K-ONE 기술 문서 #6
OPNFV High Availability Clustering service for VNF Manager and Orchestrator

Document No. K-ONE #6
Version 1.0
Date 2016-4-30
Author(s) Do Truong Xuan

■ 문서의 연혁

버전	날짜	작성자	비고
0.1	Feb 7 2016	Do Truong Xuan	Chapter 1
0.2	Mar 15 2016	Do Truong Xuan	Chapter 2
1.0	April 30 2016	Do Truong Xuan	Chapter 3,4

본 문서는 2015년도 정부(미래창조과학부)의 재원으로 정보통신 기술진흥센터의 지원을 받아 수행된 연구임 (No. B0190-15-2012, 글로벌 SDN/NFV 공개소프트웨어 핵심 모듈/기능 개발)

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. B0190-15-2012, Global SDN/NFV OpenSource Software Core Module/Function Development)

기술문서 요약

In this technical document, we present about our automated clustering manager for orchestration in OPNFV platform. This is our very first achievement targeting at high availability and live migration mechanism in OPNFV platform. This manager is used to support high availability for stateless virtual function in the automated manner. First, we introduce about OPNFV which is an platform for orchestrating and managing virtual network functions at the carrier grade. We also introduce about ongoing related projects such as requirement project High availabitity and implementation projects Senlin, ceilometer. Then, we present about our detailed design of our automated clustering manager. Last, we show our demo results.

Contents

K-ONE 기술 문서 #6 OPNFV High Availability Clustering service for VNF Manager and Orchestrator

그림 목차

표 목차

1. Overview about OPNFV

1.1. Network Function Virtualization

Modern telecoms networks contain an ever increasing variety of proprietary hardware. The launch of new services or network reconfiguration demands the installation of yet more equipment that in turn requires additional floor space, power and trained maintenance staff. As the innovation cycles continue to accelerate, hardware-based appliances rapidly reach end of life. Simply having a hard-wired network with boxes dedicated to single functions is not the optimal way to achieve dynamic service offerings. Network design must be more agile and able to respond on-demand to the dynamic needs of the traffic and services running over it. Key enabling technologies for this include SDN (Software Defined Networking) and NFV (Network Functions Virtualisation), two complimentary concepts that are being developed by both the IT and the telecoms industries.

In November 2012 seven of the world's leading telecoms network operators selected ETSI to be the home of the Industry Specification Group for NFV. Now three years on, we see a large community of expert working intensely to develop the required standards for Network Functions Virtualisation as well as sharing their experiences of NFV development and early implementation. The membership of ISG NFV has grown to over 270 individual companies including 38 of the world's major service providers as well as representatives from both telecoms and IT vendors.

Figure 1 shows the architectural framework of NFV defined by ESTI

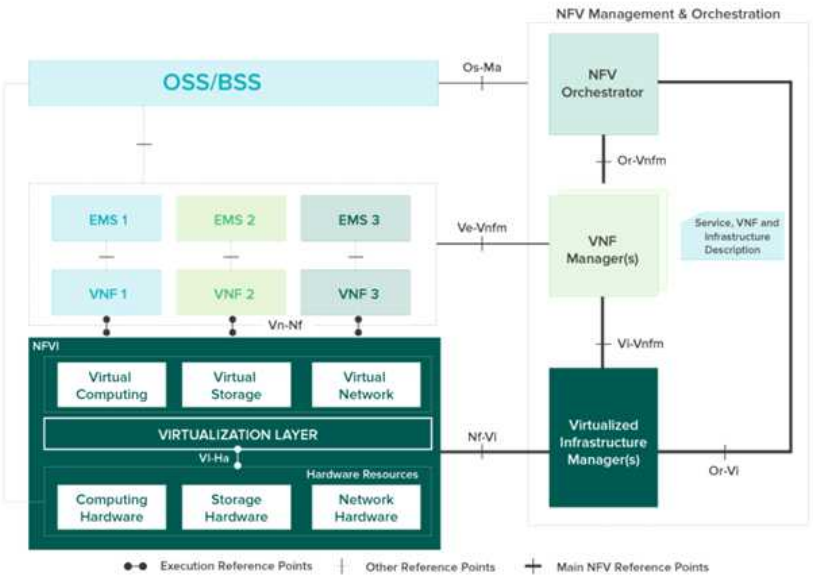

Figure 1. NFV architecture

1.2. OPNFV Project

1.2.1. Scope

OPNFV is a carrier-grade, integrated, open source platform to accelerate the introduction of new NFV products and services. Goals include accelerating time to market for NFV solutions, ensuring the platform meets the industry's needs, and enabling end user choice in specific technology components based on their use cases and deployment architectures. Creating a strong foundation for the OPNFV baseline platform is essential in ensuring that the industry has a good basis to build on top of moving forward. OPNFV looks to realize the ETSI NFV ISG's architectural framework by bringing together upstream software components to implement an end-to-end platform for NFV. As a testing and integration project, OPNFV brings together upstream components across compute, storage and network virtualization in order create an end-to-end platform. Activities within OPNFV focus on integration of components and automated build and deployment of the integrated environment. Continuous integration and automated testing of the platform for key NFV use cases is key to ensure the platform meets NFV industry needs. Defining new requirements and features

