCClient] Input vCD IP,
user name or password parameter is not specified'
        elif (not self._vsm_ip) or (not self._vsm_user) or (not
self._vsm_pwd):
            raise ValueError, '[VCDWSCClient] Input vSM IP,
user name or password parameter is not specified'
        logger.info('[VCDWSCClient] vCD IP: %s, vCD User: %s,
vSM IP: %s, vSM User: %s.' % (self._vcd_ip, self._vcd_user,
self._vsm_ip, self._vsm_user))
        def get_tenant_vdc_network(self):
            """ Retrieve all the organization, VDC and networks.
            This method does not return all the data in one shot;
            instead for code efficiency, it returns
            organization, VDC or network during its looping through
            all the network related data in vCD.
            :returns: * tuple (str) -- Organization name

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        * Or tuple (str, str) -- (Organization name,
VDC name)
        * Or tuple (str, str, dict) -- (Organization
name, VDC name, network info)
        :note: The caller needs to loop through the returned
data until no more data is available.
        """
        try:
            #login
            self._login()
            # retrieve tenants list
            tenants_url = 'https://%s/api/org' % (self._vcd_ip)
            tenants_msg = self._get_response(tenants_url)
            tenants_root = et.fromstring(tenants_msg)
            # extract namespace
            tag_vcd = tenants_root.tag
            ns_vcd = ''
            if tag_vcd.startswith('{'):
                ns_vcd = tag_vcd[1:].split('}')[0]
            # extract tenant nodes
            tenant_nodes = tenants_root.findall('.//{%s}Org'
% ns_vcd)

            for tenant_node in tenant_nodes:
                tenant_name = tenant_node.attrib['name']
                tenant_url = tenant_node.attrib['href']
                # ignore system built-in tenant: System
                if tenant_name == 'System':
                    continue
                yield (tenant_name)
            tenant_msg = self._get_response(tenant_url)
            tenant_root = et.fromstring(tenant_msg)
            vdc_name = []
            vdc_nodes =
tenant_root.findall('.//*[@type="application/vnd.vmware.vcloud.vdc+xml"']')

            for vdc_node in vdc_nodes:
                vdc_name = vdc_node.attrib['name']
                yield (tenant_name, vdc_name)
            tenant_id = tenant_url.split('/')[-1]

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        networks_nodes =
tenant_root.findall('.//*[@type="application/vnd.vmware.vcloud.orgNet
work+xml"]')

        for network_node in networks_nodes:
            network_url =
network_node.attrib['href']

            network_info =
self._compose_network_info(tenant_id, tenant_name, network_url)

            yield (tenant_name, vdc_name,
network_info)

        except (urllib2.HTTPError, urllib2.URLError) as e:
            if isinstance(e, urllib2.HTTPError):
                reason = 'Error reaching URL (%s) with
code %s.' % (e.url, e.code)
            else:
                reason = 'Error reaching URL (%s) with
reason %s.' % (e.url, e.reason)

            logger.exception(reason)

        finally:
            self._logout()

    def process_network_create_update_message(self, msg,
is_vapp_network):
        """Process vCD's network creation and update event.
        It retrieves detailed network info from vCD via vCD REST
        APIs, adds the vCD network Id
        to class network mapping table (for segment Id retrieval
        later) and deletes the vSE (due
        to some duplicated features offered by DFA leaf nodes).
        :param str msg: The vCD's AMQP notification.
        :param bool is_vapp_network: The flag to indicate whether
        the input message
        is related to vApp network.
        :returns: dict -- Network data which includes tenant_name
        (organization name),
        network_name, segment_id as keys.
        """
        logger.info('[VCDWSCClient] Start processing network
create/update message ...')
        network_info = None
        tenant_name = None
        network_name = None

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        segment_id = None
    try:
        # parse the event first, so as to generate DCNM event
        network_event_info =
self._parse_vcd_network_event(msg)
        if not network_event_info:
            return
        # login
        self._login()
        tenant_id = network_event_info['tenant_id']
        tenant_name = network_event_info['tenant_name']
        urn_network = network_event_info['urn_network']
        network_id = network_event_info['network_id']
        network_link = self._get_network_link(urn_network)
        network_info = self._compose_network_info(tenant_id,
tenant_name, network_link)
        if not network_info:
            return
        segment_id = network_info['segment_id']
        network_name = network_info['network_name']
        # add network Id to global network mapping table,
with segment Id being filled later
        self._set_segment_id(network_id, segment_id,
is_vapp_network)

        self._logout()
        # delete vSM edge (vSE)
        self._delete_vsm_edge(tenant_id, network_name)
        # adding a 10s delay to ensure that vSE is deleted
        # before ldap is populated
        time.sleep(10)
    except (urllib2.HTTPError, urllib2.URLError) as e:
        if isinstance(e, urllib2.HTTPError):
            reason = 'Error reaching URL (%s) with
code %s.' % (e.url, e.code)
        else:
            reason = 'Error reaching URL (%s) with
reason %s.' % (e.url, e.reason)
        logger.exception(reason)
    return network_info

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

def process_network_delete_message(self, msg, is_vapp_network):
    """Process vCD's network deletion event.

