K-ONE Technical Document #19

The Study of architecture for supporting VNF High Availability in OpenStack Tacker

Document No.	**K-ONE #19**
Version	**0.2**
Date	**2017-05-12**
Author(s)	**Hyunsik Yang, Do troung Xuan, Doan Van Tung**

K-ONE

ONK
OpenNetworkingKorea

History

Version	Date	Author(s)	Contents
Draft - 0.1	2017. 04. 30	SSU Team	Write first draft
0.2	2017. 05. 12	SSU Team	Fix document compatibility issue

본 문서는 2015년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 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)

Summary

Network function virtualization (NFV) provides flexible and scalable network services by leveraging software-based network appliances. In an NFV architecture, the virtual network function manager (VNFM) has a critical role in provisioning, configuring, and operating virtual network functions (VNFs). In addition, the VNF manager should provide other functions to guarantee high availability of network services, such as fault management, monitoring and auto scaling. Currently, many open source projects aim to implement these functions, but there is no standard method of monitoring and detecting failure of VNFs. Therefore, we review current research of High availability and describe what we done for HA in OpenStack environment.

Contents

K-ONE #1. K-ONE Technical Document

List of Figures

K-ONE #19. K-ONE Technical Document

1. Review of High Availability

1.1. Introduction

For a high availability system, failure detection, isolation and recovery should be automatic and reliable. NFV can provide support for a range of HA mechanisms, such as redundancy, heartbeats, data synchronization, clustering and periodic snapshots or lock-step shadow copies to provide stateful service recovery. The OSS/BSS may already have some level of HA designed into their provisioning of the VNFs and services. In initial stages as operators move NFs to NFV, they may leave these existing mechanisms in place, and use the NFV opaquely for abstraction of resource pool. However, as they design NFV-aware services, they can leverage HA mechanisms offered by NFV. To support HA Mechanism, ETSI also defined HA Definition and OPNFV also implement the part of the HA function in NFV architecture. To check the current status of HA research, we review the document that is related to HA, and review the implementation in OPNFV. Finally, we explained our result of HA in this document.

1.1.1. ETSI GS NFV-Reliability

The document ETSI GS NFV REL 3 presents some VNF protection schemes, with which the reliability of NFs is realized. VNF protection schemes investigated include traditional Active-Standby method, Active-Active method and load balancing method with state transfer among VNFs[1][2].

1. **Active-Standby method**

The Active-Standby method is one of the popular redundancy methods adopted in many high availability systems. This configuration is depicted in Figure 1. Note that the Active-Standby method is referred to as one of the 2N redundancy model (N=1) in "Service Availability Forum Application Interface Specification (SA Forum AIS)".

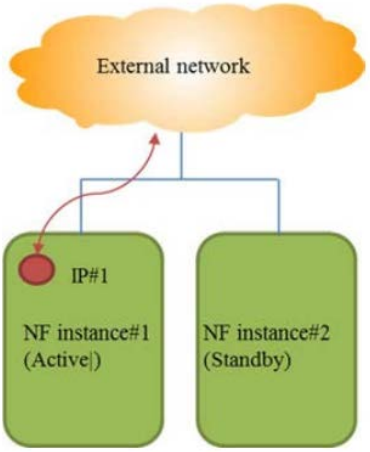


Figure 1 An active-standby configuration of NFs

State protection

Some NFs, such as stateful SIP proxies, use session state information for their operation. In this case, the active NF shares the state information with its standby NF to enable service continuity after a switch-over. The same principle applies for VNF Active-Standby protection schemes. There are two methods to store state information. The state information can be externalized or stored_at/shared_among peer mates. Restoration mechanisms with externalised state data have been described in ETSI GS NFV-REL 001. Hence, this clause focuses on the latter case where state information is stored at peer mates. Since shared state information needs to be kept consistent within the system, checkpointing or other methods will be used as described in ETSI GS NFV-REL 002[3].

Recovery and remediation phase

When an active NF instance fails, the standby NF instance takes over the active role so that the availability of the system is maintained. The possible main steps are as follows:

1) Failure detection of the active NF instance.

2) Failover from the former active NF instance to the former standby NF instance. The former standby instance becomes the new active instance.

3) Replacement of the failed NF instance with a new one which becomes the new standby NF

instance.

4) Start of the state information replication to the new standby NF instance.

Transparent failover in traditional deployments is enabled by assigning one logical IP address that the clients use to communicate with the system. In addition, each instance has its own fixed IP address used for designating each NF instance distinctively. During normal operation, the active NF instance serves the logical IP address. If the active NF instance fails and the failure is detected, the former standby NF instance will become active, and starts serving packets destined to the logical IP address. Hence, the system can continue providing the service.

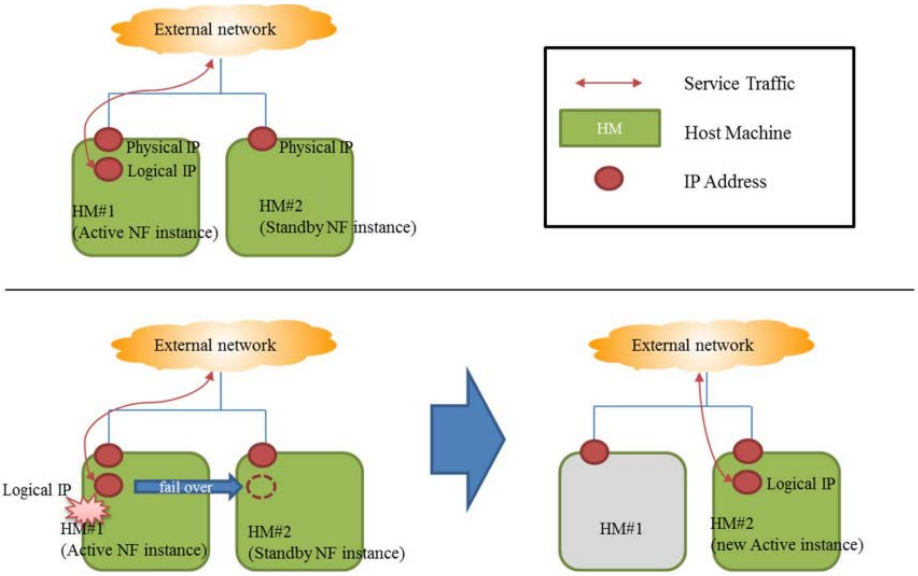


Figure 2 An example of the Active-Standby method in traditional environments

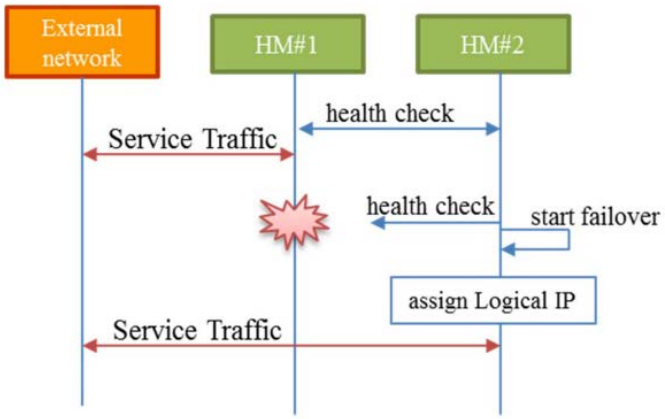


Figure 3 Failover flow for a traditional system with the Active-Standby method

In NFV deployments, the active VNF and clients on the external network are typically connected via NFVI (network infrastructure domain). This makes failover mechanism of virtualised systems different from that of the traditional ones. Since the interconnection between the active VNF and the next NF in the service chain is managed by NFV-MANO, NFV-MANO's availability influences the availability of the NS

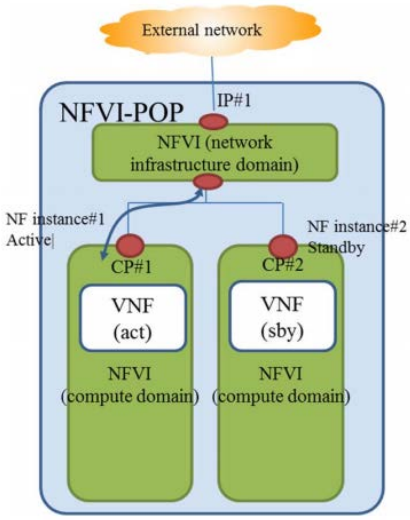


Figure 4 A typical Active-Standby configuration of VNFs in the NFV architecture

In this configuration, the assumption is that the active VNF and the standby VNF are deployed in a location-disjoint mode. For this, NFV-MANO need to support anti-affinity rules for deploying VNFs so that each VNF instance is installed on a different compute domain. Each

VNF instance has an internal connection point (CP), with an associated IP address. The two internal connection points are aggregated to a single connection point that is visible to systems on the external network. For example, a virtual router connects the IP address assigned to the external CP with the internal CP of the active VNF. Thereby, the active VNF can communicate with clients on the external network and can provide services. Like in traditional systems with Active-Standby protection, the standby VNF periodically checks the operational status of the active VNF, e.g. using a heartbeat mechanism, which enables failure detection of the active system within the required time for the service. Though NFV-MANO may also perform health check of the VNFs, it increases dependency on NFV-MANO to propagate information concerning the failure of an active VNF to the standby VNF via NFV-MANO. In case the active VNF fails, the standby VNF will initiate the fail-over procedure.

It is realised by reassigning the external CP to the internal CP of the former standby VNF.

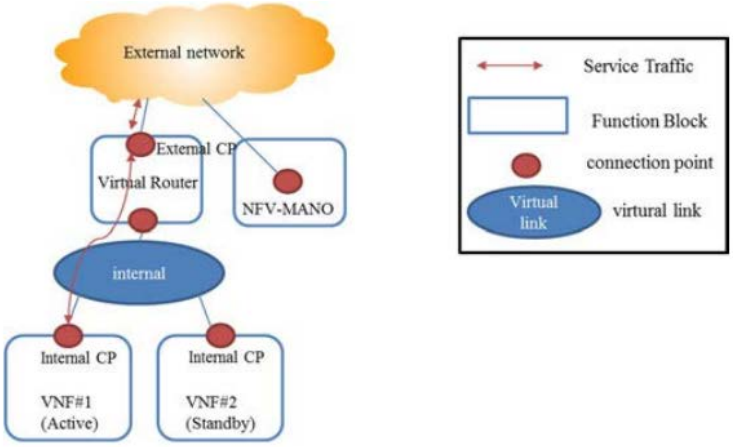


Figure 5 Active-Standby method in the virtualised environment

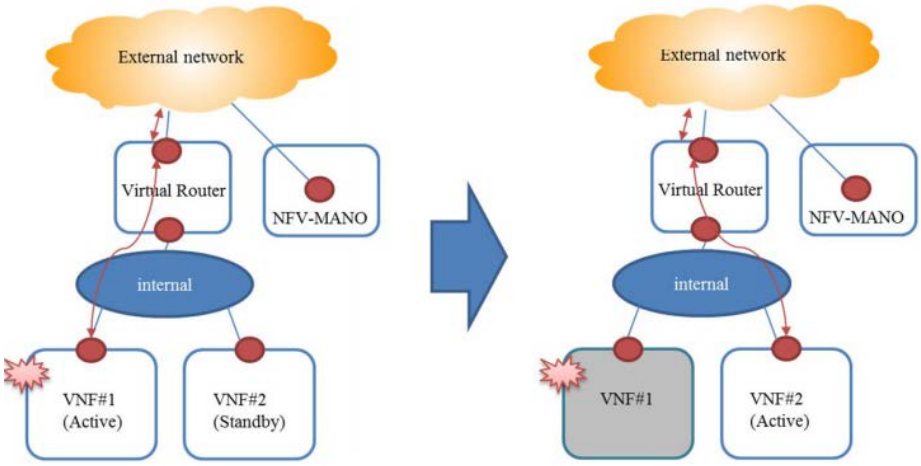


Figure 6 Active-Standby failover in the virtualised environment

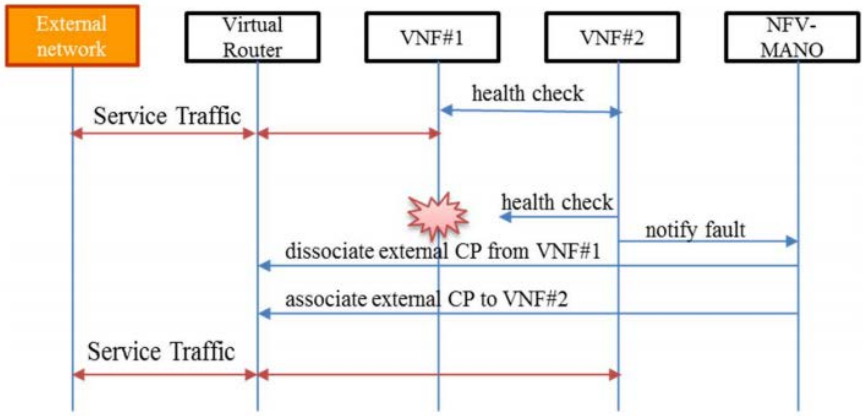


Figure 7 A failover procedure of VNFs in the Active-Standby configuration

In NFV environments, the mapping of the external CP to the internal CP is configured on the virtual router. This mapping cannot be reconfigured without NFV-MANO. Therefore, NFV-MANO should be informed about the failure of the active VNF, and associate the external CP with the internal CP of the former standby VNF instance. Moreover, it is also expected that NFV-MANO instantiates a new standby VNF. Note that the time required for changing the CP mapping affects the availability of the system, since the service is unavailable before reconfiguration, and that the time needed for instantiating a new standby VNF is related to the remediation period's duration. In addition to reconfiguring the CP mapping of the system,

further consideration to keep the system's state information consistent is required. Figure 8 shows VNFs configured with the Active-Standby method with state replication between VNFs. The active VNF stores session state information in a storage within itself, and replicates the information to the storage within the standby VNF. Figure 9 shows the restoration procedure for this setup. Since the new active VNF has to check the consistency of the backup state information to judge whether the service can be continued before receiving the subsequent service traffic flow, the route change request sent to NFV-MANO should be triggered by the new active (i.e. the former standby) VNF. For the remainder of this clause, only Active-Standby configurations with state synchronization are described since they are a superset of those configurations without state synchronization.

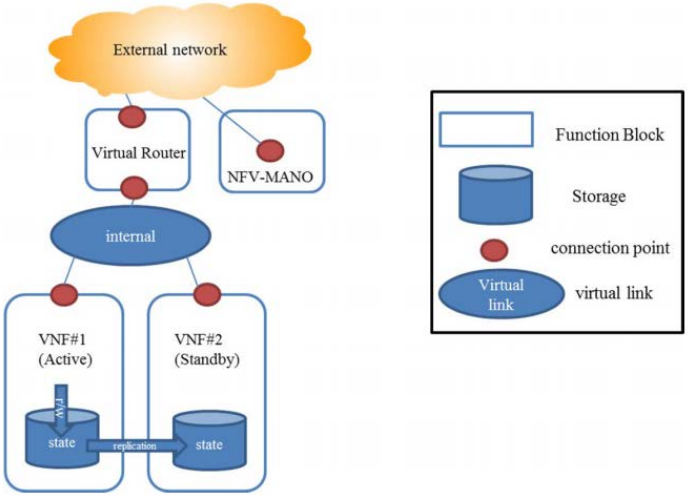


Figure 8 Stateful VNFs with internal storage for states in the Active-Standby configuration

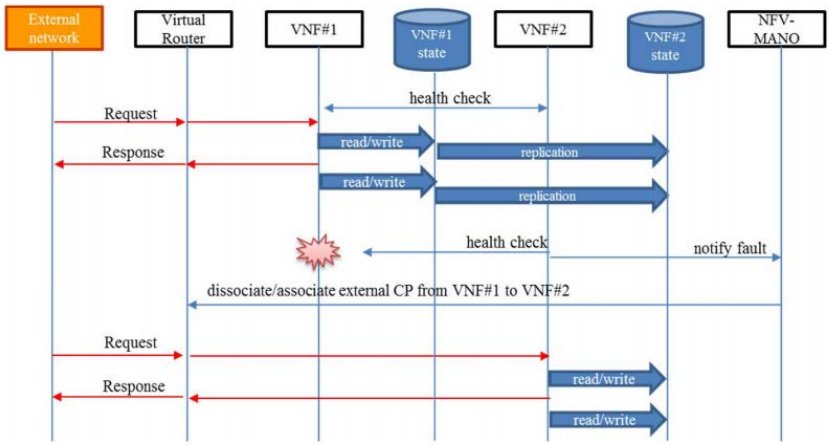


Figure 9 A failover procedure of VNFs synchronizing state information in the Active-Standby configuration

2. Active-Active method

The Active-Active method is a configuration where two NF instances are simultaneously active for the same service (Figure 10). The external network can receive the service by accessing either IP addresses of these NF instances. This configuration is referred to as an Active-Active redundancy configuration (N-way active redundancy model N=2) in the SA Forum AIS document.

