Scaling Virtual Network Functions

Scaling Virtual Network Functions Using ETSI API

One of the main benefits of ESC is its capability to elastically scale a service. This allows a VNFC that performs a particular role or aspect within the VNF to be able to service requests and scale out to meet high demand or scale in when being under utilized. This aspect may span across multiple VNFCs.

The scaling requests may be manual or automatic. The different approaches to accomplishing scaling are detailed below.

For more details on these concepts and specification, please see Annex B of *ETSI GS NFV-SOL 003*.

For information on Scaling VNFs using REST and NETCONF APIs, see the *Cisco Elastic Services Controller User Guide*.

Scale

The Scale VNF request uses the *scaleStatus*, an attribute found as part of the instantiatedVnfInfo when querying a VnfInstance resource. This attribute describes the current scale level of each aspect in the VNF, for example:

```
"scaleInfo": [
  {
    "aspectId": "webserver", "scaleLevel": "4"
  },
  {
    "aspectId": "processing", "scaleLevel": "2"
  }
]
```

This forms the starting point for a Scale VNF request, which allows a single aspect to be scaled horizontally (i.e. adding or removing VNFCs) relative to the current *scaleLevel* for that dimension of the VNF. Any scaling operation on an aspect will be applied to each VNFC that supports that aspect.

Note The current specification does not support vertical scaling (adding/removing resources to/from existing VNFC instances) at this time.

Request Payload (ETSI data structure: ScaleVNFRequest)

```
{
    "type": "SCALE_OUT",
    "aspectId": "processing",
    "numberOfSteps": 1,
    "additionalParams": {}
}
```

The above payload results in the *scaleStatus* example above being updated to and the addition of the number of VNFCs for this step required to scale out to scaleLevel 3:

```
"scaleInfo": [
  {
    "aspectId": "webserver", "scaleLevel": "4"
  },
  {
    "aspectId": "processing", "scaleLevel": "3"
  }
]
```

To understand the scaling steps and other related policies configured to support scaling, see the VNFD Policies for Scaling.

Scale To Level

The Scale VNF To Level request, rather than the relative scaling that Scale VNF offers, specifies the absolute scale result desired and so some aspects may be scaled out and others scaled in. This option uses one of the two approaches to define the scaling required:

- instantiation level
- scale level

These are mutually exclusive and allow for more than one aspect to be scaled in a single request.

Instantiation Level

An Instantiation level is a predefined size for each aspect, where each level has a scale level associated with each aspect. There is no further granularity offered and so the entire VNF (that is, all aspects) is scaled according to the instantiation level requested.

Example:

Request Payload (ETSI data structure: ScaleVNFToLevelRequest)

```
{
    "instantiationLevelId": "premium"
}
```

See the VNFD Policies for the definition of instantiation levels.

Scale Level

The Scale Level is also a pre-defined size for each aspect where each aspect has target VNFCs, defined step_deltas (since each scaling step may not be uniform) and a maximum scale level. The policies that define this option allow the different targets to have different scaling outcomes.

Note The scale level does not represent the number of VMs; for example scaleLevel=0 means the initial number of instances (initial delta) for that aspect on the target VNFC and scaleLevel=1 is the initial delta plus the first scaling step defined for that aspect and VNFC tuple.

Request Payload (ETSI data structure: ScaleVNFToLevelRequest)

```
{
    "scaleInfo": [
      {
            "aspectId": "processing",
            "scaleLevel": "2"
      },
      {
            "aspectId": "webserver",
            "scaleLevel": "3"
      }
    ]
}
```

For information on definition of scale levels, See the VNFD Policies for Scaling.

ESC ETSI Support for Trunks and Subports

ETSI VLAN Trunk:

Introduction:

For OpenStack VIMs, starting from 5.8, ESC supports trunks and VLANs. The initial release was limited to the ESC Netconf/APIs and trunk enabled VNFs were not scalable. The introduction of TOSCA SOL003 3.5.1 version provided new node types allowing an ETSI VNFD to define trunks and subports. With the ESC 5.9 release, the ETSI VNFM supports scalable trunks and subports.