    It retrieves detailed network info from vCD via vCD REST
    APIs, and finds the matched
        segment Id from class network mapping table based on
    extracted vCD network Id.

    :param str msg: The vCD's AMQP notification.
    :param bool is_vapp_network: The flag to indicate whether
    the input message
        is related to vApp network.

    :returns: dict -- Network data which includes tenant_name
    (organization name),
        and segment_id as keys.

    """
    logger.info('[VCDWSCClient] Start processing network
    delete message ...')
    network_event_info = self._parse_vcd_network_event(msg)
    if not network_event_info:
        return
    network_id = network_event_info['network_id']
    segment_id = self._lookup_segment_id(network_id,
    is_vapp_network, True)
    network_info = {'tenant_name':
    network_event_info['tenant_name'],
                    'segment_id': segment_id
                    }
    return network_info

def _lookup_segment_id(self, vcd_network_entity_id,
    is_vapp_network, remove_entry):
    """Find segment Id based on input vCD network entry Id.

    DFA uses segment Id to identify the network, whereas
    vCD uses UUID formatted network
        Id to identify the network. So a network mapping table
    needs to be maintained to
        map between vCD network Id and DFA segment Id.

    :param str vcd_network_entry_id: The network Id used by
    vCD to identify the network.

    :param bool is_vapp_network: The flag to indicate whether
    the input message
        is related to vApp network.

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        :param bool remove_entry: The flag to indicate whether
to remove the entry from class
                                network mapping table.
        :returns: str -- The matched segment Id if found.

        """

        for network_id, (segment_id, vapp_network_flag) in
self._network_mapping.iteritems():
            if (network_id == vcd_network_entity_id) and
(vapp_network_flag == is_vapp_network):
                if remove_entry:
                    del self._network_mapping[network_id]
                return segment_id

    def _set_segment_id(self, vcd_network_entity_id, segment_id,
is_vapp_network):
        """Add segment Id and vCD network entry Id entry to
internal mapping table
        for network deletion message processing later.

        :param str vcd_network_entity_id: The network Id used by
vCD to identify the network.
        :param str segment_id: The segment Id used by DFA to
identify the network.
        :param bool is_vapp_network: The flag to indicate whether
the input message
                                is related to vApp network.

        """

        if not segment_id:
            return
        for network_id, (network_segment_id, vapp_network_flag)
in self._network_mapping.iteritems():
            if segment_id == network_segment_id:
                return
        self._network_mapping.update({vcd_network_entity_id:
(segment_id, is_vapp_network)})

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```
def _compose_request_header(self, url):
    """Compose HTTP request header.

    :param str url: The URI that the request is sent to.
    :returns: Request -- The HTTP request object.

    """

    req = urllib2.Request(url)
    for key, value in self._req_header.iteritems():
        req.add_header(key, value)
    return req

def _get_response(self, url):
    """Generalize the HTTP(S) request/response processing.

    :param str url: The URI that the request is sent to.
    :returns: str -- The HTTP response body message.

    """

    req = self._compose_request_header(url)
    content = None
    try:
        with contextlib.closing(urllib2.urlopen(req,
            timeout = self._TIMEOUT_URL_OPEN)) as res:
            content = res.read()
    except urllib2.URLLError as e:
        # add url to the exception for caller to display
        e.url = url
        raise
    return content

def _parse_vcd_network_event(self, msg):
    """Parse vCD network event.
```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        :param str msg: The vCD AMQP notification.
        :returns: dict -- The extracted network data which have
tenant_id (organization Id),
                                tenant_name (organization name),
network_id (vCD network
                                entry Id) and urn_network (URN
of vCD network entry) as keys.