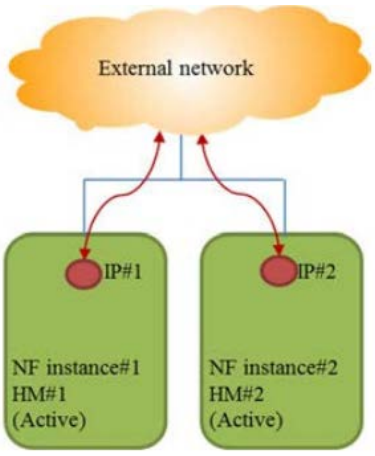


Figure 10 An Active-Active configuration of NFs

State protection

This configuration does not contain a standby system. Therefore, externalisation should be used to maintain the state information for service continuity in case of system failure. The same principle applies for VNF Active-Active protection schemes, and external storages provided by NFVI may be used for the state protection.

Recovery and remediation phase

Figure 11 depicts a configuration change when NF instance#1 fails in a traditional system. External NFs can receive services from NF instance #2 when no answer is received from the NF instance #1, by changing the destination of requests for the NF from IP#1 to IP#2, which is usually done by the DNS in which the NFs are assigned to a primary NF and to a secondary NF. Note that when a load balancer is placed in front of the NF instances, external NFs can get services without knowing the failure of NF instance#1, since the load balancer can deliver all the requests to the NF instance #2 when the NF instance #1 fails

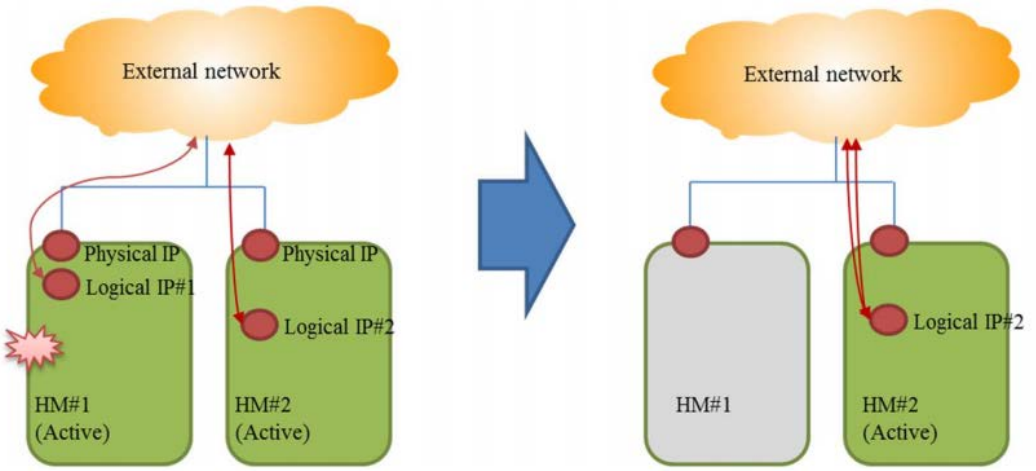


Figure 11 An example of remediation mechanism with Active-Active method in traditional physical environments

In NFV deployment, the active VNF and clients on the external network are connected via NFVI (network infrastructure domain) instead of using a logical IP address assigned to a virtual machine where the VNF is deployed. In this case, two VNFs are deployed on different compute domains according to anti-affinity rules. Two external CPs are assigned to the virtual router in a network infrastructure domain. Each external CP is connected respectively to an internal CP of each VNF. Figure 12 shows an example of Active-Active configuration in NFV environments.

In this figure, NFV-MANO does not have to reconfigure the mapping of external CP and internal CP, since the clients in the external network change the destination of the request message so as to be able to get services from the VNF instance #2

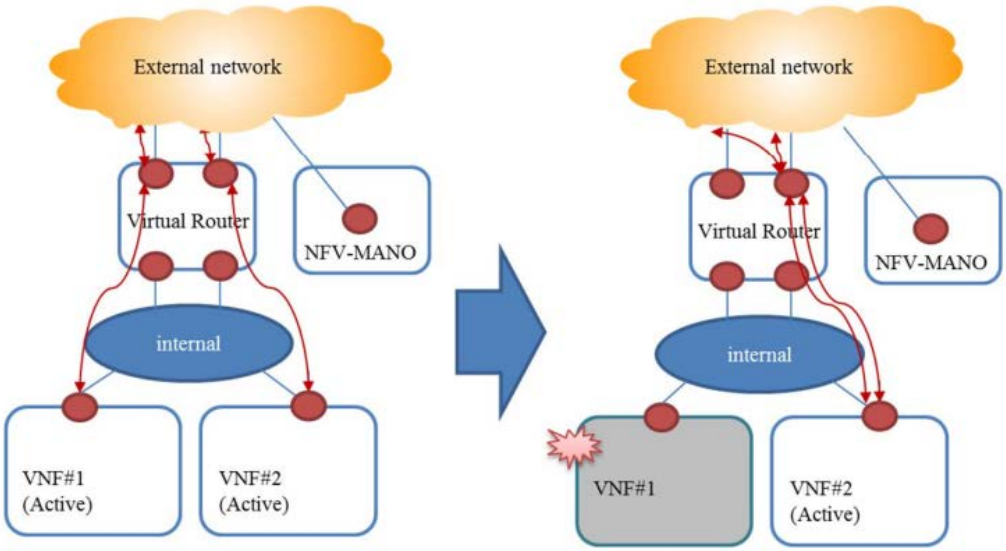


Figure 12 An example of Active-Active method in the NFV environment

3. Load balancing Method

Load balancing is a popular method used for Web servers. In cloud environments, a load balancer is deployed in front of several VNFs. The service traffic goes through the load balancer and is distributed to several VNFs that handle the whole traffic (Figure 13). A load balancing method that corresponds to the N-way redundancy model of the SA Forum AIS document is investigated in this clause.

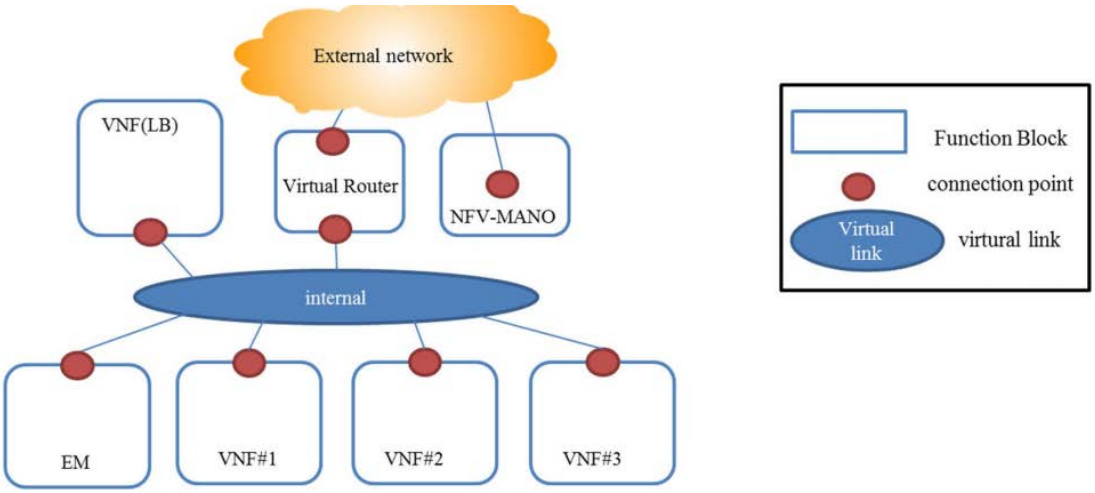


Figure 13 A load balancing model in virtualised environments

State protection

When a VNF is stateful, e.g. SIP proxies, there are two methods to store state information. One method is to externalize the state information. The other is to share the state information among peer mates, which means that the state information of the service instances within VNFs have their standby information in other VNFs as shown in Figure 14. In this figure, the state information of the service instances running on VNF#1 is replicated in VNF#2 as a backup. Similarly, the one of service instances on VNF#2 (resp. #3) is replicated on VNF#3 (resp. #1)

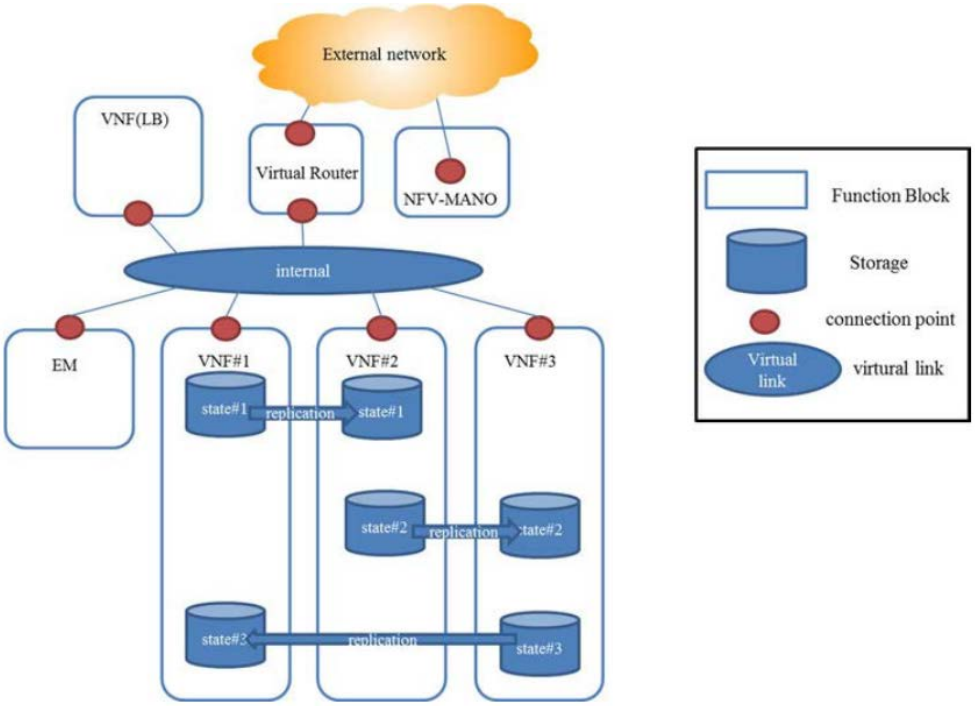


Figure 14 Load balancing model with internal storage

Restoration mechanisms with externalized state data have been described in ETSI GS NFV-REL 001. Hence, this clause mainly focuses on the latter case where state information is stored at peer mates. Since shared state information is to be kept consistent within the system, checkpointing or other methods will be used as described in ETSI GS NFV-REL 002.

Remediation and recovery phases

As an assumption, a load balancer periodically checks the operational status of VNFs to which service traffic is delivered to avoid delivering traffic to the failed VNF. When the load balancer detects the failure of a VNF, it stops providing traffic to the failed VNF and delivers it to other VNFs so that service continuity is maintained (Figure 15). When VNFs are stateful, the traffic to the failed VNF has to be delivered to the VNF that stores the state information of the failed service instances as a backup. The standby service instances of the VNF continue the service with the backup information. In some cases, the standby service instances recognize the failure of the corresponding active service instances so as to take over the ongoing services. A possible way to do this is that VNFs check one another, so that the standby service instances within the VNF are aware of the failure of their corresponding active service instances. Since

the external CP that is visible to clients on the external network is only assigned to the virtual load balancer, NFV-MANO does not have to explicitly change the configuration between the load balancer and VNFs for remediation. For recovery, NFV-MANO should deploy a new VNF that compensates the performance and the availability lost due to the failure.

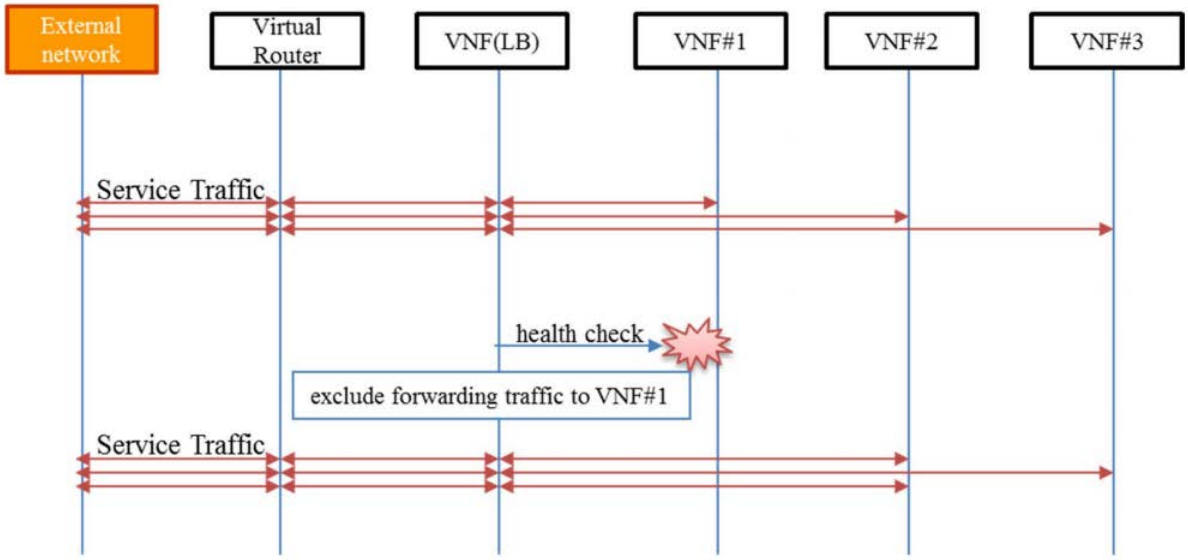


Figure 15 A sample procedure that will occur when a VNF with the service application fails

Note that a virtual load balancer should also be redundant, e.g. Active-Standby, to avoid single point of failure. For Active-Standby virtual load balancers, the same requirements for NFV-MANO as those derived from the investigation of the Active-Standby method.

1.1.2. NFV Monitoring framework

In the ETSI standard, the fault-management interfaces are defined in terms of the management of fault monitoring at the NFVI level. Fig. 16 shows the three ETSI-defined interfaces for the NFVI-fault management.

The Or-Vi reference point is used for exchanges between the NFVO and the VIM, and it also supports the fault management. This interface supports the collection of the virtualized-resource fault information, the provision of the alarm notifications that are related to the virtualized-resource faults to the NFVO [4], and the provision of the alarm status (change/delete/create); in addition, it also defines the attributes of the alarm-information element.

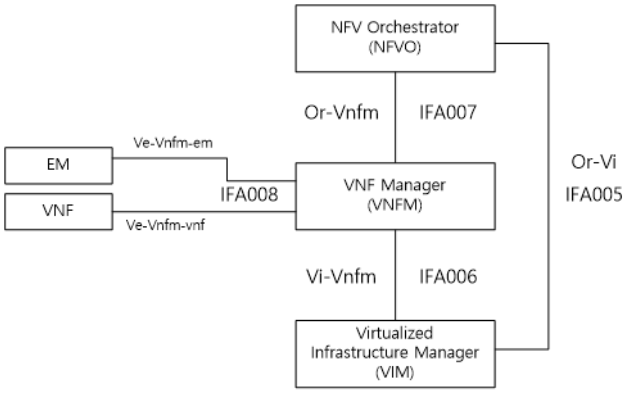


Figure 16 Fault-management interfaces of the ETSI

The fault type, state, and perceived severity have already been defined in this case. The fault type such as the CPU failure, memory failure, and network-card failure is related to the type resource. Through the VIM's use of a filter, the NFVO can obtain the specific virtualized-resource alarms that need to be reported.