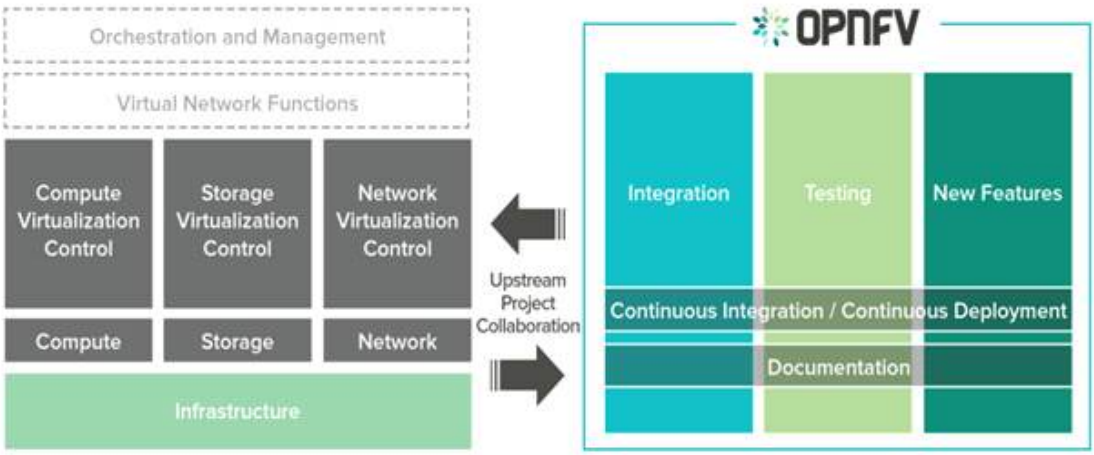

Figure 2. OPNFV platform

necessary to the platform and working closely with upstream communities to incorporate these capabilities within their code bases is also an important area of focus.

1.2.2. B release

The second release of OPNFV is Brahmaputra, which is a lab-ready release focused on providing a set of testing scenarios that can be utilized and adapted by developers for developing NFV services and Virtual Network Function (VNF) applications.

The following diagram shows an architectural view of Brahmaputra. The left side of the diagram highlights upstream components in Brahmaputra along with the community lab infrastructure, where users can test the platform in different environments and on different hardware. The right side of the diagram shows representative capabilities in the areas of integration, testing, and adding features to services and applications. These capabilities are added to OPNFV via approved OPNFV projects.

1.2.3. Use cases

Virtual network functions range from mobile deployments, where mobile gateways (e.g. SGW, PGW, etc.) and related functions (e.g. MME, HLR, PCRF, etc.) are deployed as VNFs, to deployments with "virtual" customer premise equipment (CPE), tunneling gateways (e.g. VPN gateways), firewalls or application level gateways and filters (e.g. web and email traffic filters) to test and diagnostic equipment (e.g. SLA monitoring). These VNF deployments need to be easy to operate, scale, and evolve – independently from the type of VNF being deployed. OPNFV aims to create a flexible platform, which can support a set of qualities and use-cases such as the following:

The common mechanism for life-cycle management of VNFs, which include deployment, instantiation, configuration, start and stop, upgrade/downgrade and final decommissioning.

The consistent mechanism for specifying and interconnecting VNFs, VNFCs and PNFs; agnostic of the physical network infrastructure, network overlays, etc., i.e., virtual link.

The common mechanism for dynamically instantiating new VNF instances or decommissioning sufficient VNF instances to meet the current performance, scale and network bandwidth needs.

The mechanism for detecting faults and failure in the NFVI, VIM and other components of the infrastructure and recovering from those failures.

The mechanism for sourcing/sinking traffic from/to a physical network function to/from a virtual network function.

NFVI as a Service for hosting different VNF instances from different vendors on the same infrastructure

1.3. High Availability Project

This project is focused on the high availability requirements of the OPNFV platform, with regards to the Carrier Grade NFV scenarios. In this project, we address HA requirements and solutions in 3 different perspectives; the hardware

HA, the virtual infrastructure HA and the service HA, to be specific. Requirement and API definition of high availability of OPNFV will be output from this project.

Scenario Analysis for HA in NFV:

	VNF Statefullness	VNF Redundancy	Failure detection	Use Case
VNF	yes	yes	VNF level only	UC1
			VNF & NFVI levels	UC2
		no	VNF level only	UC3
			VNF & NFVI levels	UC4
	no	yes	VNF level only	UC5
			VNF & NFVI levels	UC6
		no	VNF level only	UC7
			VNF & NFVI levels	UC8

Figure 3. High availability scenarios

2. Related opensource projects: Senlin

2.1. Features of Senlin

· A generic clustering/collection service for managing groups of homogeneous cloud objects on OpenStack.
· A set of APIs for managing cluster membership, e.g. add/remove nodes.
· A plugin-based object profile management enabling the creation and management of any object pools.
· A plugin-based policy enforcement framework featuring flexible policy customization for cluster management.
· A asynchronous execution engine for ensuring the state consistency of clusters and nodes.
· A open design for action execution that can be extended to accommodate complex application deployment.