```

```

"""

if not msg:
    return

# entity resolver
root = et.fromstring(msg)

# find tenant Id (org)
node_org = root.find('.//*[@type="vcloud:org]')
org_id = node_org.attrib['id'].split(':')[0]
org_name = node_org.attrib['name']

# find entityResolver
node_resolver = root.find('.//*[@rel="entityResolver]')
url_resolver = ''
if node_resolver is not None:
    url_resolver = node_resolver.attrib['href']

# find network id
node_network = root.find('.//*[@type="vcloud:network]')
network_id = ''
if node_network is not None:
    network_id = node_network.attrib['id']

urn_network = ''.join([url_resolver, network_id])

vcd_network = {'tenant_id': org_id,
               'tenant_name': org_name,

```



**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        'network_id': network_id,
        'urn_network': urn_network
    }
    return vcd_network

```

```

def _get_network_link(self, urn_network):
    """Retrive network reference link.

```

It sends HTTP(S) GET request to vCD and extracts the network reference

link from the response body.

:param str urn\_network: The URN of network entry in vCD.

:returns: str -- The URI of network reference link.

```

    """

```

```

    network_link = None
    # resolve network entity
    entity_resolver_res = self._get_response(urn_network)

    # find the first href for network
    if entity_resolver_res:
        root = et.fromstring(entity_resolver_res)
        node_alternate = root.find('.//*[@rel="alternate"]')
        if node_alternate is not None:
            network_link = node_alternate.attrib['href']