The Vi-Vnfm reference point is used for exchange of information elements between the VIM and the VNFM, and it also supports the fault management [5]. This interface supports the collection of the virtualized-resource fault information, the provision of the alarm notifications that are related to the virtualized-resource faults to the VNFM, and the provision of the alarm status (change/delete/create) to the VNFM; also, the attributes that it defines are the same as those of the Or-Vi interface. The VNFM is also similar to the Or-Vi in that it obtains the specific alarms for the virtualized resources that need to be reported through the filter usage of the

VIM.

The ETSI standard also defines the fault-management interfaces for the management of the fault monitoring at the VNF level. At the VNF level, a fault is caused by an internal error such as a VNF-configuration fault, an inappropriate parameter, or communications problem with the VNFC. As shown in Fig. 16, two interfaces have been defined between the VNFM and the VNF and the VNFM and the EM[6].

The IFA008 comprises two interfaces. One is the Ve-Vnfm-em, which is used between the VNFM and the EM, and the other is the Ve-Vnfm-vnf, which is used between the VNFM and the VNF. In this architecture, EM is the element management. The EM is responsible for the FCAPS (i.e., fault, configuration, accounting, performance, and security)-management functionality of a VNF. The EM can be aware of the virtualization and collaborates with the VNFM to perform those functions. The IFA008 is an interface that allows the VNFM to provide alarms that are related to the VNF(s) and its VNFC(s) and are visible to the consumer. The VNFM can collect the virtualized-resource alarms using a filter and can modify the collected information. The VNFM collects the fault information from the VNF or the EM using the IFA008 interface.

At first, the Ve-Vnfm-em supports the VNF fault-management interface that is produced by the VNFM. The Ve-Vnfm-em supports the collection of the VNF/VNFC fault information and provides the alarm notifications that are related to the fault on the VNF/VNFC. In addition, it supports the sending of a notification to the EM when an alarm regarding a VNF/VNFC instance has been created or cleared. When an alarm has been created due to a VNF/VNFC fault, it also provides the fault information such as the VNF/VNFC identity and the alarm cause. The Ve-Vnfm-vnf supports the VNF fault-management interface that is produced by the VNFM; the function of this interface is the same as that of the Ve-Vnfm-em, so an explanation of the Ve-Vnfm-em is not given here.

The management of faults at the VNF level, however, is not a simple process since the VNF can consist of many kinds of fault depending on the function and the service provider.

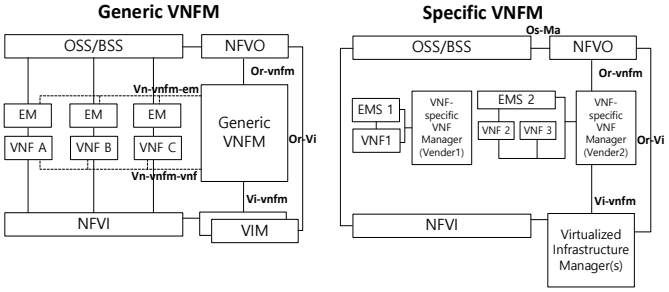


Figure 17 Fault-management interfaces of the ETSI

Moreover, the dependence between the VNFM and the VNF, or the VNF compatibility, should also be considered for the management of the VNF-fault management.

As described above, the VNF life cycle is managed by the VNFM; however, the VNFM can be changed by the service provider. Accordingly, as shown in Fig. 17, two NFV architectures are considered with respect to the service provider. One is the generic-VNFM model, and the other is the specific-VNFM model. Although, the ETSI standard defines the reference point and the interfaces for the VNFM, they can be changed based on the provider's architecture.

2. Opensource for High Availability Review

2.1. Opensource for High Availability

In this Chapter, we review the opensource project that related to HA such as OPNFV HA, MANO, SFC.

2.1.1. OPNFV 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[7].

Scenarios analysis

The VNF HA is divided into 8 use cases depending whether VNF is stateful or stateless, VNF redundancy, or failure source.

	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 18 VNF HA use cases

Use case 1: VNFC failure in a stateful VNF with redundancy

Use case 1 represents a statefull VNF with redundancy managed by an HA manager, which is part of the VNF. The VNF consists of VNFC1, VNFC2 and the HA Manager. The latter managing the two VNFCs, e.g. the role they play in providing the service named "Provided NF" .The failure happens in one of the VNFCs and it is detected and handled by the HA manager. On practice the HA manager could be part of the VNFC implementations or it could be a separate entity in the VNF. The point is that the communication of these entities inside the VNF is not visible to the rest of the system. The observable events need to cross the boundary represented by the VNF box.

Initially VNFC2 is active, i.e. Provides the Provided NF and VNFC1 is a standby. It is not shown, but it is expected that VNFC1 has some means to get the update of the state of the Provided NF from the active VNFC2, so that it is prepared to continue to provide the service in case VNFC2 fails. The sequence of events starts with the failure of VNFC2, which also interrupts the Provided NF. This failure is detected somehow and/or reported to the HA Manager, which in turn may report the failure to the VNFM and simultaneously it tries to isolate the fault by cleaning up VNFC2. Once the cleanup succeeds (i.e. the OK is received) it fails over the active role toVNFC1 by setting it active. This recovers the service, the Provided NF is indeed provided again. Thus this point marks the end of the outage caused by the failure that need to be considered from the perspective of service availability. The repair of the failed VNFC2, which might have started at the same time when VNFC1 was assigned the active state, may take longer but without further impact on the availability of the Provided NF service. If the HA Manager reported the interruption of the Provided NF to the VNFM, it should clear the error condition

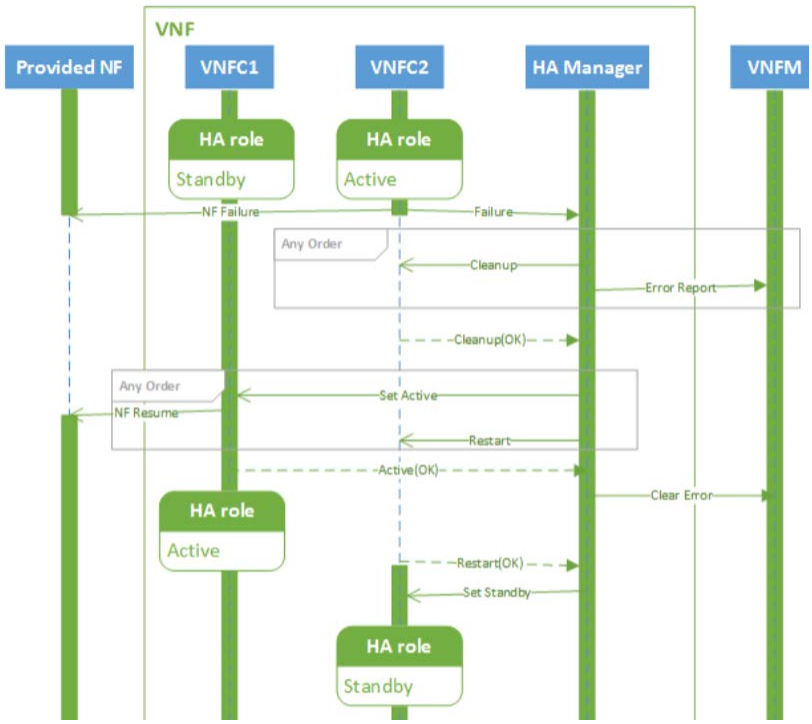


Figure 19 Sequence diagram for use case 1

Use case 2: VM failure in a stateful VNF with redundancy

Use case 2 also represents a statefull VNF with its redundancy managed by an HA manager, which is part of the VNF. The VNFCs of the VNF are hosted on the VMs provided by the NFVI (Fig 19).The VNF consists of VNFC1, VNFC2 and the HA Manager (Fig 19). The latter managing the role the VNFCs play in providing the service - Provided NF. The VMs provided by the NFVI are managed by the VIM. In this use case it is one of the VMs hosting the VNF fails. The failure is detected and handled at both the NFVI and the VNF levels simultaneously. The coordination occurs between the VIM and the VNFM. Again initially VNFC2 is active and provides the Provided NF, while VNFC1 is the standby. It is not shown in Fig 19, but it is expected that VNFC1 has some means to learn the state of the Provided NF from the active VNFC2, so that it is able to continue providing the service if VNFC2 fails. VNFC1 is hosted on VM1, while VNFC2 is hosted on VM2 as indicated by the arrows between these objects in Fig 19. The sequence of events starts with the failure of VM2, which results in VNFC2 failing and interrupting the Provided NF.

The HA Manager detects the failure of VNFC2 somehow and tries to handle it the same way as in use case 1. However because the VM is gone the clean up either not initiated at all or interrupted as soon as the failure of the VM is identified. In either case the faulty VNFC2 is considered as isolated. To recover the service the HA Manager fails over the active role to VNFC1 by setting it active. This recovers the Provided NF. Thus this point marks again the end of the outage caused by the VM failure that need to be considered from the perspective of service availability. If the HA Manager reported the interruption of the Provided NF to the VNFM, it should clear the error condition. On the other hand the failure of the VM is also detected in the NFVI and reported to the VIM. The VIM reports the VM failure to the VNFM, which passes on this information to the HA Manager of the VNF. This confirms for the VNF HA Manager the VM failure and that it needs to wait with the repair of the failed VNFC2 until the VM is provided again. The VNFM also confirms towards the VIM that it is safe to restart the VM.

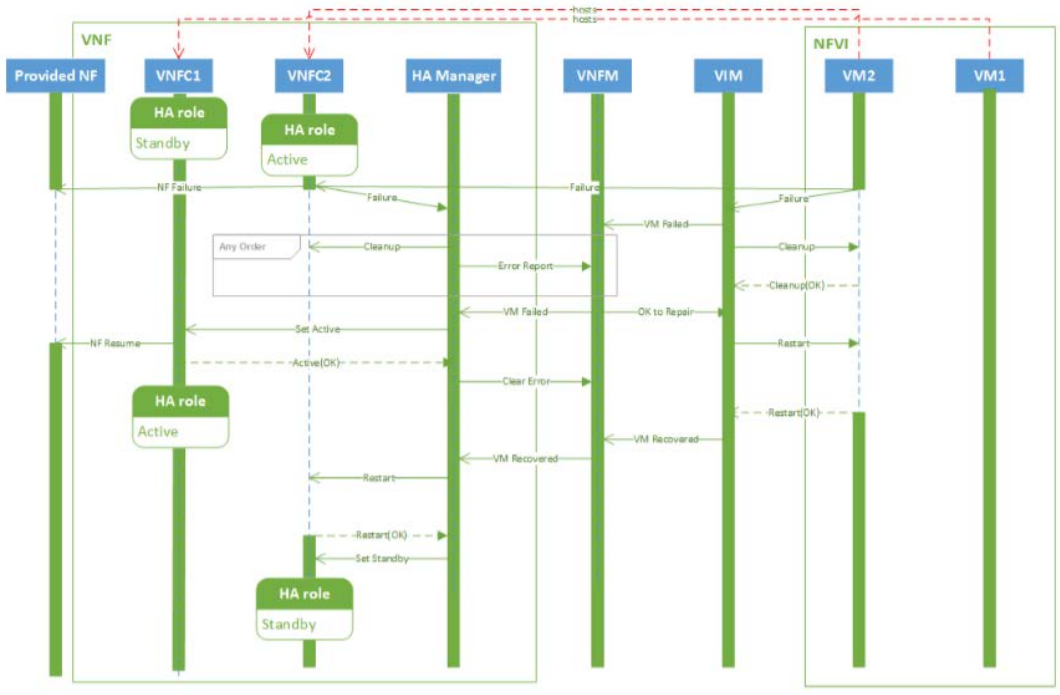


Figure 20 Sequence diagram for use case 2

Use Case 3: VNFC failure in a statefull VNF with no redundancy

Use case 3 also represents a statefull VNF, but it stores its state externally on a virtual disk provided by the NFVI. It has a single VNFC and it is managed by the VNFM. In this use case the VNFC fails and the failure is detected and handled by the VNFM. The VNFC periodically checkpoints the state of the Provided NF to the external storage, so that in case of failure the Provided NF can be resumed. When the VNFC fails the Provided NF is interrupted. The failure is detected by the VNFM somehow, which to isolate the fault first cleans up the VNFC, then if the cleanup is successful it restarts the VNFC. When the VNFC starts up, first it reads the last checkpoint for the Provided NF, then resumes providing it. The service outage lasts from the VNFC failure till this moment.

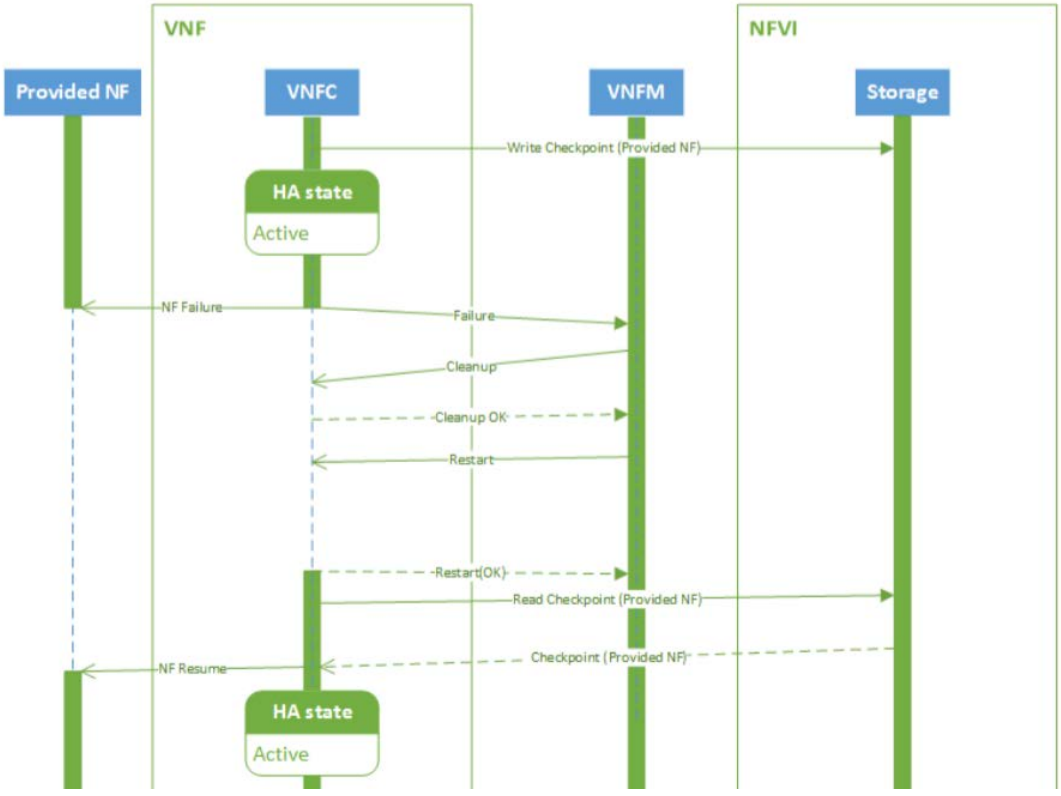


Figure 21 Sequence for use case 3

Use Case 4: VM failure in a statefull VNF with no redundancy

Use case 4 also represents a statefull VNF without redundancy, which stores its state externally on a virtual disk provided by the NFVI. It has a single VNFC managed by the VNFM as in use case 3.In this use case the VM hosting the VNFC fails and the failure is detected and handled by the VNFM and the VIM simultaneously. Again, the VNFC regularly checkpoints the state of the Provided NF to the external storage, so that it can be resumed in case of a failure. When the VM hosting the VNFC fails the Provided NF is interrupted. On the one hand side, the failure is detected by the VNFM somehow, which to isolate the fault tries to clean the VNFC up which cannot be done because of the VM failure. When the absence of the VM has been determined the VNFM has to wait with restarting the VNFC until the hosting VM is restored. The VNFM may report the problem to the VIM, requesting a repair. On the other hand the failure is detected in the NFVI and reported to the VIM, which reports it to the VNFM, if the VNFM hasn't reported it yet. If the VNFM has requested the VM repair or if it acknowledges the repair, the VIM restarts the VM. Once the VM is up the VIM reports it to the VNFM, which in turn can restart the VNFC. When the VNFC restarts first it reads the last checkpoint for the Provided NF, to be able to resume it. The service outage last until this is recovery completed.