Defining a Trunk in the VNFD:

The TOSCA type *tosca.nodes.nfv.VduSubCp* is available from SOL001 3.5.1. Use the VNFD version that is SOL001 3.5.1 or higher.

Apply the ETSI Trunk Mode to CPs that is Connection Points between the VDU which is Virtualisation Deployment Unit and VL which is the Virtual Link or network. For a given CP, setting a trunk_mode property value as *true* signifies it as being the parent port for a trunk.

Example Payload:

```
s3_nic0:

type: tosca.nodes.nfv.VduCp
  properties:
    layer_protocols: [ ipv4 ]
    protocol:
      - associated_layer_protocol: ipv4
    trunk_mode: true  # denotes the parent port
    order: 0
    management: false
    allowed_address_pairs:
      - ip_address: 192.168.0.0/18
  requirements:
    - virtual_binding: s3
```

Setting the *trunk_mode* property creates a trunk. The CP is the primary port for the VDU linked by *virtual_binding*. The trunk name is generated in the format "trunk-" + VDU name + "-" + index number. The index is based on the number of CPs in trunk mode for the current VDU. Note that setting *trunk_mode* can be done at instantiation time.

Defining Subports in the VNFD:

To make a trunk useful, the trunk needs to connect to other networks through subports. A subport is defined with a node of type *tosca.nodes.nfv.VduSubCp* as follows:

Sample Payload:

```
s3_nic0_1:
  type: tosca.nodes.nfv.VduSubCp
  properties:
    layer_protocols: [ ethernet, ipv4 ]
    segmentation_type: vlan
    segmentation_id: 303
    management: false
  requirements:
    - trunk_binding: s3_nic0
    - virtual_link: a_vlan_VL
```

Here the segmentation type and ID are configured. The requirements properties have two links:

- trunk_binding: The node name of the CP where the primary port is defined that is trunk_mode set to true
- virtual_link: The name of the VL node to which this subport will be connected.

Example Payload for VL that is type tosca.nodes.nfv.VnfVirtualLink:

```
a_vlan_VL:
  type: tosca.nodes.nfv.VnfVirtualLink
  properties:
    connectivity_type:
      layer_protocols: [ ethernet ]
    description: subport VL
    vl_profile:
      max_bitrate_requirements:
        root: 100000
      min_bitrate_requirements:
        root: 0
      virtual_link_protocol_data:
      - associated_layer_protocol: ethernet
        l2_protocol_data:
          vlan_transparent: false
          segmentation_id: 303
```

Note Configure the subports with the JSON payload at instantiation time together with user data that are input variables.

The following shows the traditional dep.xml produced by ETSI constructs:

```
 <trunk>
    <name>trunk-name-0</name> <!-- Derived from VDU name and index -->
    <parent_nicid>0</parent_nicid> <!-- Primary port -->
    <subports>
        <subport>
            <name>trunk-name-0-subport-0</name> <!-- Derived from trunk name and subport
index -->
```

```
            <network>child-net</network>
            <segmentation_type>vlan</segmentation_type>
            <segmentation_id>500</segmentation_id>
            <binding_profile>
                <property>
                    <name>physical_network</name>
                    <value>physnet_tenant1</value>
                </property>
                <property>
                    <name>trusted</name>
                    <value>true</value>
                </property>
            </binding_profile>
        </subport>
    </subports>
</trunk>
```

ETSI VNF SCALING:

Trunk and subports scale automatically depending on the policy defined in the VNFD. As ESC scales the VNF up and down, additional trunks and subports are created or deleted as necessary. These are managed by ESC. ESC ensures VIM resources are cleaned up during LCM operations that are modify, delete.

Note During scaling, ESC duplicates the trunk and port names and relies on resource IDs when updating or deleting.