    return network_link

```

```

def _vsm_ws_request(self, vsm_url, delete_action = False):
    """Send HTTP(S) GET or DELETE request to vSM.

```

This method is called to retrieve networks (virtual wires), edges and delete individual edge.

:param str vsm\_url: The URL of vSM resource.

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        :param bool delete_action: The flag to indicate whether
it is DELETE request.

        :returns: str -- The HTTP(S) response body.

        """

        req = urllib2.Request(vsm_url)
        req.add_header('Accept', 'application/*+xml')
        if delete_action:
            req.get_method = lambda:'DELETE'

        # use base64
        base64string = base64.encodestring('%s:%s' %
(self._vsm_user, self._vsm_pwd))[:-1]
        req.add_header("Authorization", "Basic %s" % base64string)
        content = None
        try:
            with contextlib.closing(urllib2.urlopen(req,
timeout = self._TIMEOUT_URL_OPEN)) as res:
                content = res.read()
        except urllib2.URLError as e:
            # add url to the exception for caller to display
            e.url = 'vSM: ' + vsm_url
            raise
        return content

    def _get_vsm_segment_id(self, tenant_id, network_name):
        """Retrieve segment Id from vSM which has the matched
organization Id and network name.

        :param str tenant_id: The organization Id in vCD.
        :param str network_name: The network name.
        :returns: tuple -- (segment Id in vSM, port profile name
in N1KV)

        """

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

url_networks = 'https://%s/api/2.0/vdn/virtualwires' %
(self._vsm_ip)
vsm_networks = self._vsm_ws_request(url_networks)
segment_id = None
port_profile_n1kv_name = None
if vsm_networks:
    # network is vSM's virtual wire
    root = et.fromstring(vsm_networks)
    for network in root.findall('./virtualWire'):
        tenant_id_value =
network.find('tenantId').text
        name_value = network.find('name').text
        if((tenant_id_value == tenant_id) and
(network_name in name_value)):
            # find segment Id
            segment_id = network.find('vdnId').text
            # compose N1KV (vDS) port profile name

#<virtualWire><objectId>virtualwire-6</objectId>
        virtualwire_id =
network.find('objectId').text

#<vdsContextWithBacking><switch><objectId>dvs-136
        dvs = network.find('./switch')
        dvs_id = ''
        if dvs:
            dvs_id = dvs.find('objectId').text
            port_profile_name =
'vxw-%s-%s-sid-%s-%s' % (dvs_id, virtualwire_id, segment_id,
name_value)
            # example:
vxw-dvs-136-virtualwire-6-sid-10003-dvs.VCDVSNetPepsiInternal-50bc778
c-2fcd-454a
            # N1kV port profile name max length: 80
            port_profile_n1kv_name =
port_profile_name[:80]
            break

return (segment_id, port_profile_n1kv_name)

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

def _delete_vsm_edge(self, tenant_id, network_name):
    """Delete vSE by sending HTTP(S) DELETE to vSM edge
resource.

:param str tenant_id: The organization Id in vCD.
:param str network_name: The network name.
:returns: str -- The vSE edge Id used in vSM.

"""

    edge_id = None

    url_edges = 'https://%s/api/3.0/edges' % (self._vsm_ip)
    try:
        vsm_edges = self._vsm_ws_request(url_edges)
        if vsm_edges:
            root = et.fromstring(vsm_edges)
            for edge in root.findall('.//edgeSummary'):
                name_value = edge.find('name').text
                node_tenant_id = edge.find('tenantId')
                if node_tenant_id is None:
                    continue
                tenant_id_value = node_tenant_id.text
                if((tenant_id_value == tenant_id)
and (network_name in name_value)):
                    # find edge Id
                    edge_id =
edge.find('objectId').text

                    break

            # delete edge
            if edge_id:
                url_edge = 'https://%s/api/3.0/edges/%s'
% (self._vsm_ip, edge_id)

                self._vsm_ws_request(url_edge, True)
                logger.debug('[vCDWSCClient] Delete vSM
edge. Edge Id: %s.' % edge_id)

    except urllib2.HTTPError as e:

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        logger.exception('[vCDWSCClient] Error code: %s
' % e.code)

        except urllib2.URLError as e:
            logger.exception('[vCDWSCClient] Error reaching
URL (%s) with reason %s.' % (e.url, e.reason))

        return edge_id

    def _compose_network_info(self, tenant_id, tenant_name,
network_url):
        """Retrive detailed network info and parse/compose
network data.

        :param str tenant_id: The organization (tenant) Id in vCD.
        :param str tenant_name: The organization name.
        :param str network_url: The URI that the HTTP(S) GET
request is sent to

            for detailed network info.

        :returns: dict -- The detailed network data which have
network_id (organization Id),
            tenant_name, vrf_name (VDC name),
network_name, segment_id,
            gateway, netmask, port_profile
(on N1KV), dns, ip_start and
            ip_end as keys.

        """
        if not network_url:
            return
        vcd_network = self._get_response(network_url)
        node_vcd_network = et.fromstring(vcd_network)

        # find VDC name (network's parent)
        node_vdc =
node_vcd_network.find('.//*[@type="application/vnd.vmware.vcloud.vdc+
xml" ]')

        vdc_name = ''
        if node_vdc is not None: # vCD 1.5 does not have VDC

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        vdc_url =
node_vcd_network.find('.*[@type="application/vnd.vmware.vcloud.vdc+
xml"]').attrib['href']

        vdc_msg = self._get_response(vdc_url)
        vdc_name = et.fromstring(vdc_msg).get('name')

        network_name = node_vcd_network.get('name')

        # contact vSM for segment id
        (segment_id, port_profile_n1kv_name) =
self._get_vsm_segment_id(tenant_id, network_name)

        # extract namespace
tag_vcd = node_vcd_network.tag
if tag_vcd.startswith('{'):
    ns_vcd = tag_vcd[1:].split('}') [0]
    # find subnet info
node_ipscope = node_vcd_network.find('.//{%s}IpScope'
% ns_vcd)

gateway = node_ipscope.find('{%s}Gateway' % ns_vcd).text

netmask = node_ipscope.find('{%s}Netmask' % ns_vcd).text
node_dns = node_ipscope.find('{%s}Dns1' % ns_vcd)
dns = ''
if node_dns is not None:
    dns = node_dns.text

    gateways = gateway.split('.')
node_ip_start = node_ipscope.find('.//{%s}StartAddress'
% ns_vcd)

    if node_ip_start is not None:
        ip_start = node_ip_start.text
    else:
        ip_start = '.'.join((gateways[0], gateways[1],
gateways[2], str(int(gateways[3])+ 1)))

node_ip_end = node_ipscope.find('.//{%s}EndAddress' %
ns_vcd)

    if node_ip_end is not None:

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        ip_end = node_ip_end.text
    else:
        netmasks = netmask.split('.')
        ip_end_last_num = 255 - int gateways[3] -
int(netmasks[3])
        ip_end = ''.join((gateways[0], gateways[1],
gateways[2], str(ip_end_last_num)))

    network_info = {'tenant_id': tenant_id,
                    'tenant_name': tenant_name,
                    'vrf_name': vdc_name,
                    'segment_id': segment_id,
                    'network_name' : network_name,
                    'gateway': gateway,
                    'netmask': netmask,
                    'port_profile' : port_profile_n1kv_name,
                    'dns': dns,
                    'ip_start': ip_start,
                    'ip_end': ip_end }

    logger.debug('[vCDWClient] Network info: %s ' %
network_info )

    return network_info

def _login(self):
    """Find out vCD version and log into vCD.

    vCD returns session token in login response header after
successful login, and that token
    will be added to the class request header field to be
used for subsequent request
    composition.

    """

    # for error report
    url = ''

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

try:
    if not self._url_login:
        # first time to retrieve login url
        url_version = 'http://%s/api/versions'
% (self._vcd_ip)
        ns_version = '%sversions' % (self._NS_VCD)

        url = url_version
        url_login = None
        with
contextlib.closing(urllib2.urlopen(url_version, timeout =
self._TIMEOUT_URL_OPEN)) as res:
            root_version = et.fromstring(res.read())
                versions =
root_version.findall('.//{%s}VersionInfo' % (ns_version))
                for version_info in versions:
                    version =
version_info.find('{%s}Version' % (ns_version)).text
                    if float(version) >
self._version_num:
                        self._version_num
= float(version)
                        url_login =
version_info.find('{%s>LoginUrl' % (ns_version)).text

                if not url_login:
                    self._url_login = None
                    return

                # change url to https as vCD only supports
https
                self._url_login = url_login.replace('http://',
'https://')

                # basic authentication from session login
                pwd_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
                pwd_mgr.add_password(None, self._url_login,
self._vcd_user, self._vcd_pwd)
                handler = urllib2.HTTPBasicAuthHandler(pwd_mgr)

                opener = urllib2.build_opener(handler)