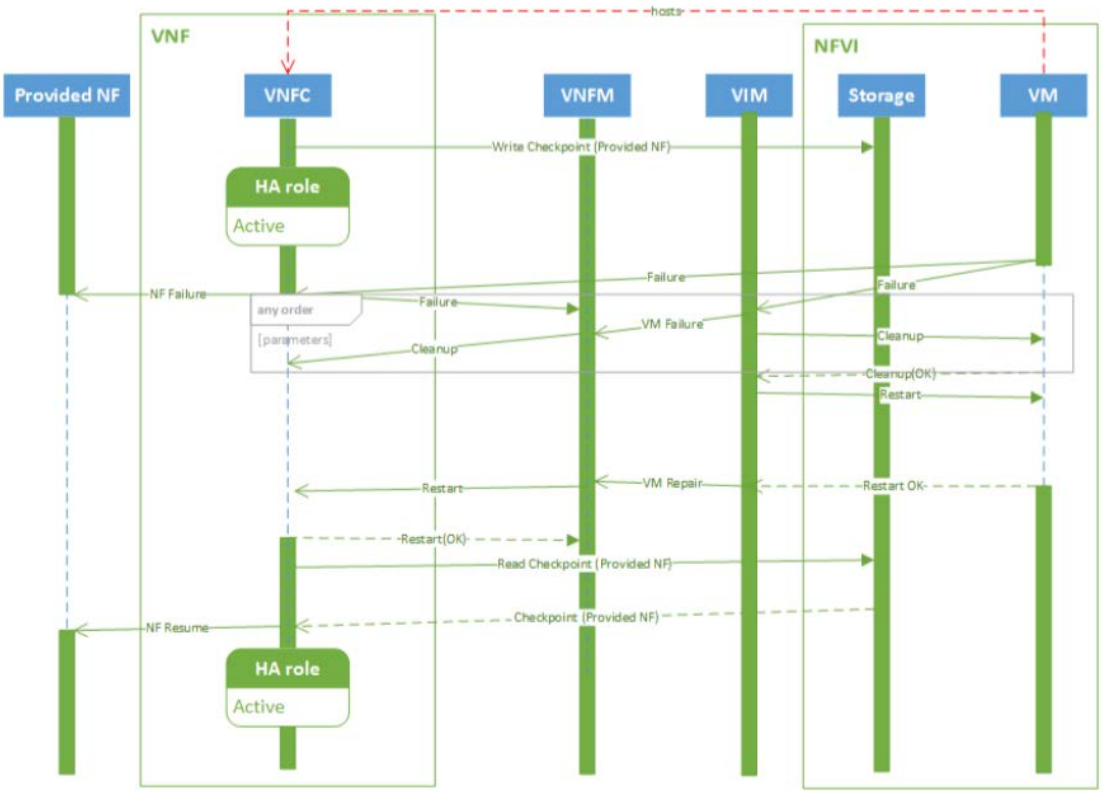


Figure 22 Sequence for use case 4

Use Case 5: VNFC failure in a stateless VNF with redundancy

Use case 5 represents a stateless VNF with redundancy, i.e.it is composed of VNFC1 and VNFC2.They are managed by an HA manager within the VNF. The HA manager assigns the active role to provide the Provided NF to one of the VNFCs while the other remains a spare meaning that it has no state information for the Provided NF therefore it could replace any other VNFC capable of providing the Provided NF service. In this use case the VNFC fails and the failure is detected and handled by the HA manager. Initially VNFC2 provides the Provided NF while VNFC1 is idle or might not even been instantiated yet.

When VNFC2 fails the Provided NF is interrupted. This failure is detected by the HA manager, which as a first reaction cleans up VNFC2 (fault isolation), then it assigns the active role to VNFC1. It may report an error to the VNFM as well.

Since there is no state information to recover, VNFC1 can accept the active role right away and resume providing the Provided NF service. Thus the service outage is over. If the HA manager reported an error to the VNFM it should clear it at this point.

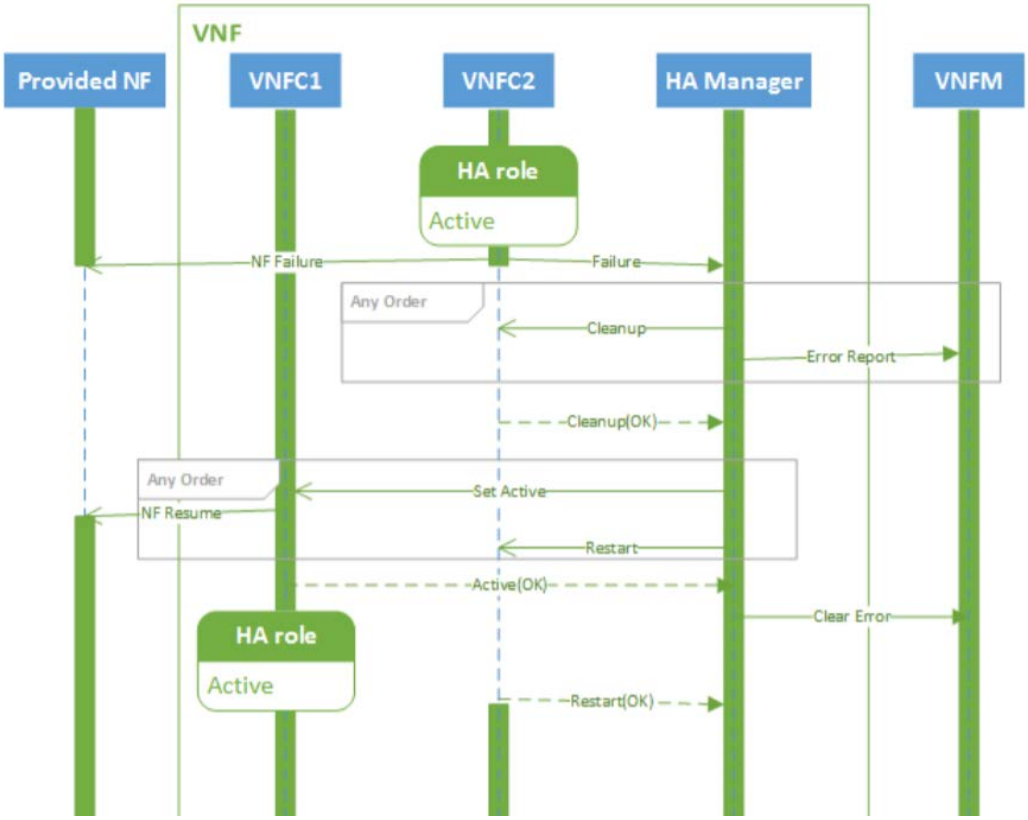


Figure 23 Sequence for use case 5

Use Case 6: VM failure in a stateless VNF with redundancy

Similarly to use case 5, use case 6 represents a stateless VNF composed of VNFC1 and VNFC2,which are managed by an HA manager within the VNF. The HA manager assigns the active role to provide the Provided NF to one of the VNFCs while the other remains a spare meaning that it has no state information for the Provided NF and it could replace any other VNFC capable of providing the Provided NF service.

As opposed to use case 5 in this use case the VM hosting one of the VNFCs fails. This failure is detected and handled by the HA manager as well as the VIM. initially VNFC2 provides the Provided NF while VNFC1 is idle or might not have been instantiated yet as in use case 5. When VM2 fails VNFC2 fails with it and the Provided NF is interrupted. The failure is detected by the HA manager and by the VIM simultaneously and independently.

The HA manager's first reaction is trying to cleanup VNFC2 to isolate the fault. This is considered to be successful as soon as the disappearance of the VM is confirmed. After this the HA manager assigns the active role to VNFC1. It may report the error to the VNFM as well requesting a VM repair. Since there is no state information to recover, VNFC1 can accept the assignment right away and resume the Provided NF service. Thus the service outage is over. If the HA manager reported an error to the VNFM for the service it should clear it at this point. Simultaneously the VM failure is detected in the NFVI and reported to the VIM, which reports it to the VNFM, if the VNFM hasn't requested a repair yet. If the VNFM requested the VM repair or if it acknowledges the repair, the VIM restarts the VM. Once the VM is up the VIM reports it to the VNFM, which in turn may restart the VNFC if needed.

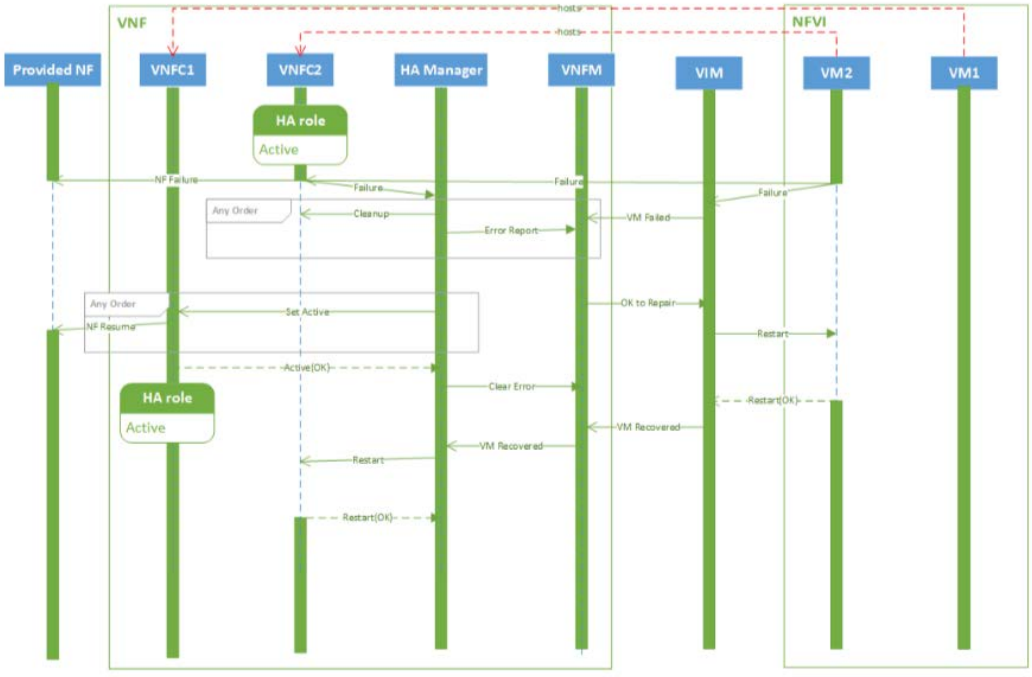


Figure 24 Sequence for use case 6

Use Case 7: VNFC failure in a stateless VNF with no redundancy

Use case 7 represents a stateless VNF composed of a single VNFC, i.e. with no redundancy. The VNF and in particular its VNFC is managed by the VNFM through managing its life-cycle. In this use case the VNFC fails. This failure is detected and handled by the VNFM. This use case requires that the VNFM can detect the failures in the VNF or they are reported to the VNFM.

The failure is only detectable at the VNFM level and it is handled by the VNFM restarting the VNFC. The VNFC is providing the Provided NF when it fails. This failure is detected or reported to the VNFM, which has to clean up the VNFC to isolate the fault. After cleanup success it can proceed with restarting the VNFC, which as soon as it is up it starts to provide the Provided NF as there is no state to recover. Thus the service outage is over, but it has included the entire time needed to restart the VNFC. Considering that the VNF is stateless this may not be significant still.

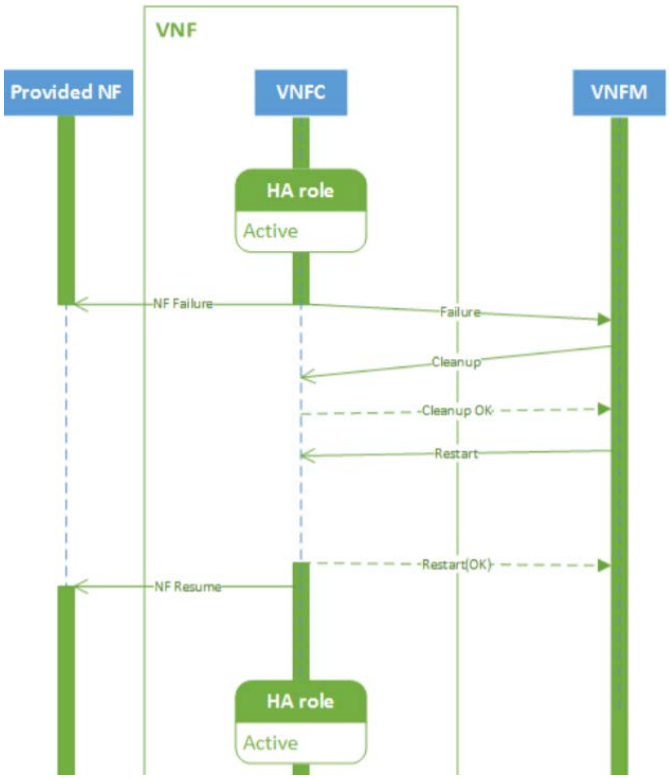


Figure 25 Sequence for use case 7

Use Case 8: VM failure in a stateless VNF with no redundancy

Use case 8 represents the same stateless VNF composed of a single VNFC as use case 7, i.e. with no redundancy. The VNF and in particular its VNFC is managed by the VNFM through managing its life-cycle. In this use case the VM hosting the VNFC fails. This failure is detected and handled by the VNFM as well as by the VIM.

The VNFC is providing the Provided NF when the VM hosting the VNFC fails. (This failure may be detected or reported to the VNFM as a failure of the VNFC. The VNFM may not be aware at this point that it is a VM failure. Accordingly its first reaction as in use case 7is to clean up the VNFC to isolate the fault. Since the VM is gone, this cannot succeed and the VNFM becomes aware of the VM failure through this or it is reported by the VIM. In either case it has to wait with the repair of the VMFC until the VM becomes available again. Meanwhile the VIM also detects the VM failure and reports it to the VNFM unless the VNFM has already requested the VM repair. After the VNFM confirming the VM repair the VIM restarts the VM and reports the successful repair to the VNFM, which in turn can start the VNFC hosted on it.