For ETSI, scaling is controlled according to the scaling policies.

```
######
# VM #
######
    - vm_initial_delta:
        type: tosca.policies.nfv.VduInitialDelta
        properties:
          initial_delta:
            number_of_instances: 2
        targets: [ s3_nic0 ]

    - vm_instantiation_levels:
        type: tosca.policies.nfv.VduInstantiationLevels
        properties:
          levels:
            default:
              number_of_instances: 2
        targets: [ s3_nic0 ]

    - vm_scaling_aspect_deltas:
        type: tosca.policies.nfv.VduScalingAspectDeltas
        properties:
          aspect: default_scaling_aspect
          deltas:
            delta_1:
              number_of_instances: 2
            delta_2:
              number_of_instances: 3
        targets: [ s3_nic0 ]
```

The following shows the traditional dep.xml produced by ETSI constructs:

```
<scaling>
    <min_active>1</min_active>
    <max_active>2</max_active>
</scaling>
```

And the appropriate VM group blocks are created.

Scaling Behaviour within ESC:

When a VMGroup is scaled up, then the corresponding trunks and subports are created, and the deployment detail queries through REST or Netconf APIs show the trunk and subport details.

When a VMGroup is scaled down, then the corresponding trunks and subports are deleted from the VIM, and deployment detail queris through REST or Netconf show the new trunk and subport details.

Updating SOL001 Parser to Support The trunk_mode Property for the Connection Points

The interfaces currently configured by ESC are not trunk ports, and so they do not support the definition of sub-ports. To use the networks more efficiently, segment the network using VLANs to connect multiple Layer 2 networks to a single pass-through interface. The following data model supports this configuration.

The following is an extract of a VNFD for a VPC-DI, with a parent port shown to be a trunk port, with 2 subports defined - one with an external VL connection that is exposed as an external connection through `substitution_mappings` and the other connected to an internal VL that is both of which specify their own segmentation Id.

```
s3_nic0:
      type: tosca.nodes.nfv.VduCp
      properties:
        layer_protocols: [ ipv4 ]
        protocol:
          - associated_layer_protocol: ipv4
        trunk_mode: true  # denotes the parent port
        order: 0
        management: false
        allowed_address_pairs:
          - ip_address: 192.168.0.0/18
      requirements:
        - virtual_binding: vdu_node_1

    s3_nic0_1:
      type: tosca.nodes.nfv.VduSubCp
      properties:
        layer_protocols: [ ipv4 ]
        protocol:
          - associated_layer_protocol: ipv4
        trunk_mode: false
        segmentation_type: vlan
        segmentation_id: 303
        management: false
      requirements:
        - trunk_binding: s3_nic0
        - virtual_link: a_vlan_VL
```

Note The `trunk_mode` is set to `true`, indicating that when the port is created, it is used as a trunk port and sub-ports are configured within the trunk network.

This results in the following deployment XML:

```
<trunks>
                    <trunk>
                       <name>trunk-vdu_node_1-0</name>
                       <parent_nicid>0</parent_nicid>
                       <subports>
                          <subport>
                             <name>trunk-vdu_node_1-0-subport-0</name>
                             <network>a_vlan_VL</network>
                             <segmentation_type>vlan</segmentation_type>
                             <segmentation_id>303</segmentation_id>
                          </subport>
                          <subport>
                             <name>trunk-vdu_node_1-0-subport-1</name>
                             <network>a_vlan_VL</network>
                             <segmentation_type>vlan</segmentation_type>
                             <segmentation_id>304</segmentation_id>
                          </subport>
                       </subports>
                    </trunk>
                 </trunks>
```

VNFD Policies for Scaling

There are a number of policies that make up the overall scaling behavior of a VNF. These policies will support the various scaling approaches described above. The first policy defines the aspects that may be scaled (or not):

```
policies:
    - scaling_aspects:
        type: tosca.policies.nfv.ScalingAspects
        properties:
          aspects:
            webserver:
              name: 'webserver'
              description: 'The webserver cluster.'
              max_scale_level: 5
              step_deltas:
                - delta_1
            processing:
              name: 'processing'
              description: 'An example processing function'
              max_scale_level: 3
              step_deltas:
                - delta_1
                - delta_2
                - delta_1
            database:
              name: 'database'
              description: 'A test database'
              max_scale_level: 0
```

You can see in this example that the database aspect has a max_scale_level of 0, which denotes that it cannot be scaled out - this does not mean 0 instances of that aspect - see the algorithm below to see why. The webserver aspect only has a single step_delta, meaning that all scaling steps are uniform whereas the processing aspect has different step_deltas specified for each scaling step. This is called non-uniform scaling. This is only the declaration of the aspects of this VNF, and this is one of the policies used to perform the validation when a scaling request is received.