```



**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        # add header
        opener.addheaders = [('Accept',
'application/*+xml;version=%s' % str(self._version_num))]
        url = self._url_login
        with
contextlib.closing(opener.open(self._url_login, data = '', timeout =
self._TIMEOUT_URL_OPEN)) as res:
            # session Id
            session_id =
res.info().getheader('x-vcloud-authorization')
            # update global request header
            self._req_header =
{'Accept': 'application/*+xml;version=%s' % self._version_num,
'x-vcloud-authorization': session_id }

    except urllib2.URLError as e:
        # add url to the exception for caller to display
        e.url = 'login: ' + url
        raise

def _logout(self):
    """Log out from vCD.
    """

    if not self._url_login:
        return

    # replace 'sessions' to 'session'
    url_logout = self._url_login.replace('sessions',
'session')

    req = self._compose_request_header(url_logout)
    req.get_method = lambda: 'DELETE'
    try:
        with contextlib.closing(urllib2.urlopen(req,
timeout = self._TIMEOUT_URL_OPEN)) as res:
            logger.debug('[vCDWClient] Logout
result: %s' % (res.read()))
    except urllib2.URLError as e:
        # add url to the exception for caller to display

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        e.url = 'logout: ' + url_logout
        raise

class DCNMClient():
    """ This DCNM client class interacts with DCNM via DCNM REST
    API to populate
        organization (tenant), partition (vrf, VDC) and network data.
    """

    def __init__(self, params_dcnm, params_tenant):
        """Create a new instance of DCNM client.

        :param dict params_dcnm: DCNM configuration parameters,
        e.g. DCNM server IP, user name.

        :param dict params_tenant: Default parameters for
        organization (tenant) such as orchestration source.

        """

        self._ip = params_dcnm.get('ip')
        self._user = params_dcnm.get('user')
        self._pwd = params_dcnm.get('password')
        if (not self._ip) or (not self._user) or (not self._pwd):
            raise ValueError, '[DCNMClient] Input DCNM IP,
            user name or password parameter is not specified'
            logger.info('[DCNMClient] DCNM IP: %s, User: %s.' %
            (self._ip, self._user))

        # tenant defaults
        self._default_forwarding_mode =
        params_tenant.get('defaultforwardingmode', 'proxy-gateway')
        self._default_profile =
        params_tenant.get('defaultprofilename', 'GoldProfile')
        self._orchestration_source =
        params_tenant.get('orchestrationsource', 'vCloud Director')

        # url timeout: 10 seconds
        self._TIMEOUT_RESPONSE = 10

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

def create_org(self, org_name):
    """Create organization (tenant) by sending POST request
    to DCNM auto-config organizations resource.

    :param str org_name: The name of organization to be
    created.

    """

    url = 'http://%s/rest/auto-config/organizations' %
(self._ip)

    payload = {'organizationName': org_name,
               'profileName': self._default_profile,
               'forwardingMode':
self._default_forwarding_mode,
               'orchestrationSource':
self._orchestration_source
               }

    self._send_request('POST', url, payload, 'organization')

def delete_org(self, org_name):
    """Delete organization (tenant) by sending DELETE request
    to DCNM auto-config organizations resource.

    :param str org_name: The name of organization to be
    deleted.