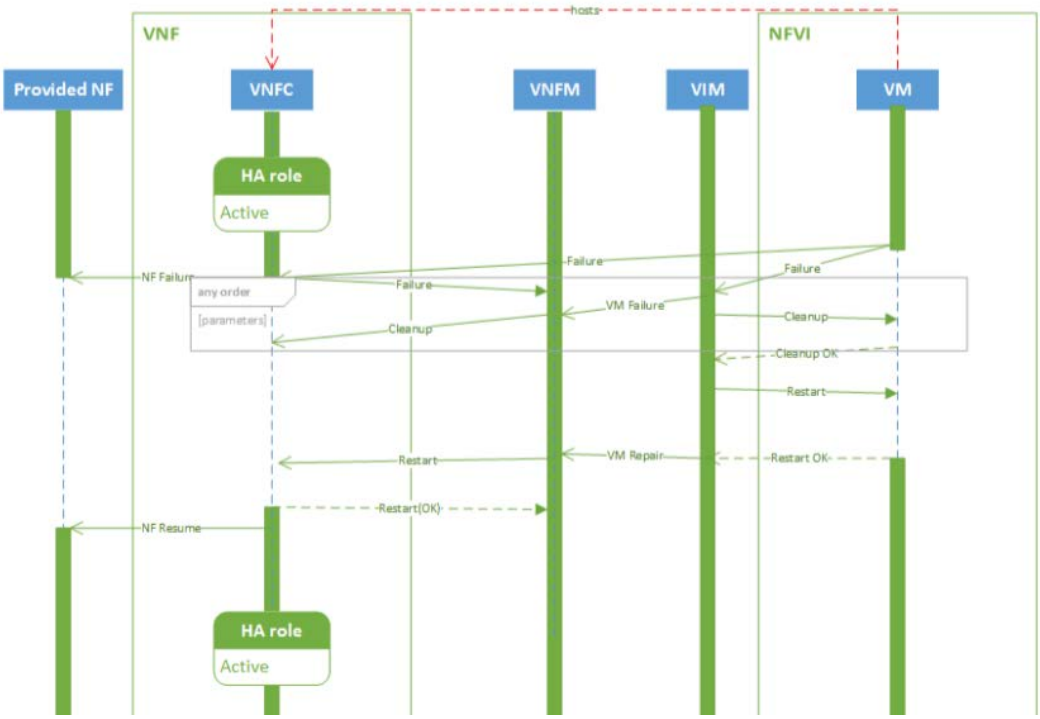


Figure 26 Sequence for use case 8

2.1.2. Openstack Clustering service Senlin

Basic features

Senlin is a clustering service for OpenStack clouds. It creates and operates clusters of homogeneous objects exposed by other OpenStack services. The goal is to make orchestration of collections of similar objects easier. Senlin interacts with other OpenStack services so that clusters of resources exposed by those services can be created and operated. These interactions are mostly done through the via profile plugins. Each profile type implementation enable Senlin to create, update, delete a specific type of resources. A Cluster can be associated with different Policy objects that can be checked/enforced at varying enforcement levels. Through service APIs, a user can dynamically add Node to and remove node from a cluster, attach and detach policies, such as creation policy, deletion policy, load-balancing policy, scaling policy, health policy etc. Through integration with other OpenStack projects, users will be enabled to manage deployments and orchestrations large-scale resource pools much easier. Senlin is designed to be capable of managing different types of objects. An object's lifecycle is managed using Profile Type implementations, which are plugins that can be dynamically loaded by the service engine[8].

The primary features of the Senlin service are listed below:

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 plugin-based event notification that allows for dumping or pumping cluster actions to storage or downstream software.

A asynchronous execution engine for ensuring the state consistency of clusters and nodes.

Architecture

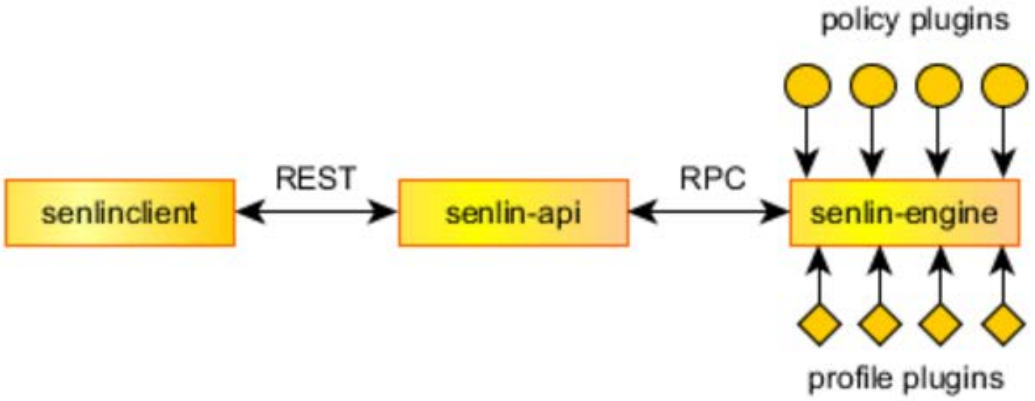


Figure 27 Senlin Components

The main components in the picture are:

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

2.1.3. MANO related Projects

1. OpenStack Tacker

Tacker is an official OpenStack project building a Generic VNF Manager (VNFM) and a NFV Orchestrator (NFVO) to deploy and operate Network Services and Virtual Network Functions (VNFs) on an NFV infrastructure platform like OpenStack. It is based on ETSI MANO Architectural Framework and provides a functional stack to Orchestrate Network Services end-to-end using VNFs.

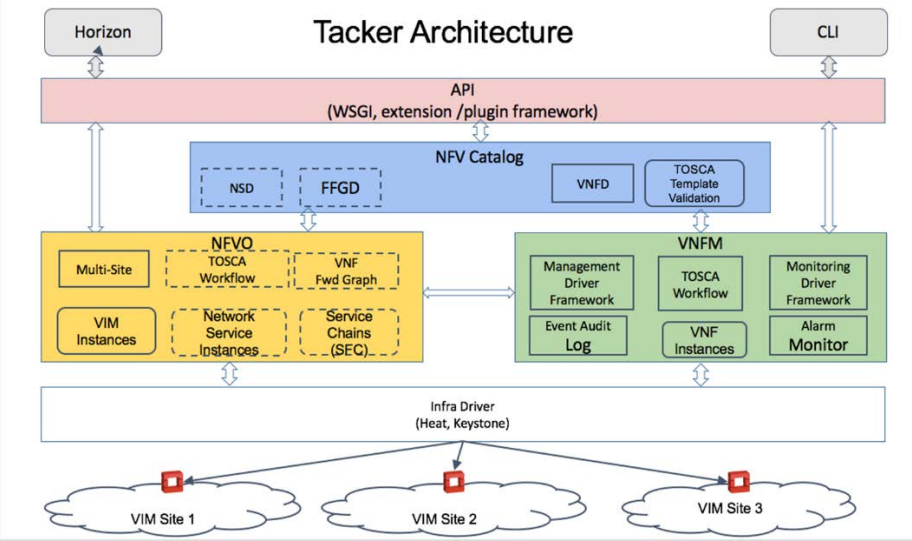


Figure 28 Tacker Architecture

NFV Catalog

- VNF Descriptors
- Network Services Descriptors
- VNF Forwarding Graph Descriptors

VNFM

- Basic life-cycle of VNF (create/update/delete)
- Enhanced platform-aware (EPA) placement of high-performance NFV workloads
- Health monitoring of deployed VNFs
- Auto Healing / Auto Scaling VNFs based on Policy
- Facilitate initial configuration of VNF

NFVO

- Templatized end-to-end Network Service deployment using decomposed VNFs
- VNF placement policy – ensure efficient placement of VNFs
- VNFs connected using an SFC - described in a VNF Forwarding Graph Descriptor
- VIM Resource Checks and Resource Allocation
- Ability to orchestrate VNFs across Multiple VIMs and Multiple Sites (POPs)

o **Highlight features**

Tacker VNF Forwarding Graph:

This spec describes the plan to introduce VNF Forwarding Graph (VNFFG) capability into Tacker. In its current state, Tacker allows for managing VNFs; the purpose of this spec is to also include managing network traffic through paths of ordered VNFs.

Problem description:

There is a large desire from the NFV community to be able to orchestrate and manage traffic through VNFs, also known as Service Function Chaining (SFC)[9]. A user of NFV would not only like to be able to create VNFs, but also define SFCs to direct traffic between VNFs. In the ETSI MANO context[10] a SFC construct is part of a larger graph of VNFs (VNFFG) which defines how VNFs are connected in a graph and the network traffic paths which flow through the graph.

The goal is to be able to define a graph in orchestration via a logical and abstract construct, while being able to render that graph down to the overlay network as SFCs. The next step is to be able to classify tenant traffic that should flow through that SFC. The combination of VNFs, SFC, and the classification of traffic to flow through them is described as the VNF Forwarding Graph (VNFFG).

A VNFFG can be complex with multiple paths through the graph. Today the SFC implementations that Tacker VNFFG will rely on to render a graph are only capable of creating single-path SFCs.

In order to solve this problem, Tacker VNFFG can be made capable to parse a multi-path graph into several single-path SFCs to create the larger VNFFG.

This feature can be thought of as "chain optimization" by gathering info about all of the defined paths through the graph, and breaking up common pieces of path into single chains. This is a key part of VNFFG orchestration, but will be handled as a follow up spec due to the complexity of logic required to orchestrate and manage such a task. This spec addresses the changes to Tacker necessary to orchestrate full paths through a VNFFG using single SFCs.

In addition, a VNF may also be able to re-classify traffic once inside of the graph. For example, a VNF capable of L7 Deep Packet Inspection (DPI) determines from packet payload that the next path through the graph should be modified. To implement such a feature there would need to be some type of coordination between VNFM and a Network Service (NS) extension, and is beyond the scope of this VNFFG specification (but may integrate with such a feature in a future spec).

Proposed changes:

The high-level changes needed to Tacker in order to accommodate this new feature will include changes to Tacker Client, Horizon, and Server. Changes include:

Add a VNFFG tab to tacker-horizon where a user can create a graph from already created VNFs, as well as a Classification sub-tab to declare traffic into the graph. These inputs can be implemented in multiple ways, including (1) a TOSCA VNF Forwarding Graph Descriptor (VNFFGD)[11], as well as (2) a simple drop down menu of chaining VNFs in order and then defining classification schemes for tenants. VNFFG describes network functions and how they are connected via yaml templates as input. This is similar to how VNFDs already work in Tacker VNFM. This spec proposes to implement VNFFG creation via (1) as a first priority.

Tacker Client will also need similar changes to allow passing the CRUD VNFFG calls to Tacker Server.

Tacker Server will need updates to the NFVO extension and plugin in order to integrate VNFFG resources and functionality.

Drivers for 'vnffg' will need to be written. The known drivers to create SFCs are networking-sfc [12](Neutron based SFC) and OpenDaylight SFC[13]. The driver that will be supported for the VNFFG plugin will be the networking-sfc driver[14].

The OpenDaylight functionality and driver will be addressed as a separate spec in the networking-sfc project as a driver to networking-sfc. The networking-sfc driver will handle both: SFC and Classification CRUD operations.

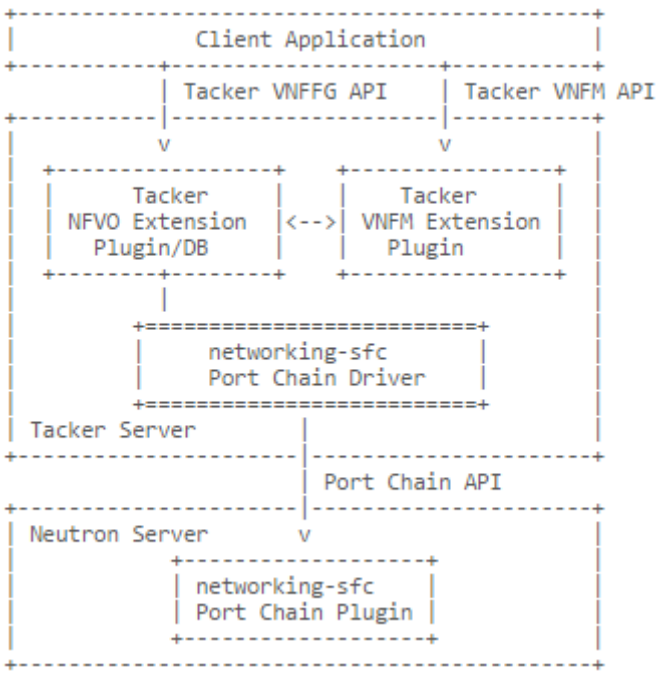


Figure 29 Integration model of tacker

Data model impact:

Data model impact includes the creation of 'vnffgd', 'vnffgd_nfp', 'vnffg', 'vnffg_nfp', 'vnffg_chain' and 'vnffg_classifier' tables. The 'vnffgd' table will hold all of the currently defined VNFFGD templates as defined by TOSCA, while the associated table 'vnffgd_nfp' will hold the Network Forwarding Paths (NFP) associated with that VNFFGD.

The 'vnffg' table will hold relevant VNFFG instance creation attributes, along with associated NFPs held in the 'vnffg_nfp' table and references to the associated SFC and classifier created in the 'vnffg_chain' and 'vnffg_classifier' tables. Another table 'acl_match_criteria' will hold the entries of match criteria mapped to the classifier created in 'vnffg_classifier'.

In VNFM the VNFD template will need to add the following TOSCA properties:

```
tosca.nodes.nfv.VNF:
  properties:
    nsh_aware:
      type: boolean
      required: false
      description: Does this VNF support IETF NSH

tosca.nodes.nfv.CP:
  properties:
    sfc_encapsulation:
      type: string
      required: false
      description: Identifies the method of encapsulation for NSH/SFC
      constraints:
        - [vxlan_gpe, ethernet, mpls]
```

Figure 30 The example of TOSCA

These properties will allow the VNFFG to indicate to the SFC provider if the VNF is Network Service Header (NSH) aware and what encapsulation to use in transporting the packet. NSH is an IETF protocol which passes information about a SFC hop by hop. The NSH header is added to each packet which traverses the chain, and holds properties about that chain so that when a packet arrives at the next VNF in the chain, the VNF is able to determine which chain that packet belongs to, and some idea about how many nodes in the chain the packet has traversed previously.

Restful API impact:

VNFFGD will need to be created in order to instantiate VNFFGs. The method of creating VNFFGD follows the TOSCA template scheme. The format will require one or more VNFFGs defined in Groups along with one or more associated "Forwarding_paths".

VNFs, Connection Points (CPs) and Virtual Links (VLs) are described the VNFD template. Due to this dependency, to validate a VNFFGD the VNFD templates need to be created first. In TOSCA the VL should actually be defined in the Network Service (NS) template, as well as the VNF itself as an abstract construct. In addition, the TOSCA specification defines a "capability" object for each abstract VNF which resolves via substitution mappings to a specific CP (which is exposed as a VNF external type CP). The combination of the capability object and abstract

VNF object make up the "forwarders" in an Network Forwarding Path (NFP). However, the NS is outside of the scope of this spec and will be addressed as a follow up specification. Therefore this spec takes the more simple and direct approach to define a path "forwarder" as a CP associated to a VNFD name. The "forwarder" key listed in the above yaml specifies the VNFD name, while the "capability" key references the external CP for that VNF. Example:

```
Forwarding_path1:
  type: tosca.nodes.nfv.FP
  id: 51
  description: creates path (CP11->CP12->CP32)
  properties:
    policy:
      type: ACL
      criteria:
        - neutron_net_name: tenant1_net
        - dest_port_range: 80-1024
        - ip_proto: tcp
        - ip_dest: 192.168.1.2
  requirements:
    - forwarder: VNF1
      capability: CP11
    - forwarder: VNF1
      capability: CP12
    - forwarder: VNF3
      capability: CP32

groups:
  VNFFG1:
    type: tosca.groups.nfv.VNFFG
    description: HTTP to Corporate Net
    properties:
      vendor: tacker
      version: 1.0
      number_of_endpoints: 5
      dependent_virtual_link: [VL1,VL2,VL3]
      connection_point: [CP11,CP12,CP32]
      constituent_vnfs: [VNF1,VNF3]
    members: [Forwarding_path1]
```

Figure 31 TOSCA example for VNFFG

The basic method of VNFFG creation will be accomplished by instantiating a created VNFFGD. The default behavior of VNFFG creation will rely on selecting abstract VNF types. The VNFFGD contains one or more NFPs, each containing a list of forwarders used in the path. The "forwarder" in requirements references a VNFD name to be used in the path. At VNFFG creation time, the NFVO plugin will query VNFM to find available VNF instances that exist from the corresponding VNFDs for each NFP.

For the first iteration of this spec the selection algorithm to choose which VNF to use if more than one exist for a given VNFD will be random, but may be enhanced in a future spec. VNFs will be allowed to be part of multiple paths, but are not allowed to be part of multiple VNFFGs. The ability to specify the VNF instances (already created via VNFM) to use in the graph can be done by using the '–vnf-mapping' argument. This argument will map <VNFD>:<VNF Instance ID/NAME>. For example, if using the above "Forwarding_path1" yaml input as an example, it contains VNF1 and VNF3 VNFDs. Therefore if there were two instances spawned from those VNFDs, VNF1Test and VNF3Test, the argument would look like '–vnf-mapping VNF1:VNF1Test,VNF3:VNF3Test' in order to indicate to NFVO to specifically use those VNF instances (rather than searching).