2.2. Architecture of Senlin

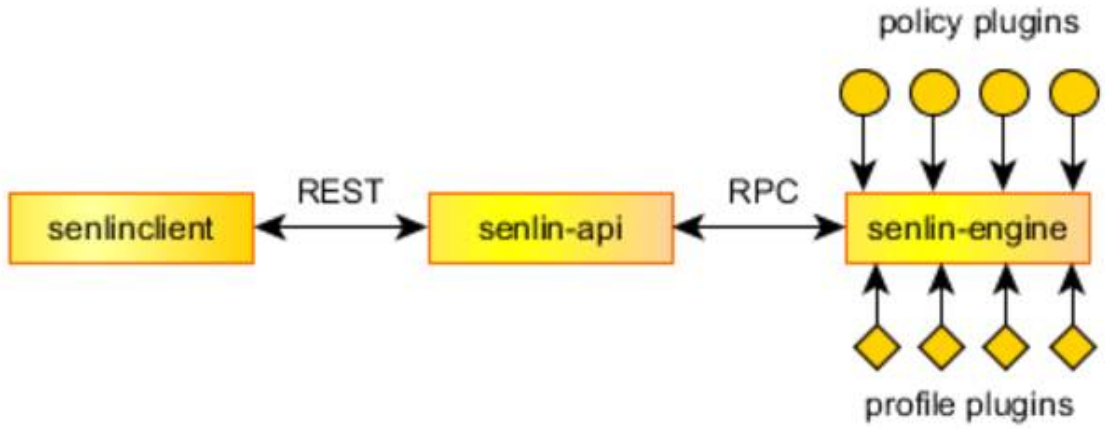

Figure 4. Senlin architecture

The senlinclient component provides the command line interface (CLI) for users to interact with the Senlin service;
The senlin-api service provides the RESTful APIs for the senlinclient component or other services;
The senlin-engine service sits behind the senlin-api service and operates on the clusters/nodes, using profiles and policies that are loaded as plugins.

· Cluster

Senlin provides a clustering service to create and manage objects of the same nature, e.g. Nova servers, Heat stacks, Cinder volumes, etc. The collection of these objects is referred to as a cluster.

· Nodes

Senlin abstracts the physical objects as nodes, which can belong to any cluster of the same profile type.

· Profiles

Senlin supports object creation, deletion and update via a concept called Profile. Each profile is in essential a driver to communicate with certain services for object manipulation.

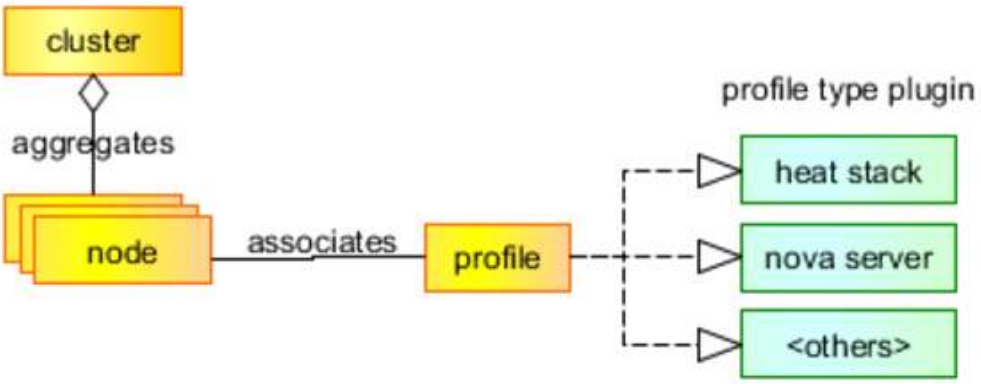

Figure 5. Senlin Profile

· Policies

With Senlin, user can attach a policy to a cluster and enforce it, or detach it from a cluster. Some built-in policies are provided to meet requirements such as auto-scaling, load-balancing, high-availability etc.