    """

    url = 'http://%s/rest/auto-config/organizations/%s' %
(self._ip, org_name)

    self._send_request('DELETE', url, '', 'organization')

def create_update_partition(self, org_name, partition_name,
is_create = True):
    """Create or update partition (vrf, VDC) by sending POST
    or PUT request to

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

DCNM auto-config partitions resource.

:param str org_name: The organization name.
:param str partition_name: The partition name.
:param bool is_create: The flag to indicate whether to
create organization.

"""

        url =
'http://%s/rest/auto-config/organizations/%s/partitions' % (self._ip,
org_name)

        composed_partition_name =
self._compose_partition_name(org_name, partition_name)

        operation = 'POST'
        if not is_create:
            operation = 'PUT'
            url =
'http://%s/rest/auto-config/organizations/%s/partitions/%s' %
(self._ip, org_name, composed_partition_name)

        payload = {'organizationName': org_name,
                    'partitionName': composed_partition_name,
                    'profileName': self._default_profile,
                    'forwardingMode':
self._default_forwarding_mode
                    }

        self._send_request(operation, url, payload, 'partition')

    def delete_partition(self, org_name, partition_name):
        """Delete partition (vrf, VDC) by sending DELETE request
to DCNM auto-config partitions resource.

:param str org_name: The organization name.
:param str partition_name: The partition name.

"""

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        composed_partition_name =
self._compose_partition_name(org_name, partition_name)

        url =
'http://%s/rest/auto-config/organizations/%s/partitions/%s' %
(self._ip, org_name, composed_partition_name)

        self._send_request('DELETE', url, '', 'partition')

def create_update_network(self, network_info, is_create = True):
    """Create or update network by sending POST or PUT
request to DCNM auto-config networks resource.

    :param dict network_info: The network info which includes
tenant_name (organization name),
                                vrf_name (partition name),
segment_id, gateway and netmask.
    :param bool is_create: The flag to indicate whether to
create network.

    """

    org_name = network_info.get('tenant_name', '')
    partition_name = network_info.get('vrf_name', '')
    composed_partition_name =
self._compose_partition_name(org_name, partition_name)

    url =
'http://%s/rest/auto-config/organizations/%s/partitions/%s/networks'
% (self._ip, org_name, composed_partition_name)
    operation = 'POST'
    if not is_create:
        operation = 'PUT'
        url =
'http://%s/rest/auto-config/organizations/%s/partitions/%s/networks/s
egment/%s' % (self._ip, org_name, composed_partition_name, segment_id)

    segment_id = network_info['segment_id']
    gateway = network_info.get('gateway', '')
    netmask = network_info.get('netmask', '')
    netmask_len = 24
    if netmask != '':

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        netmask_len = self._convert_netmask(netmask)

        config_args = []
        config_args.append('$vrfName=%s' %
composed_partition_name)
        config_args.append('$segmentId=%s' % segment_id)
        config_args.append('$netMaskLength=%d' % netmask_len)
        config_args.append('$gatewayIpAddress=%s' % gateway)
        config_args.append('$forwardingMode=%s' %
self._default_forwarding_mode)
        config_args = ';'.join(config_args)

        ip_start = network_info.get('ip_start', '')
        ip_end = network_info.get('ip_end', '')
        subnet = gateway[:gateway.rfind('.') + 1] + '0'

        dhcp_scopes = {'ipRange': ('%s-%s' % (ip_start, ip_end)),
                        'subnet': ('%s/%d' % (subnet, netmask_len)),
                        'routers': gateway,
                        'segmentID': segment_id
                        }

        payload = {'networkName': network_info['network_name'],
                  'partitionName': composed_partition_name,
                  'profileName': self._default_profile,
                  'forwardingMode':
self._default_forwarding_mode,
                  'segmentId': segment_id,
                  'configArg': config_args,
                  'dhcpScope': dhcp_scopes
                  }

        self._send_request(operation, url, payload, 'network')

    def delete_network(self, network_info):
        """Delete network by sending DELETE request to DCNM
auto-config networks resource.

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        :param dict network_info: The network info which includes
tenant_name (organization name),
                                vrf_name (partition name) and
segment_id.

        """

        if 'segment_id' not in network_info:
            return

        # Note: vCD network deletion notification does not
contain VDC info, so waiting for DCNM to add search API to make vCD
network deletion work.

        org_name = network_info.get('tenant_name', '')
        partition_name = network_info.get('vrf_name', '')
        composed_partition_name =
self._compose_partition_name(org_name, partition_name)
        segment_id = network_info['segment_id']
        url =
'http://%s/rest/auto-config/organizations/%s/partitions/%s/networks/s
egment/%s' % (self._ip, org_name, composed_partition_name, segment_id)

        self._send_request('DELETE', url, '', 'network')

def _compose_partition_name(self, org_name, partition_name):
    """Compose partition name.