The possibility of being able to automatically spawn a non-existent VNF instance of a desired type (that matches an existing VNFD) is outside the scope of this spec, but may be supported later by an additional spec for a NS extension.

The Forwarding Path element (nfv.FP) of the TOSCA input defines a path through the graph. A VNFFGD can contain multiple paths (NFPs) through a VNFFG. Multiple NFPs are associated with a VNFFG by listing it as a target in the VNFFG definition. The initial implementation of this spec will focus on creating a single chain and classifier per path. As previously mentioned this functionality could evolve to optimize common paths through a graph into consolidated chains, but that is outside the scope of this initial spec. The classifier for a path is defined as a policy as shown in the example above, while the chain is listed under requirements. The CPs in the requirements map to a virtual port that must be defined in the VNFD for the specified forwarder. The CP must be defined as having 'forwarding' capability to be part of the chain. The logical CP in a VNFD will map to a Neutron port for the VNF instance. VNFFG will query VNFM to GET the neutron-port ID for a given CP. VNFM will then invoke it's Heat driver to find the information. This will be new behavior and change needed to VNFM. If a single CP is provided per VNF in the Forwarding Path, then it will be considered to be the ingress and egress port for that VNF. If two ordinal CPs are provided per VNF in the Forward Path, then the first will be interpreted to be the ingress port to the VNF, while the second is the egress.

An additional argument, '–symmetrical', will automatically create reverse paths for the paths listed as targets in the VNFFG. The reverse path alternatively may be defined in the VNFFGD, but as a convenience factor –symmetrical may be used instead.

Example CLI calls:

To create VNFFGD:

tacker vnffgd-create --name VNFFG1 --vnffgd-file ./test-vnffgd.yaml

tacker vnffgd-create --name VNFFG1 --vnffgd <raw vnffgd TOSCA>

To create VNFFG (where testVNF1, and testVNF2 are VNF instances):

tacker vnffg-create --name myvnffg --vnfm_mapping VNF1:testVNF2,

VNF2:testVNF1 --symmetrical True --vnffgd-name VNFFG1

tacker vnffg-create --name myvnffg --vnfm_mapping VNF1:testVNF2,

VNF2:testVNF1 --symmetrical True --vnffgd-id

65056908-1946-11e6-b6ba-3e1d05defe78

To list forwarding paths for the vnffg, which will list associated chains and classifiers:

tacker vnffg-show myvnffg

Field	Value
forwarding_paths	Forwarding_path1
id	19233232-d3e2-4c47-a94d-d1b1ab9889e5
name	myvnffg
tenant_id	0b324885958c42ad939e7d636abe2352
vnffgd_id	5279690a-2153-11e6-b67b-9e71128cae77
vnf_mapping	[{VNFD1:testVNF1}, {VNFD2:testVNF2}]
status	ACTIVE

To see the associated chains and classifiers to a specific forwarding path:

tacker vnffg-show myvnffg --nfp Forwarding_path1

Field	Value
chain_id	b8ad61b1-5fac-48ab-9231-dc7d5de6ad4d
classifier_id	0a52a0d9-2a1f-4019-94c3-5401c4af5d36
id	19233232-d3e2-4c47-a94d-d1b1ab9889e5
name	Forwarding-path1
tenant_id	0b324885958c42ad939e7d636abe2352
path_id	200
symmetrical	false
vnffg_id	19233232-d3e2-4c47-a94d-d1b1ab9889e5

To show the chain itself:

tacker vnffg-show --sfc b8ad61b1-5fac-48ab-9231-dc7d5de6ad4d

Field	Value
chain	0a52a0d9-2a1f-4019-94c3-5401c4af5d36
id	b8ad61b1-5fac-48ab-9231-dc7d5de6ad4d
path_id	181
nfp_id	19233232-d3e2-4c47-a94d-d1b1ab9889e5
status	PENDING_CREATE
symmetrical	False
tenant_id	0b324885958c42ad939e7d636abe2352

To show the classifier itself:

tacker vnffg-show --classifier 0a52a0d9-2a1f-4019-94c3-5401c4af5d36

Field	Value
acl_match_criteria	{"source_port": 2005, "protocol": 6, "dest_port": 80}
chain_id	b8ad61b1-5fac-48ab-9231-dc7d5de6ad4d
id	0a52a0d9-2a1f-4019-94c3-5401c4af5d36
nfp_id	19233232-d3e2-4c47-a94d-d1b1ab9889e5
status	PENDING_CREATE
tenant_id	0b324885958c42ad939e7d636abe2352

Attribute Name	Type	Access	Default Value	Validation/ Conversion	Description
id	string (UUID)	RO, All	generated	N/A	identity
name	string	RW, All	None (required)	string	human+readable name
description	string	RW, All	''	string	description of VNFFG
vnffgd_id	string (UUID)	RO, All	None (required)	uuid	VNFFGD to use to create this VNFFG
tenant_id	string	RW, All	None (required)	string	project id to launch VNFFG
status	string	RO, All	generated	string	current state of VNFFG
vnf_mapping	list	RW, All	None	list	Mapping of VNFD name to VNF instances to use in VNFFG
forwarding_ paths	list	RO, All	None	list	List of associated NFPs

REST Calls	Type	Expected Response	Body Data Schema	Description
create_vnffg	post	200 OK	schema 1	Creates VNFFG and triggers underlying chain and classifier creation
update_vnffg	put	200 OK	schema 1	Updates VNFFG by name or ID
delete_vnffg	delete	200 OK	None	Deletes VNFFG by name or ID
show_vnffg	get	200 OK	None	Returns output of specific VNFFG ID, including associated chains and classifiers
list_vnffgs	get	200 OK	None	Returns list of configured VNFFG Names/IDs

Figure 32 VNFFG List

Allow a user to access and show the nfp resource for a vnffg: /vnffg/nfp

Attribute Name	Type	Access	Default Value	Validation/ Conversion	Description
id	string (UUID)	RO, All	generated	N/A	identity
name	string	RO, All	''	string	human+readable name
vnffg_id	string (UUID)	RO, All	generated	uuid	Associated VNFFG ID
tenant_id	string	RO, All	None (required)	string	project id to for this NFP
status	string	RO, All	generated	string	current state of the NFP
classifier_id	string	RO, All	None	string	ID of associated classifier
chain_id	string	RO, All	None	string	ID of associated chain
path_id	integer	RO, All	nfv.FP ID	string	Path ID described in VNFFGD
symmetrical	bool	RO, All	True	bool	Path allows reverse traffic

REST Calls	Type	Expected Response	Body Data Schema	Description
show_nfp	get	200 OK	None	Returns output of specific forwarding_path for a VNFFG
list_nfps	get	200 OK	None	Returns list of configured NFPs fora specific VNFFG

REST Call Failures	Type	Negative Response	Response Message	Scenario
show_nfp	get	404 Not Found	Instance Not Found	No NFP exists with provided Name/ID provided Name/ID

Figure 33 VNFFG NFP

Allow a user to access and show the chain resource as it was rendered: /vnffg/chain

Attribute Name	Type	Access	Default Value	Validation/ Conversion	Description
id	string (UUID)	RO, All	generated	N/A	identity
tenant_id	string	RO, All	None (required)	string	project id to launch SFC
status	string	RO, All	generated	string	current state of SFC
symmetrical	bool	RO, All	True	bool	Chain allows reverse traffic
chain	list	RO, All	None	list	SFC Chain as list of ordered VNF name/IDs
path_id	integer	RO, All	generated	string	NFP/SFC Path ID (e.g. NSH SPI)
nfp_id	string (UUID)	RO, All	None	string	Associated NFP ID

REST Calls	Type	Expected Response	Body Data Schema	Description
show_chain	get	200 OK	None	Returns output of specific chain

REST Call Failures	Type	Negative Response	Response Message	Scenario
show_chain	get	404 Not Found	Instance Not Found	No chain exists with provided Name/ID provided Name/ID

Figure 34 VNFFG Chain

Allow a user access and show the classifier information as it was rendered: /vnffg/classifier

Attribute Name	Type	Access	Default Value	Validation/ Conversion	Description
id	string (UUID)	RO, All	generated	N/A	identity
tenant_id	string	RO, All	None (required)	string	project id to create Classifier
status	string	RO, All	generated	string	current state of Classifier
match	dict	RO, All	True	acl_dict	Match criteria (see supported list)
chain_id	string (UUID)	RO, All	None	string (UUID)	SFC Chain to classify on
nfp_id	string (UUID)	RO, All	None	string	Associated NFP ID

REST Calls	Type	Expected Response	Body Data Schema	Description
show_ classifier	get	200 OK	None	Returns output of specific classifier

REST Call Failures	Type	Negative Response	Response Message	Scenario
show_ classifier	get	404 Not Found	Instance Not Found	No classifier exists with provided Name/ID provided Name/ID

Figure 35 VNFFG Classifier

2.1.3.1. Open Source MANO (OSM)

ETSI OSM is an operator-led ETSI community that is delivering a production-quality open source Management and Orchestration (MANO) stack aligned with ETSI NFV Information Models and that meets the requirements of production NFV networks[15][16].

OSM Release ONE has been engineered, tested and documented to allow for rapid installation in operator labs worldwide that seek to create a scalable and interoperable open source MANO environment. It substantially enhances interoperability with other components (VNFs, VIMs, SDN controllers) and creates a plug-in framework to make platform maintenance and extensions significantly easier to provide and support.

In addition, Release ONE improves administrator and developer experience, both in terms of usability and installation procedure as well as enhances modelling of VNFs and network services. In line with the goals of the OSM open source project, the output of this modelling work will be contributed to ETSI NFV. Release ONE also provides extremely flexible virtualized network functions (VNF) configuration and advanced networking management as well as improved troubleshooting capabilities, with advanced logging.

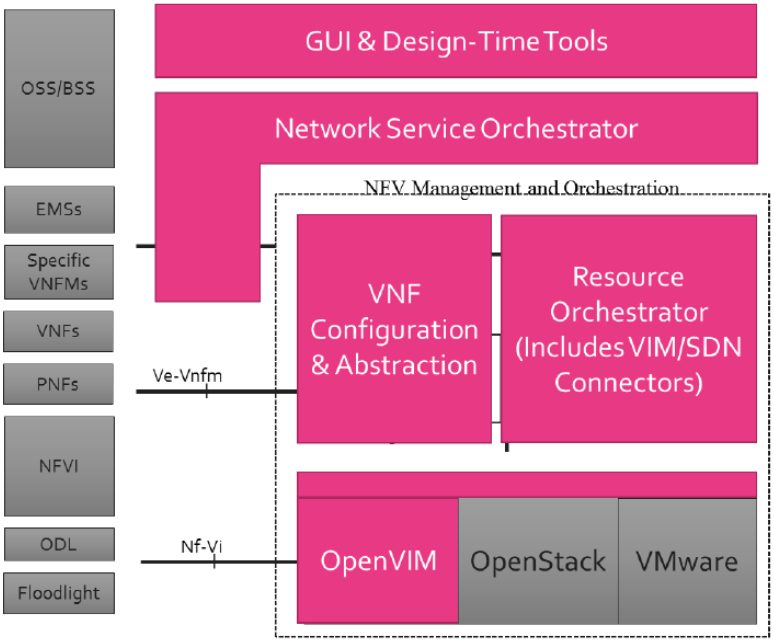


Figure 36 Scope Of OSM

- Automating end-to-end Service Orchestration environment that enables and simplifies the operational considerations of the various lifecycle phases involved in running a complex service based on NFV.
- Supersetting of ETSI NFV MANO where the salient additional area of scope includes Service Orchestration but also explicitly includes provision for SDN control.
- Delivering on a plug-in model for integrating multiple SDN controllers.
- Delivering on a plug-in model for integrating multiple VIMs
- Including one reference VIM that has been optimized for Enhanced Platform Awareness (EPA) to enable high performance VNF deployments.
- Integrating a "Generic" VNFM with support for integrating "Specific" VNFMs.
- Facilitating support for OSM to integrate Physical Network Functions into an automated Network Service deployment.
- Being suitable for both Greenfield and Brownfield deployment scenarios.
- GUI, CLI and REST interfaces to enable access to all features.

The Release ONE development themes are:

- On-boarding experience & VNF Packaging to lower the barrier of entry for VNF vendors.
- Simplified install and upgrade process to accelerate adoption and deployment combined with an improved development environment to facilitate an expansion of the developer community.
- EPA based resource allocation to facilitate high performance VNF deployments with lower Total Cost of Ownership for the operator.
- Service Modelling to simplify, accelerate and standardize the design-time phase.
- Multi-VIM support expanding OSM so that VMware, OpenStack and OpenVIM are enabled.
- Multi-Site support enabling automated service delivery across multiple sites where a site is represented as a grouping of infrastructure managed by a VIM.

2.1.3.2. Open Network Automation Platform (ONAP)

ONAP (Open Network Automation Platform) is an open source software platform that delivers capabilities for the design, creation, orchestration, monitoring, and life cycle management of:

- Low-level Virtual Network Functions (VNFs)
- The carrier-scale Software-Defined Networks (SDNs) that contain them
- Higher-level services that combine the above.

ONAP provides for automatic, policy-driven interaction of functions and services in a dynamic, real-time cloud environment[17][18].

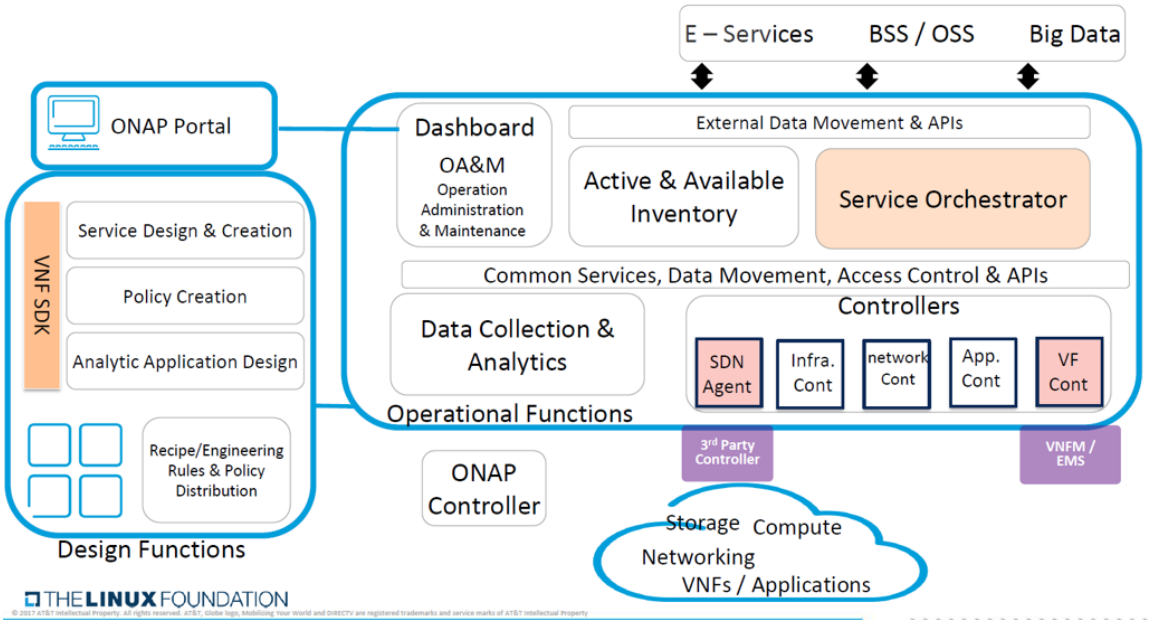


Figure 37 ONAP is a combination of Open-O and ECOMP

Release 1 is scheduled for first quarter, 2017 focusing on:

- Automation of VNF onboarding and other aspects of VNF life cycle.
- Consistent VNF packaging for automated onboarding using ONAP
- Other implementations examples for targeted virtualization environments, such as those located beyond data center at the edge or customer premises.

2.1.3.3. OpenBaton

Open Baton is an open source project providing a comprehensive implementation of the ETSI Management and Orchestration (MANO) specification[19][20].