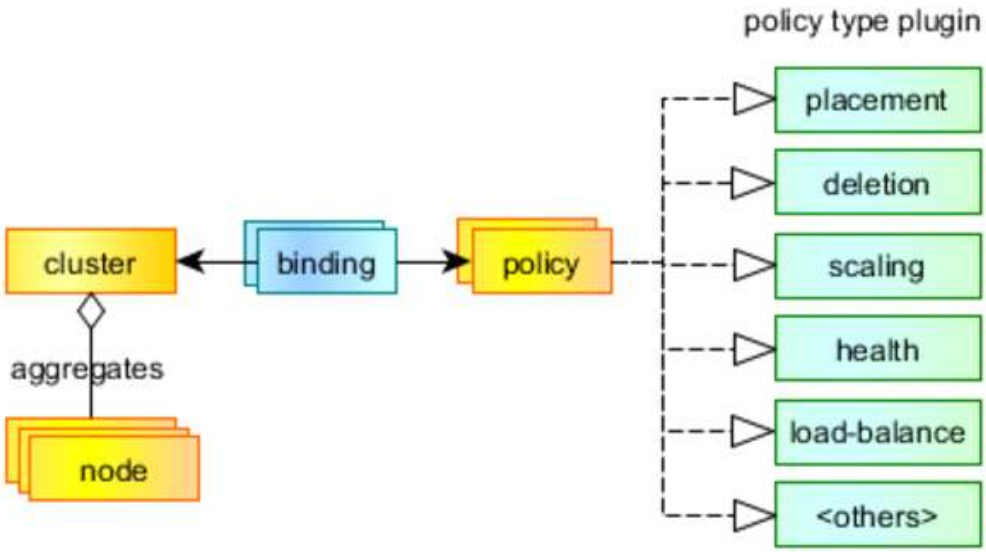

Figure 6. Senlin policy

3. Design of automated clustering manager for orchestration

3.1. Overview

Figure 7 shows the relation of opensource projects and ESTI reference architecture. Currently Openstack is selected as candidate for Network function virtualization infrastructure (NFVI) and virtual infrastructure manager (VIM). Openstack has some basic projects in charge of basic functions of VIM such as Ceilometer for monitoring virtual resources, heat for orchestration, nova for

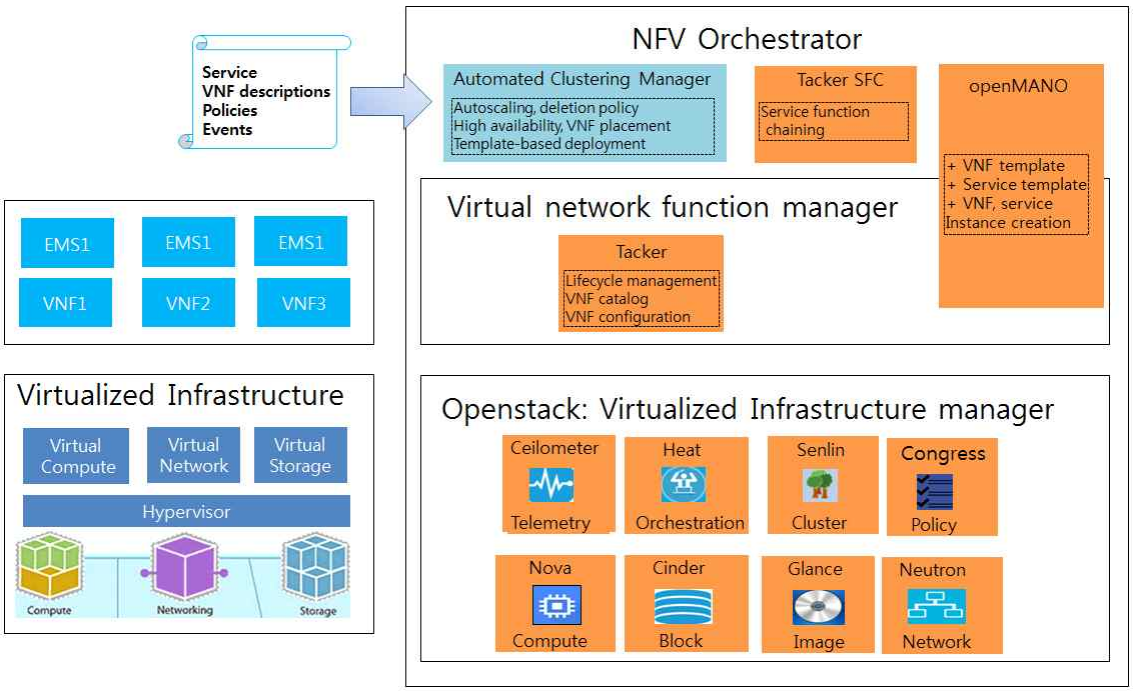

Figure 7. Automated Clustering manager

virtualized processor, neutron for network, and Senlin for clustering service. At the level of Virtual network function manager (VNF Manager), there is Tacker in charge of life cycle management, VNF configuration. At the orchestration level, there OpenMaNO, TackerSFC in charge of deploying a network service, service function chaining. We introduce a automated clustering service manager which have following functions: provide automated deployment for a node cluster, support high availability of virtual network function, support autoscaling, VNF placement... This manager is automated based on template file. This manager communicates with Senlin APIs for clustering management.

3.2. Architecture of automated clustering manager for orchestration

Figure 8 show the architecture of automated clustering manager for orchestration. This manager consists of two main components: parser and Senlin driver. Parser is used to analyze the template files for input parameters such as profile definition, cluster size, policies, and receivers for making and managing the cluster. Senlin driver is used to communicate with Senlin api and senlin engine to invoke APIs for cluter management. In our architecture, ceilometer is used to monitor virtual resources and trigger some actions when some ultilization metrics are over threshold.