Next, they must be applied to VNFCs to control their behavior:

```
- db_initial_delta:
        type: tosca.policies.nfv.VduInitialDelta
        properties:
          initial_delta:
            number_of_instances: 1
        targets: [ vdu1 ]

    - ws_initial_delta:
        type: tosca.policies.nfv.VduInitialDelta
        properties:
          initial_delta:
            number_of_instances: 1
        targets: [ vdu2, vdu4 ]

    - pc_initial_delta:
        type: tosca.policies.nfv.VduInitialDelta
        properties:
          initial_delta:
            number_of_instances: 1
        targets: [ vdu3 ]

    - ws_scaling_aspect_deltas:
        type: tosca.policies.nfv.VduScalingAspectDeltas
        properties:
          aspect: webserver
          deltas:
            delta_1:
              number_of_instances: 1
        targets: [ vdu2, vdu4 ]

    - pc_scaling_aspect_deltas:
        type: tosca.policies.nfv.VduScalingAspectDeltas
        properties:
          aspect: processing
          deltas:
            delta_1:
              number_of_instances: 1
            delta_2:
              number_of_instances: 2
        targets: [ vdu2, vdu4 ]
```

In the examples above, the VNFCs are identified as targets; the aspects could have different behaviours on different VNFCS, but this is not shown here. The definition of the step_deltas are also shown here which are used in the validation and generation of scaling requests (these steps are inferred by the scale level requested). The minimum number of instances of a VNFC is always assumed to be 0 and the maximum number is calculated by the following algorithm:

initial_delta plus the number of instances for each step up to the max_scale_level.

These policies are considered for the scale-level based scaling. There are similar constructs used for instantiation-level based scaling.

```
- instantiation_levels:
        type: tosca.policies.nfv.InstantiationLevels
        properties:
          levels:
            default:
              description: 'Default instantiation level'
              scale_info:
                database:
                  scale_level: 0
                webserver:
                  scale_level: 0
                processing:
```

```
            scale_level: 0
        premium:
          description: 'Premium instantiation level'
          scale_info:
            database:
              scale_level: 0
            webserver:
              scale_level: 2
            processing:
              scale_level: 3
      default_level: default
```

Similar to the scaling aspects, the first part of the definition of instantiation levels is just their declaration. Here each aspect must already be declared and then each aspect's scale_level is declared for the instantiation level; a default instantiation level is also stipulated in the event that no other is specified. What each scale_level means for each VNFC is further elaborated upon in the VduInstantiationLevels policies, for example:

```
- ws_instantiation_levels:
        type: tosca.policies.nfv.VduInstantiationLevels
        properties:
          levels:
            default:
              number_of_instances: 1
        targets: [ vdu2, vdu4 ]
```

So these policies together state that the default instantiation level is 'default' which will result in the webserver aspect being instantiated at scale_level 0 which is 1 VNFC instance.

Dependencies on Multiple IP Addresses

Static IP Addresses

If the VNFC has connection points configured with a static IP address, the VNFC cannot scale as there are no further IP addresses to assign to the connection points on the newly spun up VNFC instances. Instead, you can specify a pool of static IP addresses in the instantiate request or Grant response (in the extVirtualLinks element) as a list:

- in fixedAddresses in a single cpProtocolData
- of individual fixedAddresses in multiple cpProtocolData

Note A list of ipAddresses in a single cpProtocolData assigns all the IP addresses to a single port on a single VNFC instance.

Alternatively, a contiguous range can also be supplied in an ipAddresses entry, as an addressRange. If the specific IP addresses need not be stipulated, then a subnetId can be used, as per the example in Instantiating Virtual Network Functions.