OpenBaton provides many different features and components:

- A Network Function Virtualisation Orchestrator (NFVO) completely designed and implemented following the ETSI MANO specification.
- A generic Virtual Network Function Manager (VNFM) able to manage the lifecycle of VNFs based on their descriptors.
- A Juju VNFM Adapter in order to deploy Juju Charms or Open Baton VNF Packages using the Juju VNFM.
- A driver mechanism for adding and removing different type of VIMs without having to re-write anything in your orchestration logic.
- A powerful event engine useful based on a pub/sub mechanism for the dispatching of lifecycle events execution.
- An autoscaling engine which can be used for automatic runtime management of the scaling operation operations of your VNFs.
- A fault management system which can be used for automatic runtime management of faults which may occur at any level.
- It integrates with the Zabbix monitoring system.
- A set of libraries (the openbaton-libs) which could be used for building your own VNFM.
- A Marketplace useful for downloading VNFs compatible with the Open Baton NFVO and VNFMs.

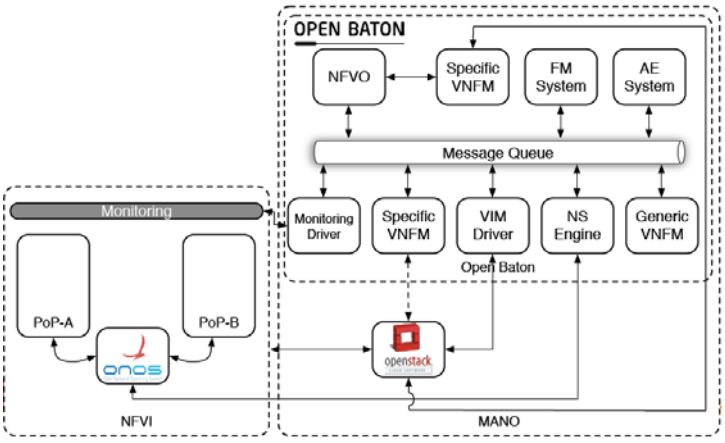


Figure 38 OpenBaton Framework

2.1.4. SFC-related projects

2.1.4.1. OpenStack Networking SFC

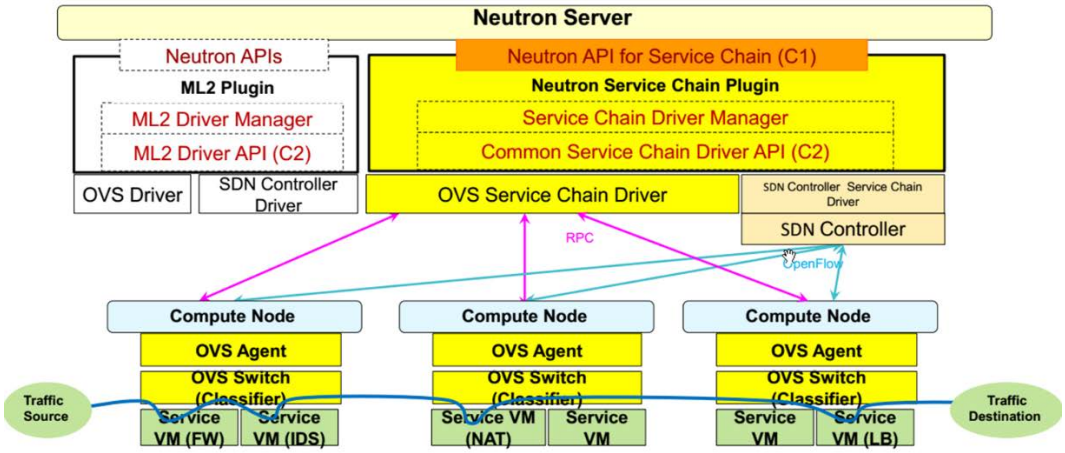


Figure 39 The framework of OpenStack Networking SFC

2.1.4.2. OpenDaylight SFC

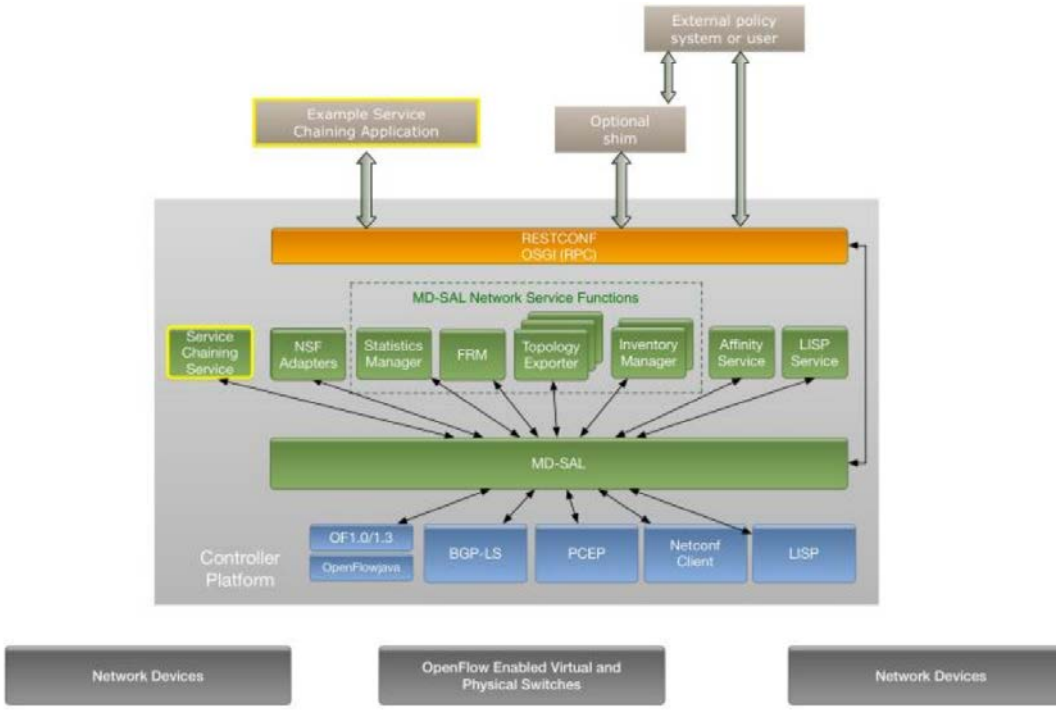


Figure 40 ODL SFC Architecture

OpenDaylight Service Function Chaining (SFC) provides the ability to define an ordered list of a network services (e.g. firewalls, load balancers). These service are then "stitched" together in the network to create a service chain. This project provides the infrastructure (chaining logic, APIs) needed for ODL to provision a service chain in the network and an end-user application for defining such chains[21][22][23][24].

ACE - Access Control Entry

ACL - Access Control List

SCF - Service Classifier Function

SF - Service Function

SFC - Service Function Chain

SFF - Service Function Forwarder

SFG - Service Function Group

SFP - Service Function Path

RSP - Rendered Service Path

NSH - Network Service Header

2.1.4.3. OPNFV SFC

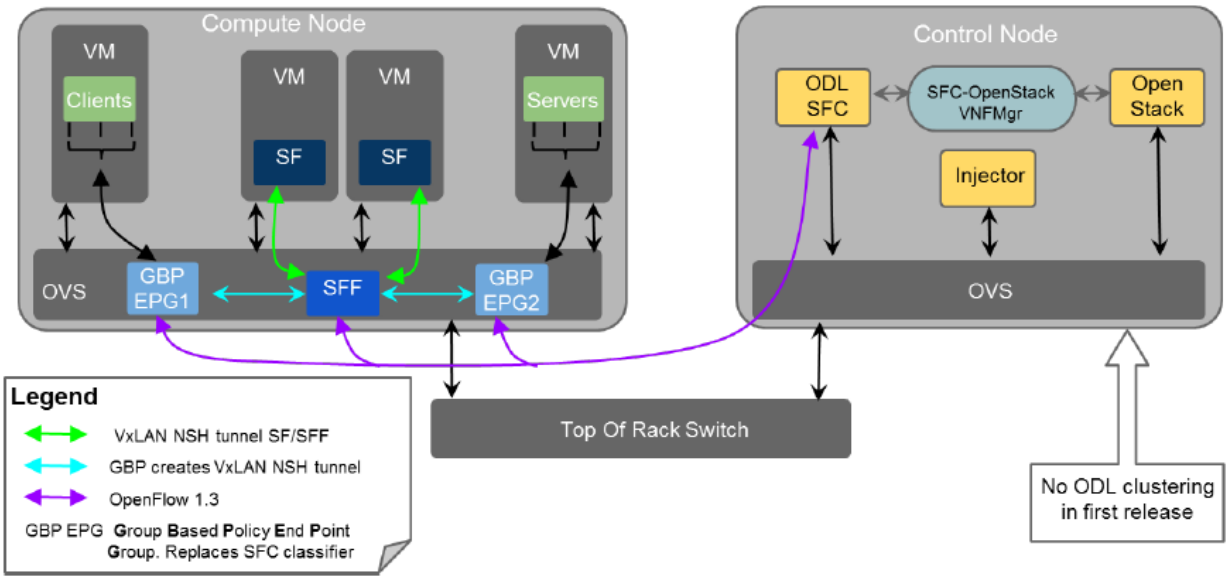


Figure 41 The current topology of OPNFV SFC

This project provides the infrastructure to install the upstream OpenDaylight SFC implementation project in an NFV environment[25][26].

2.1.5. Monitoring projects

OpenStack Telemetry

Project mission: To reliably collect data on the utilization of the physical and virtual resources comprising deployed clouds, persist these data for subsequent retrieval and analysis, and trigger actions when defined criteria are met.

The telemetry requirements of an OpenStack environment are vast and varied, they include use cases such as metering, monitoring, and alarming to name a few. The scope of these uses cases are diverse and beyond the scope of a single project and team. Currently, the telemetry project provides a set of functionality split across multiple projects; each project designed to provide a discrete service in the telemetry space[27].

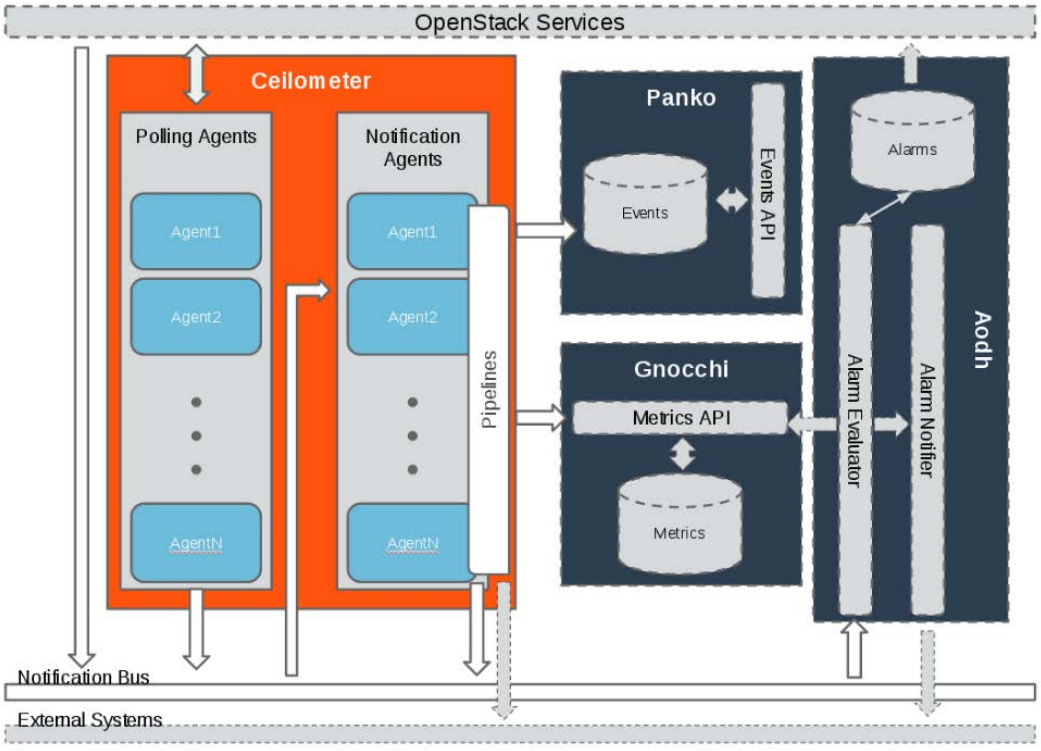


Figure 42 OpenStack Ceilometer Architecture

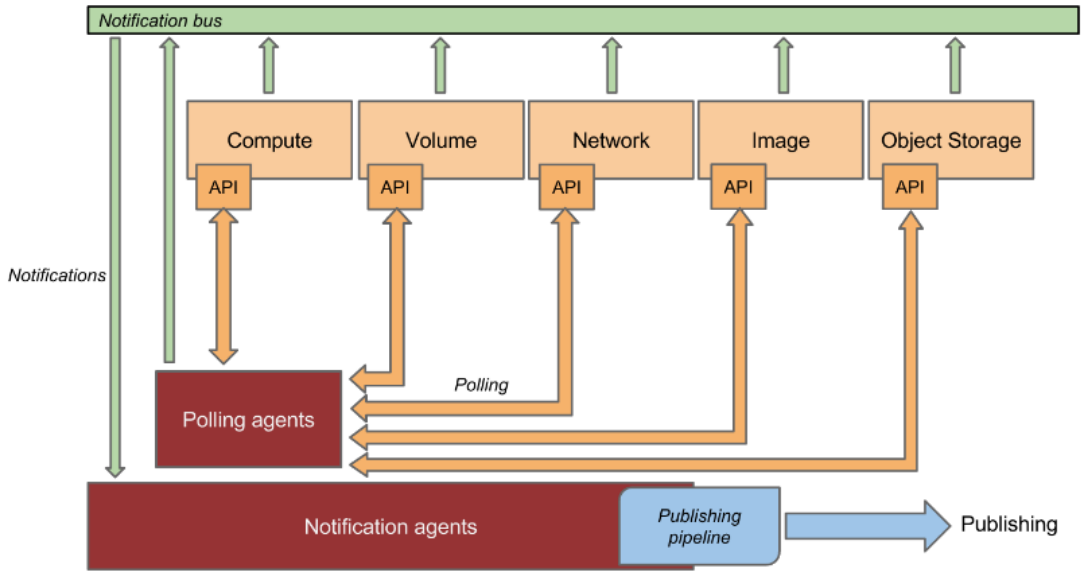
2.1.5.1. OpenStack Ceilometer – A data collection service

Each of Ceilometer's services are designed to scale horizontally[28]. Additional workers and nodes can be added depending on the expected load. Ceilometer offers two core services:

- Polling agent - daemon designed to poll OpenStack services and build Meters.
- Notification agent - daemon designed to listen to notifications on message queue, convert them to Events and Samples, and apply pipeline actions.

Data normalized and collected by Ceilometer can be sent to various targets. Gnocchi was developed to capture measurement data in a time series format to optimize storage and querying. Gnocchi is intended to replace the existing metering database interface. Additionally, Aodh is the alarming service which can send alerts when user defined rules are broken. Lastly, Panko is the event storage project designed to capture document-oriented data such as logs and system event actions.

Gathering the data:



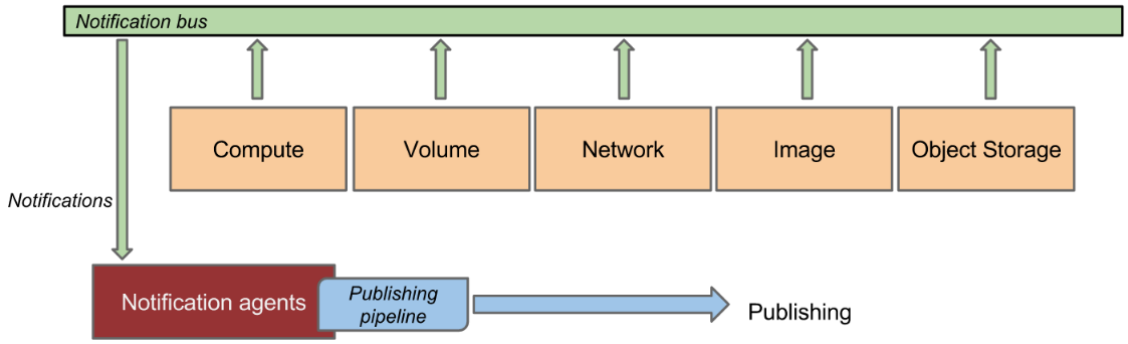
The Ceilometer project created 2 methods to collect data:

Notification agent which takes messages generated on the notification bus and transforms

them into Ceilometer samples or events. This is the preferred method of data collection.