    :param str org_name: The organization name.
    :param str partition_name: The partition name.
    :returns: str -- The name with 'orgName_partitionName'
format to avoid possible
                                duplicated partition name among
different organization.

    """

    # combine org and partition name as partition/vrf name
is MUST attribute in LDAP partition/vrf table
    return org_name + '_' + partition_name

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

def _send_request(self, operation, url, payload, desc):
    """Generalize the HTTP(S) request, which includes POST,
    PUT, DELETE.

    :param str operation: The HTTP verb with value of POST,
    PUT or DELETE.

    :param str url: The URI that the request is sent to.

    :param dict payload: The data to be put in the request
    body. It will
                                be converted into JSON format
    before being sent out.

    :param str desc: The description to be recorded in log
    message.

    :returns: Response -- The response object from HTTP(S)
    request.

    :notes: It logs into DCNM, sends HTTP(S) request, and
    log out from DCNM.

    """

    res = None
    try:
        payload_json = None
        if payload and payload != '':
            payload_json = json.dumps(payload)
        self._login()
        if operation == 'POST':
            res = requests.post(url, data =
payload_json, headers = self._req_headers, timeout =
self._TIMEOUT_RESPONSE)
            desc += ' creation'
        elif operation == 'PUT':
            res = requests.put(url, data = payload_json,
headers = self._req_headers, timeout = self._TIMEOUT_RESPONSE)
            desc += ' update'
        elif operation == 'DELETE':
            res = requests.delete(url, data =
payload_json, headers = self._req_headers, timeout =
self._TIMEOUT_RESPONSE)

```



**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        desc += ' deletion'

        logger.debug('\n [DCNMClient] REST Response
code: %d, content: %s \n' % (res.status_code, res.content))
        if res and res.status_code >= 200:
            logger.info('[DCNMClient] Sent %s to %s
successfully.' % (desc, url))
        else:
            logger.error('[DCNMClient] Sent %s to
%s unsuccessfully.' % (desc, url))

        self._logout()
    except requests.ConnectionError as e:
        # add url to the exception for caller to display
        print 'Error connecting to ', url
        logger.exception(str(e))
        raise
    except requests.HTTPError as e:
        print 'HTTP error'
        logger.exception(str(e))
    except requests.Timeout as e:
        print 'Timeout error'
        logger.exception(str(e))

    return res

def _login(self):
    """Log into DCNM by calling POST request to DCNM logon
resource.

    DCNM returns DCNM token in login response after successful
login, and that token

    will be added to the class request header field to be
used for subsequent request composition.

    """

    url_login = 'http://%s/rest/logon' % (self._ip)

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        expiration_time = 100000

        payload = {'expirationTime': expiration_time}
        self._req_headers = {'Accept': 'application/json',
'Content-Type': 'application/json; charset=UTF-8'}
        res = requests.post(url_login, data = json.dumps(payload),
headers = self._req_headers, auth = (self._user, self._pwd), timeout =
self._TIMEOUT_RESPONSE)
        logger.debug(['DCNMClient] Login response: %s' %
(res.content))

        session_id = ''
        if res and res.status_code >= 200:
            session_id = res.json().get('token')
        # update global request header
        self._req_headers.update({'Dcnm-Token': session_id })

    def _logout(self):
        """Log out from DCNM by calling POST request to DCNM
logout resource
        """

        url_logout = 'http://%s/rest/logout' % (self._ip)
        requests.post(url_logout, headers = self._req_headers,
timeout = self._TIMEOUT_RESPONSE)

    def _convert_netmask(self, netmask):
        """Convert netmask from dotted decimal to bitmask.

:param str netmask: The netmask in dotted decimal format.
:returns: int -- The bitmask (length).
        """

        arr = netmask.split('.')
        arr = map(int, arr)
        return reduce(lambda x, y: x + (y + 1)/32, arr, 0)

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```
def read_config_file(config_file):
    """Read initial configuration file.

    :param config_file: Configuration file name.

    """

    config_params = {}

    parser = ConfigParser.ConfigParser()
    parser.readfp(open(config_file))

    for section in parser.sections():
        section_params = {}
        for option in parser.options(section):
            values = parser.get(section, option)
            if ';' in values:
                values = values.split(';')
            section_params.update({option: values})
        config_params.update({section: section_params})

    return config_params

def set_logger():
    """Set logger with log file name and log message format.

    The log messages are written to 'vcdclient.log' file.
    """

    default_formatter = logging.Formatter('%(asctime)s
    %(levelname)s: %(message)s')
    handler_console = StreamHandler()
    handler_console.setFormatter(default_formatter)
    handler_console.setLevel(logging.DEBUG)

    handler_file = FileHandler('vcdclient.log', 'a')
```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