The following example explains how to create a static IP pool with four IP addresses by specifying them as a list in fixedAddresses in a single cpProtocolData:

```
    {
…
"extVirtualLinks": [
    {
      "id": "extVL-dbf477ad-199a-47ff-939a-cb0101c92585",
```

```
      "resourceId": "ext-net",
      "extCps": [
        {
          "cpdId": "ecp_1_vdu_node_1",
          "cpConfig": {
            "cp1": {
              "cpProtocolData": [
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
                    "ipAddresses": [
                      {
                        "type": "IPV4",
                        "fixedAddresses": [
                          "172.16.0.10",
                          "172.16.0.11",
                          "172.16.0.12",
                          "172.16.0.13"
                        ]
                      }
                    ]
                  }
                }
              }
            }
          ]
        }
      ]
    }
  ]
  …
}
```

The same pool of IP addresses can also be created by specifying them as individual fixedAddresses in multiple cpProtocolData:

```
{
  …
  "extVirtualLinks": [
    {
      "id": "extVL-dbf477ad-199a-47ff-939a-cb0101c92585",
      "resourceId": "ext-net",
      "extCps": [
        {
          "cpdId": "ecp_1_vdu_node_1",
          "cpConfig": {
            "cp1": {
              "cpProtocolData": [
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
                    "ipAddresses": [
                      {
                        "type": "IPV4",
                        "fixedAddresses": [
                          "172.16.0.10"
                        ]
                      }
                    ]
                  }
                },
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
```

```
                    "ipAddresses": [
                      {
                        "type": "IPV4",
                        "fixedAddresses": [
                          "172.16.0.11"
                        ]
                      }
                    ]
                  }
                },
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
                    "ipAddresses": [
                      {
                        "type": "IPV4",
                        "fixedAddresses": [
                          "172.16.0.12"
                        ]
                      }
                    ]
                  }
                },
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
                    "ipAddresses": [
                      {
                        "type": "IPV4",
                        "fixedAddresses": [
                          "172.16.0.13"
                        ]
                      }
                    ]
                  }
                }
              ]
            }
          ]
        }
      ]
    }
  ]
  …
}
```

The same pool of IP addresses created using an addressRange:

```
{
  …
  "extVirtualLinks": [
    {
      "id": "extVL-dbf477ad-199a-47ff-939a-cb0101c92585",
      "resourceId": "ext-net",
      "extCps": [
        {
          "cpdId": "ecp_1_vdu_node_1",
          "cpConfig": {
            "cp1": {
              "cpProtocolData": [
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
                    "ipAddresses": [
```

```
                        {
                          "type": "IPV4",
                          "addressRange": {
                            "minAddress": "172.16.0.10",
                            "maxAddress": "172.16.0.13"
                          }
                        }
                      ]
                    }
                  }
                ]
              }
            }
          }
        ]
      }
    ]
    …
}
```

The implementation of these IP address pools conforms to the *ETSI NFV MANO SOL003* specification, *chapter 4.4.1.10*.

Static MAC Addresses

If the VNFC has connection points configured with a static MAC address, the VNFC cannot scale as there are no further MAC addresses to assign to the connection points on the newly spun up VNFC instances. Instead, a pool of further static MAC addresses can be specified in the instantiate request or grant response.

Static MAC address pools can be created in the extVirtualLinks element of the instantiate request or grant response by specifying the macAddress in multiple cpProtocolData.