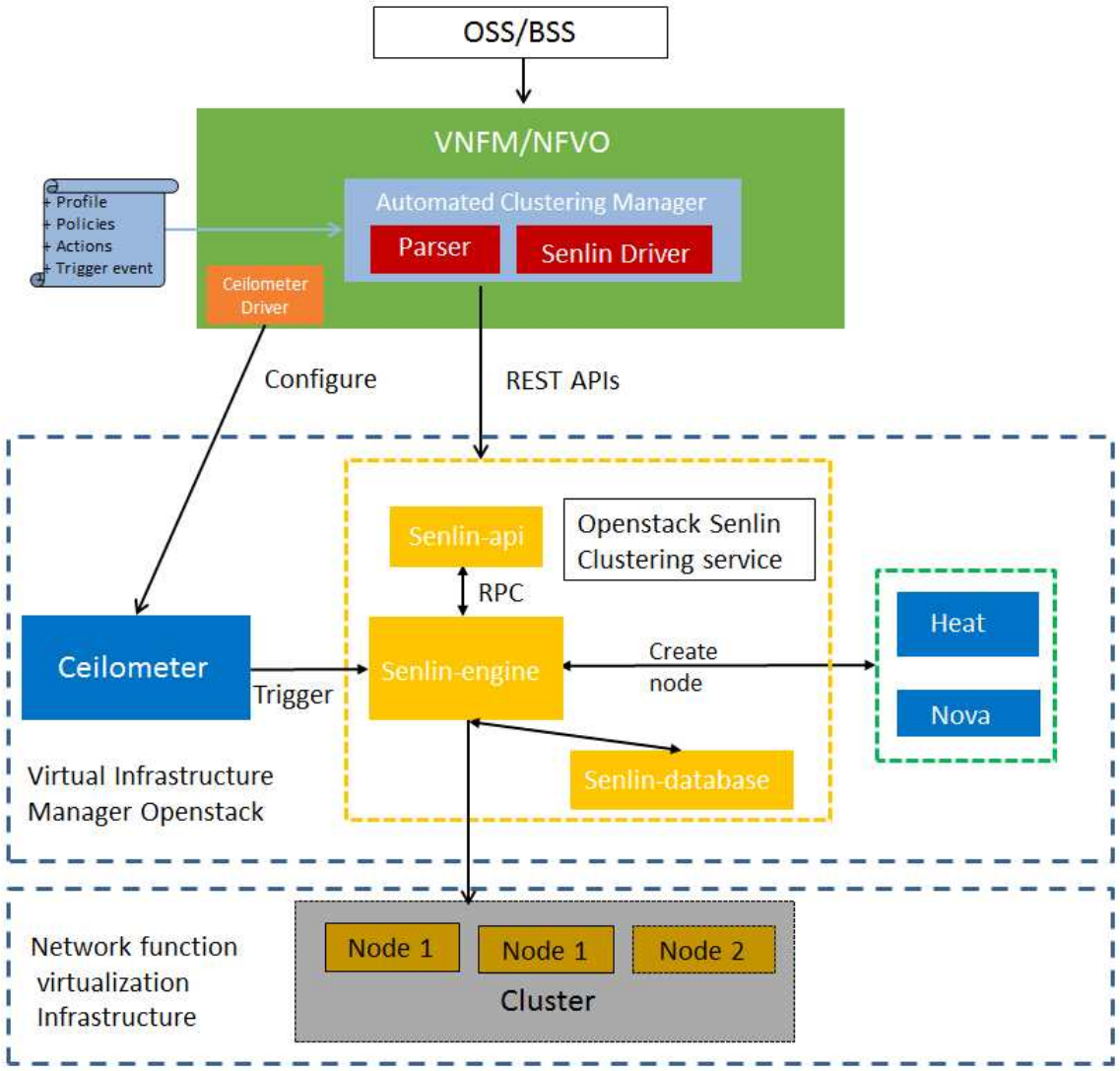

Figure 8. Automated clustering manager architecture

Following is template file definition (template.yaml)

template file for deploying a cluster of nodes

profile:
    name: // name of profile
    spec: // define a node is a nova server or heat stack
cluster:
    name: // name of cluster
    capacity: // initial number of node in a cluster
    min: // minimum number of node in a cluster
    max: // maximum number of node in a cluster

```
policies:  // define as many policies as you want
    - policy1:
        name:            // name of policy
        spec:            // definition of auto scaling policy
        enable:        // enable or not

triggers:            // define as many triggers as you want
    - trigger1:
        action:
            name:          // name of action
            value:      // action value
        receiver:
            name: // name of receiver
```

Following is table of APIs used in Senlin driver used to communicate with Senlin engine service