handler_file.setFormatter(default_formatter)

logger.addHandler(handler_console)
logger.addHandler(handler_file)

def query_process_vcd_tenants(client_vcds, client_dcnm):
    """ Retrieve the current tenants and network info from vCD, and
    calls DCNM to
        create or update organization, partition and network data.

    :param client_vcds: List of vCD instances
    :param client_dcnm: DCNM instance

    """

    # no need to proceed if DCNM client is not present
    if not client_dcnm:
        return

    logger.info('Retrieving tenants (organizations), VDC
(partitions) and network info from vCD.')
    for client_vcd in client_vcds:
        """ Delete partition (vrf) by sending DELETE request to
        DCNM auto-config partitions resource.
        """
        for tenant_info in client_vcd.get_tenant_vdc_network():
            # retrieve all the {tenant: vdc/vrf} from vCD
            # tenant
            if (type(tenant_info) == str) or (len(tenant_info)
            == 1):
                (tenant_name) = tenant_info
                # add tenant entry
                client_dcnm.create_org(tenant_name)
            elif len(tenant_info) == 2:
                (tenant_name, vdc_name) = tenant_info
                # add vrf entry

    client_dcnm.create_update_partition(tenant_name, vdc_name, True)

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        elif len(tenant_info) == 3:
            (tenant_name, vdc_name, segment_data)
= tenant_info
            # add segment entry

client_dcnm.create_update_network(segment_data, True)

if __name__ == '__main__':
    """Main function for vCDclient flow.

    It reads configuration parameters from the 'vCDclient-ini.conf'
file, retrieves all the
    organization, VDC and network data from VMware vCD, passes the
info to DCNM, and processes
    AMQP notification from vCD.

    """

    config_file_name = 'vCDclient-ini.conf'

    set_logger()

    try:

        config_params = read_config_file(config_file_name)

        # get config params
        params_log = config_params.get('Log')
        params_amqp = config_params.get('AMQP')
        params_vcd = config_params.get('vCD')
        params_vsm = config_params.get('vSM')
        params_dcnm = config_params.get('DCNM')
        params_tenant = config_params.get('Tenant')

        # set logger level
        log_levels = {
            'DEBUG': logging.DEBUG,
            'INFO': logging.INFO,
            'WARNING': logging.WARNING,

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```

        'ERROR': logging.ERROR,
        'CRITICAL': logging.CRITICAL
    }
    logger.setLevel(log_levels.get(params_log['level'],
logging.INFO))

    # check config parameters
    if (not params_amqp) or (not params_vcd) or (not
params_vsm):
        logger.error('Section [AMQP], [vCD] or [vSM] is
missing in ini.conf file.')
        exit(1)

    logger.info('Parsed config file %s' % config_file_name)

    client_vcds = []
    params_vcd_values = params_vcd.itervalues().next()

    if isinstance(params_vcd_values, list):
        len_param_vcd = len(params_vcd_values)
        for i in range(len_param_vcd):
            param_vcd = {}
            for k in params_vcd:
                param_vcd.update({k: params_vcd[k][i]})
            logger.debug('vCD input parameters: %s.'
% param_vcd)

            param_vsm = {}
            for k in params_vsm:
                param_vsm.update({k: params_vsm[k][i]})
            logger.debug('vSM input parameters: %s'
% param_vsm)

            client_vcds.append((VCDWSCClient(param_vcd,
param_vsm)))
        else:
            client_vcds.append((VCDWSCClient(params_vcd,
params_vsm)))

    client_dcnm = DCNMClient(params_dcnm, params_tenant)
    query_process_vcd_tenants(client_vcds, client_dcnm)

```

**REVIEW DRAFT – CISCO CONFIDENTIAL**

```
client_dcnm)    client_amqp = AMQPClient(params_amqp, client_vcds,  
  
                client_amqp.process_amqp_msgs()  
  
                logger.info('Exit the program.\n')  
                exit(0)  
  
except Exception as e:  
    logger.exception(str(e))  
    logger.info('Exit the program.\n')  
    exit(1)
```

***REVIEW DRAFT – CISCO CONFIDENTIAL***