The following example shows how to create a static MAC pool with four MAC addresses by specifying them in multiple cpProtocolData:

```
{
  …
  "extVirtualLinks": [
    {
      "id": "extVL-dbf477ad-199a-47ff-939a-cb0101c92585",
      "resourceId": "ext-net",
      "extCps": [
        {
          "cpdId": "ecp_1_vdu_node_1",
          "cpConfig": {
            "cp1": {
              "cpProtocolData": [
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
                  "ipOverEthernet": {
                    "macAddress": "fa:16:3e:0b:10:10",
                    "ipAddresses": [
                      {
                        "type": "IPV4",
                        "fixedAddresses": [
                          "172.16.0.10"
                        ]
                      }
                    ]
                  }
                },
                {
                  "layerProtocol": "IP_OVER_ETHERNET",
```

```
                    "ipOverEthernet": {
                      "macAddress": "fa:16:3e:0b:10:11",
                      "ipAddresses": [
                        {
                          "type": "IPV4",
                          "fixedAddresses": [
                            "172.16.0.11"
                          }
                        }
                      ]
                    }
                  },
                  {
                    "layerProtocol": "IP_OVER_ETHERNET",
                    "ipOverEthernet": {
                      "macAddress": "fa:16:3e:0b:10:12",
                      "ipAddresses": [
                        {
                          "type": "IPV4",
                          "fixedAddresses": [
                            "172.16.0.12"
                          ]
                        }
                      ]
                    }
                  },
                  {
                    "layerProtocol": "IP_OVER_ETHERNET",
                    "ipOverEthernet": {
                      "macAddress": "fa:16:3e:0b:10:13",
                      "ipAddresses": [
                        {
                          "type": "IPV4",
                          "fixedAddresses": [
                            "172.16.0.13"
                          ]
                        }
                      ]
                    }
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
    …
}
```

Day Zero Configuration

After deploying the VNFs, day 0 variables are configured in the VNFC instance for the deployment service. In most cases, the values for the day 0 configuration is constant. In other cases, there is a resource pool of values supplied to the day 0 parameter to allow new values to be assigned to the new VNFC instances.

Day 0 configuration within the vendor_section of the VNFD:

```
    vdu3:
      type: cisco.nodes.nfv.Vdu.Compute
      properties:
        name: 'Processing1'
        description: 'Processing VNFC'
        vdu_profile:
```

```
        min_number_of_instances: 1
        max_number_of_instances: 5
      vendor_section:
        cisco_esc:
          config_data:
            '/tmp/OSRESTTestETSIDay0_Inline_data.cfg':
              data: |
                NODE_NAME $NODE_NAME
                NUM_OF_CPU $NUM_OF_CPU
                MEM_SIZE $MEM_SIZE
                PROXY_ADDRS $PROXY_ADDRS
                SPECIAL_CHARS $SPECIAL_CHARS
              variables:
                NODE_NAME: vdu_node_1
                NUM_OF_CPU: 1
                MEM_SIZE: 1GB
                PROXY_ADDRS: ["1.1.1.1", "1.1.2.1", "1.1.3.1", "1.1.4.1", "1.1.5.1",
"1.1.6.1", "1.1.7.1"]
                SPECIAL_CHARS: '`~!@#$%^&*()-_=+[{]}|;:<.>/?'
```

In the above example the day 0 configuration is specified inline, with velocity variables defined in the target configuration. Each of these variables are supported by a variable with one or more values. In order to support multiple values for the $PROXY_ADDRS variable, a list of values are provided. These values are used to populate subsequent uses of the variable on new instances of the VNFC.

For information on day 0 configuration in the deployment data model, see Day Zero Configuration in the *Cisco Elastic Services Controller User Guide*.

Autoscaling of VNFs

KPIs, rules and actions defined in the VNFD determine the conditions under which scaling must be considered. The details are provided in Monitoring Virtual Network Functions. The scaling policies are also defined in the VNFD using several policy types that control the allowed scaling boundaries. These policy items are described below.

After deployment, ESC configures a monitoring agent (this may be the centralised or distributed instance) with the KPIs to monitor each VNFC. The scaling workflow begins if a KPI reaches its threshold; based on the action defined, ESC performs scale in or scale out and generates appropriate notifications and event logs. This is subject to some built-in functions that can be specified such as `log` or an onboarded script.

ESC sends appropriate notifications to the subscribed consumers. At this time, ESC interrogates the VNF instance resource for the *isAutoscaleEnabled* flag (this is set initially by the value in the VNFD but can be modified after creation). If this flag is set to true, ESC invokes the scaling workflow (instigated using a *ScaleVnfToLevelRequest* to request the scaling of multiple aspects in a single request). If the *isAutoscaleEnabled* is set to false, then the control is with an external system such as an NFVO or EM to trigger the desired action using the requests described above.

Note While creating an auto scaling or auto healing request, any new external requests are blocked. The user is notified of the corresponding response and problem details of the blocked request.