Table 2. APIs used in Senlin driver

APIs	**Meanings**
get_build_info()	Get build info for service engine and API
profile_types(***query*)	Get a generator of profile types.
get_profile_type(*profile_type*)	Get the details about a profile_type.
policy_types(***query*)	Get a generator of policy types.
get_policy_type(*policy_type*)	Get the details about a policy_type.
create_profile(***attrs*)	Create a new profile from attributes.
delete_profile(profile, *ignore_missing=True*)	Delete a profile.
find_profile(name_or_id, *ignore_missing=True*)	Find a single profile.
get_profile(*profile*)	Get a single profile.
profiles(***query*)	Retrieve a generator of profiles.
update_profile(profile, ***attrs*)	Update a profile.
create_cluster(***attrs*)	Create a new cluster from attributes.
delete_cluster(cluster, *ignore_missing=True)*	Delete a cluster.
find_cluster(name_or_id, *ignore_missing=True*)	Find a single cluster.
get_cluster(*cluster*)	Get a single cluster
update_cluster(cluster, ***attrs*)	Update a cluster.

cluster_add_nodes(cluster, *nodes*)	Add nodes to a cluster.
cluster_del_nodes(cluster, *nodes*)	Remove nodes from a cluster.
cluster_scale_out(cluster, *count=None*)	Inflate the size of a cluster.
cluster_scale_in(cluster, *count=None*)	Shrink the size of a cluster
cluster_attach_policy(cluster, policy, ***params*)	Attach a policy to a cluster.
cluster_detach_policy(cluster, *policy*)	Attach a policy to a cluster.
create_node(***attrs*)	Create a new node from attributes.
delete_node(node, *ignore_missing=True*)	Delete a node.
check_node(node, ***params*)	check a node.
get_node(node, *args=None*)	Get a single node.
update_node(node, ***attrs*)	Update a node.
create_policy(***attrs*)	Create a new policy from attributes.
delete_policy(policy, *ignore_missing=True*)	Delete a policy
get_policy(*policy*)	Get a single policy.
policies(***query*)	Retrieve a generator of policies.
get_cluster_policy(cluster_policy, *cluster*)	Get a cluster-policy binding.
create_receiver(***attrs*)	Create a new receiver from attributes.
delete_receiver(receiver, *ignore_missing=True*)	Delete a receiver.
get_receiver(*receiver*)	Get a single receiver.
get_action(*action*)	Get a single action.
get_event(*event*)	Retrieve a generator of events.

4. Demo results

4.1. Demo scenarios

Here to test our program, we use two scenarios: one is for autoscaling action and second is deletion action. We make a template file as following

```
profile:
    name: test_profile
    spec: nova_server.yaml
cluster:
    name: test-cluster
    capacity:
    min:
    max: 3
policies:
    - policy1:
        name: test-policy1
        spec: scaling_policy.yaml
```

```
        enable: TRUE
    - policy2:
        name: test-policy2
        spec: deletion_policy.yaml
        enable: FALSE
triggers:
    - trigger1:
        action:
            name: test-action1
            value: CLUSTER_SCALE_OUT
        receiver:
            name: test-receiver1
    - trigger2:
        action:
            name: test-action2
            value: CLUSTER_DEL_NODES
        receiver:
            name: test-receiver2
```

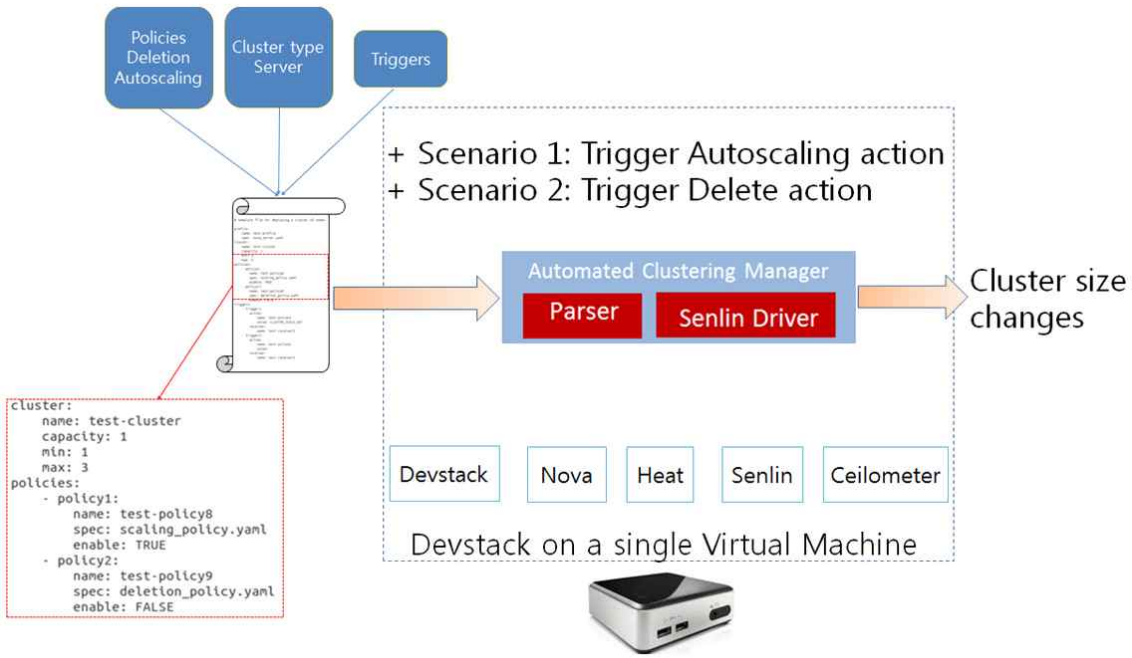

Figure 9. Demo scenarios

4.2. Results

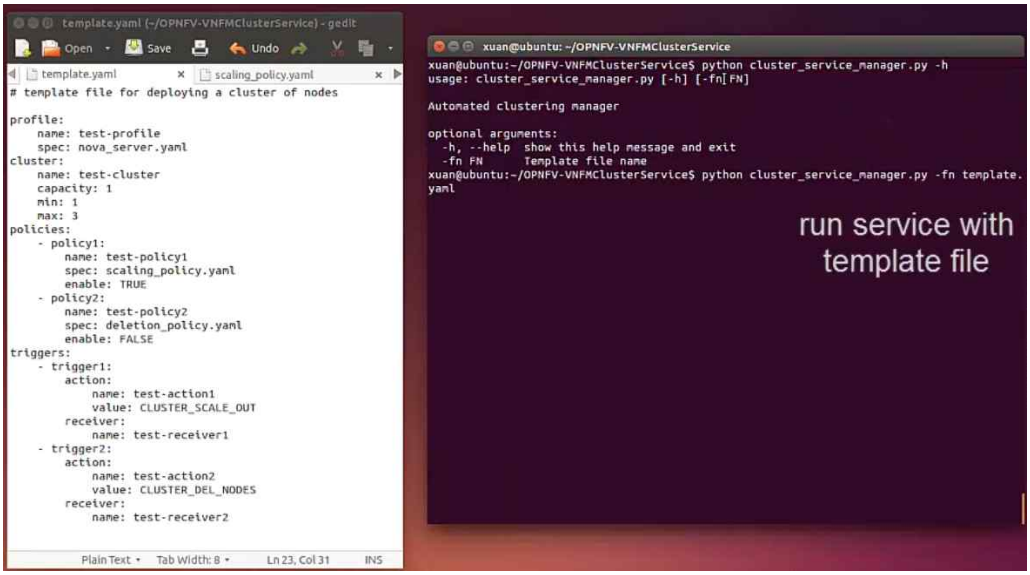

Figure 10. CLI of program

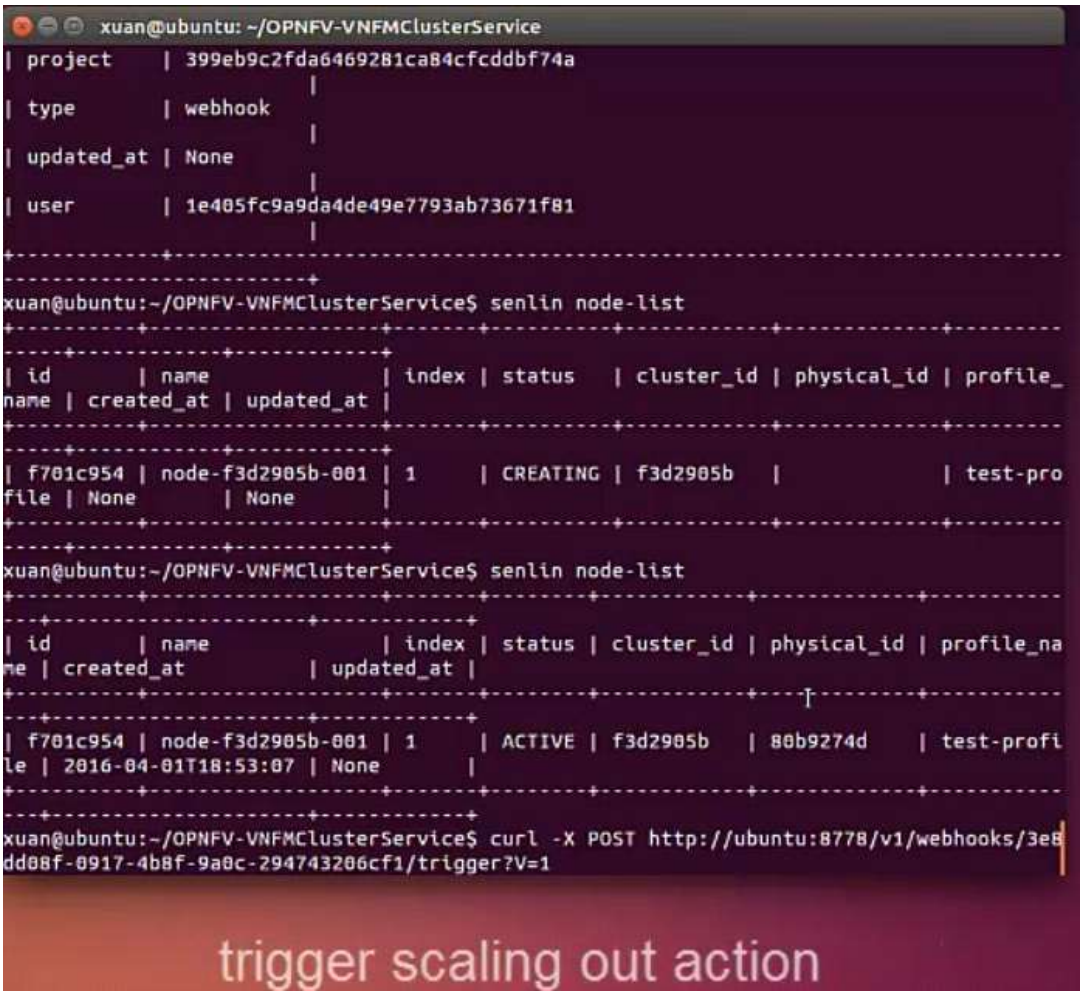

Figure 11. Before scaling out

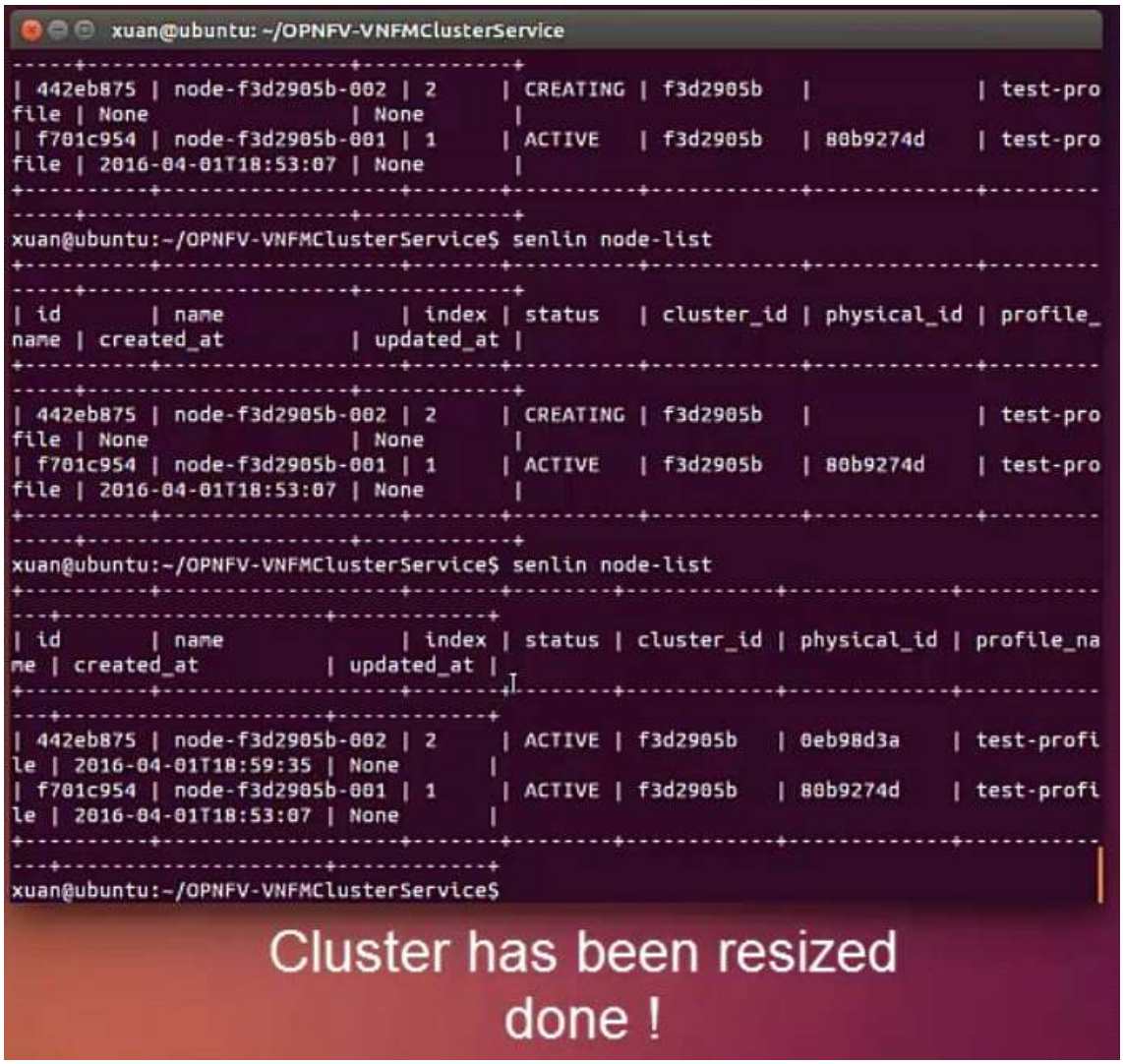

Figure 12. After scaling out