Polling agents, which is the less preferred method, will poll some API or other tool to collect information at a regular interval. The polling approach is less preferred due to the load it can impose on the API services.

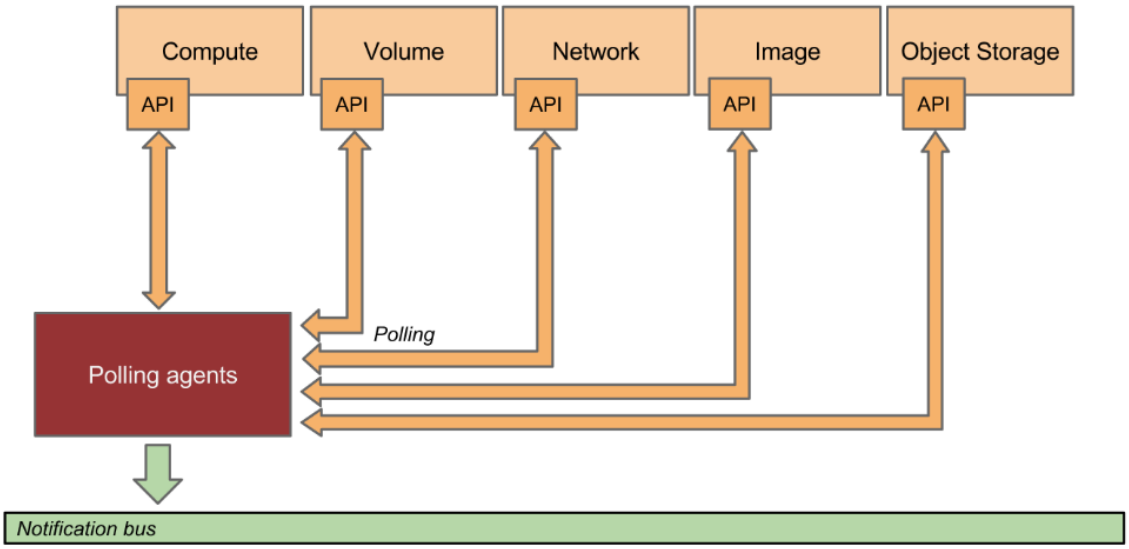
Notification Agents:



Notification agents consuming messages from services.

The heart of the system is the notification daemon (agent-notification) which monitors the message queue for data sent by other OpenStack components such as Nova, Glance, Cinder, Neutron, Swift, Keystone, and Heat, as well as Ceilometer internal communication.

Polling agents:



Polling agents querying services for data. Polling for compute resources is handled by a polling agent running on the compute node (where communication with the hypervisor is more efficient), often referred to as the compute-agent. Polling via service APIs for non-compute

resources is handled by an agent running on a cloud controller node, often referred to the central-agent.

1.1 OpenStack Aodh – An alarming service

The alarming component of Aodh, first delivered in Ceilometer service during Havana development cycle then split out to this independent project in Liberty development cycle, allows you to set alarms based on threshold evaluation for a collection of samples or a dedicate event. An alarm can be set on a single meter, or on a combination. For example, you may want to trigger an alarm when the memory consumption reaches 70% on a given instance if the instance has been up for more than 10 min[29].

2.1.5.2. OpenStack Monasca

OpenStack Monasca features[30]:

- A highly performant, scalable, reliable and fault-tolerant Monitoring as a Service (MONaaS) solution that scales to service provider metrics levels of metrics throughput. Performance, scalability and high-availability have been designed in from the start. Can process 100s of thousands of metrics/sec as well as offer data retention periods of greater than a year with no data loss while still processing interactive queries.
- Rest API for storing and querying metrics and historical information. Most monitoring solution use special transports and protocols, such as CollectD or NSCA (Nagios). In our solution, http is the only protocol used. This simplifies the overall design and also allows for a much richer way of describing the data via dimensions.
- Multi-tenant and authenticated. Metrics are submitted and authenticated using Keystone and stored associated with a tenant ID.
- Metrics defined using a set of (key, value) pairs called dimensions.
- Real-time thresholding and alarming on metrics.
- Compound alarms described using a simple expressive grammar composed of alarm sub-expressions and logical operators.
- Monitoring agent that supports a number of built-in system and service checks and also supports Nagios checks and statsd.

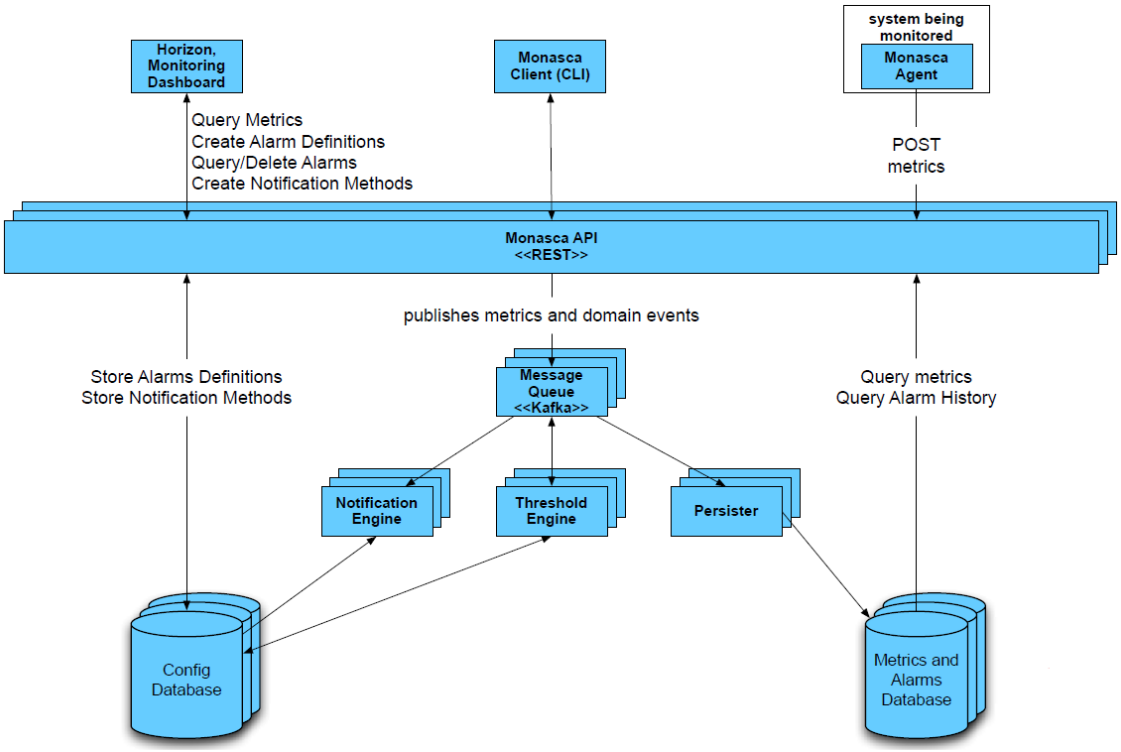


Figure 43 OpenStack Monasca Architecture

Monitoring Agent (monasca-agent): A modern Python based monitoring agent that consists of several sub-components and supports system metrics, such as cpu utilization and available memory, Nagios plugins, statsd and many built-in checks for services such as MySQL, RabbitMQ, and many others.

Monitoring API (monasca-api): A well-defined and documented RESTful API for monitoring that is primarily focused on the following concepts and areas

Persister (monasca-persister): Consumes metrics and alarm state transitions from the MessageQ and stores them in the Metrics and Alarms database.

Transform and Aggregation Engine (monasca-transform): Transform metric names and values, such as delta or time-based derivative calculations, and creates new metrics that are published to the Message Queue. The Transform Engine is not available yet.

Anomaly and Prediction Engine: Evaluates prediction and anomalies and generates predicted metrics as well as anomaly likelihood and anomaly scores. The Anomaly and Prediction Engine is currently in a prototype status.

Threshold Engine (monasca-thresh): Computes thresholds on metrics and publishes alarms to the MessageQ when exceeded. Based on Apache Storm a free and open distributed real-time computation system.

Notification Engine (monasca-notification): Consumes alarm state transition messages from the MessageQ and sends notifications, such as emails for alarms. The Notification Engine is Python based.

Analytics Engine (monasca-analytics): Consumes alarm state transisitions and metrics from the MessageQ and does anomaly detection and alarm clustering/correlation.

Message Queue: A third-party component that primarily receives published metrics from the Monitoring API and alarm state transition messages from the Threshold Engine that are consumed by other components, such as the Persister and Notification Engine. The Message Queue is also used to publish and consume other events in the system. Currently, a Kafka based MessageQ is supported. Kafka is a high performance, distributed, fault-tolerant, and scalable message queue with durability built-in. We will look at other alternatives, such as RabbitMQ and in-fact in our previous implementation RabbitMQ was supported, but due to performance, scale, durability and high-availability limitiations with RabbitMQ we have moved to Kafka.

Metrics and Alarms Database: A third-party component that primarily stores metrics and the alarm state history. Currently, Vertica and InfluxDB are supported. Support for Cassandra is in progress.

Config Database: A third-party component that stores a lot of the configuration and other information in the system. Currently, MySQL is supported. Support for PostgreSQL is in progress.

Monitoring Client (python-monascaclient): A Python command line client and library that communicates and controls the Monitoring API. The Monitoring Client was written using the OpenStack Heat Python client as a framework. The Monitoring Client also has a Python library, "monascaclient" similar to the other OpenStack clients that can be used to quickly build additional capabilities. The Monitoring Client library is used by the Monitoring UI, Ceilometer publisher, and other components.

Monitoring UI: A Horizon dashboard for visualizing the overall health and status of an OpenStack cloud.
Ceilometer publisher: A multi-publisher plugin for Ceilometer, not shown, that converts and publishes samples to the Monitoring API[31].

2.1.5.3. Custom monitoring tools

Zabix

ZABBIX was created by Alexei Vladishev, and currently is actively developed and supported by ZABBIX SIA. ZABBIX is an enterprise-class open source distributed monitoring solution[32]. ZABBIX is software that monitors numerous parameters of a network and the health and integrity of servers. ZABBIX uses a flexible notification mechanism that allows users to configure e-mail based alerts for virtually any event. This allows a fast reaction to server problems. ZABBIX offers excellent reporting and data visualisation features based on the stored data. This makes ZABBIX ideal for capacity planning.
ZABBIX supports both polling and trapping. All ZABBIX reports and statistics, as well as configuration parameters, are accessed through a web-based front end. A web-based front end ensures that the status of your network and the health of your servers can be assessed from any location. Properly configured, ZABBIX can play an important role in monitoring IT infrastructure. This is equally true for small organisations with a few servers and for large companies with a multitude of servers.
ZABBIX is free of cost. ZABBIX is written and distributed under the GPL General Public License version 2. It means that its source code is freely distributed and available for the general public. Both free and commercial support is available and provided by ZABBIX Company.

ZABBIX offers:

- auto-discovery of servers and network devices
- distributed monitoring with centralised WEB administration
- support for both polling and trapping mechanisms
- server software for Linux, Solaris, HP-UX, AIX, Free BSD, Open BSD, OS X

- native high performance agents (client software for Linux ,Solaris, HP-UX, AIX, Free BSD, Open BSD, OS X, Tru64/OSF1, Windows NT4.0, Windows 2000, Windows 2003, Windows XP, Windows Vista)
- agent-less monitoring
- secure user authentication
- flexible user permissions
- web-based interface
- flexible e-mail notification of predefined events
- high-level (business) view of monitored resources

Navigos

Nagios® Core™ is an Open Source system and network monitoring application[33]. It watches hosts and services that you specify, alerting you when things go bad and when they get better. Nagios Core was originally designed to run under Linux, although it should work under most other unices as well.

Some of the many features of Nagios Core include:

- Monitoring of network services (SMTP, POP3, HTTP, NNTP, PING, etc.)
- Monitoring of host resources (processor load, disk usage, etc.)
- Simple plugin design that allows users to easily develop their own service checks
- Parallelized service checks
- Ability to define network host hierarchy using "parent" hosts, allowing detection of and distinction between hosts that are down and those that are unreachable
- Contact notifications when service or host problems occur and get resolved (via email, pager, or user-defined method)
- Ability to define event handlers to be run during service or host events for proactive problem resolution
- Automatic log file rotation
- Support for implementing redundant monitoring hosts
- Optional web interface for viewing current network status, notification and problem history, log file, etc.

3. Overall architecture for HA

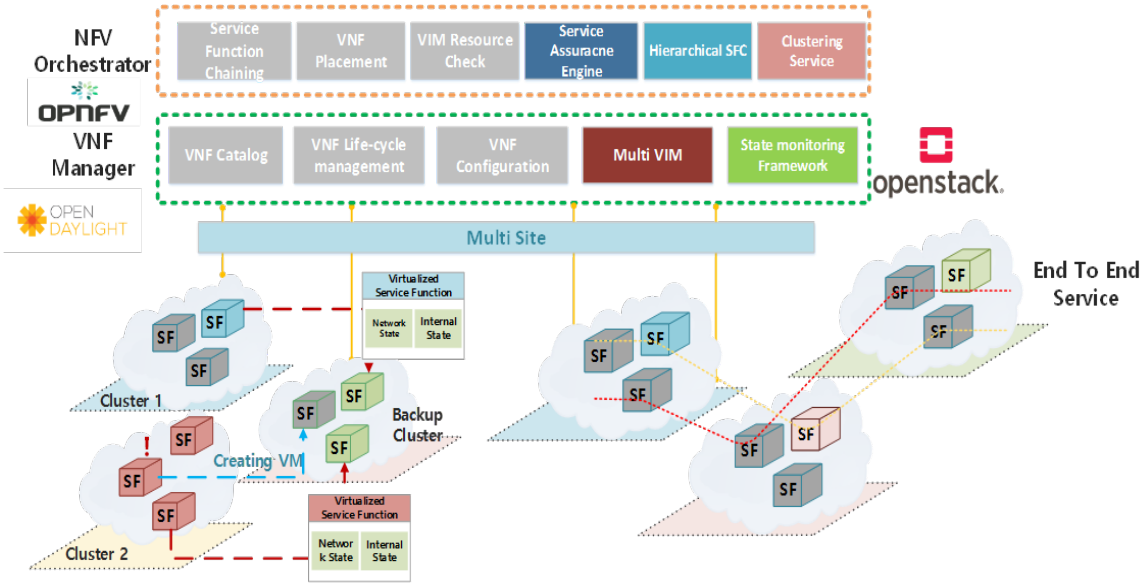


Figure 44 Overall Architecture for HA in OpenStack Environment

To support HA, we should consider three part. At first, we should think about a Monitoring framework. Monitoring framework is the one of the basic requirement for HA since it can detect failure quickly and it triggers Recovery service. Second, we should consider clustering service. To support HA, we should deploy a new machine to consist previous services. However, delay is the one of the issue, if we make a replacement machine after failure. The third, we should consider End to End service and multisite. High availability should be considered the perspective of services which made by several machine rather than one single machine. Therefore, element technology also support End to End service. In OpenStack environment, we modify the tacker(VNFM) to develop element technology for high availability. Next chapter, we introduce our implementation.

4. Implementation

4.1. Proposal: Alarm Monitoring Framework in OpenStack Tacker

Achievements

- Alarm monitoring framework is a significant feature with the highest priority for OpenStack Newton release in Tacker.
- Alarm monitoring framework is presented in OpenStack Barcelona summit 2016, OpenStack Mitaka summit 2016, and OPNFV Berlin summit 2016.
- The alarm monitoring framework is quite strong when comparing with other monitoring modules in other MANO projects like OpenBaton, Open source MANO (OSM). The alarm monitoring enable to flexibly integrate with internal monitoring drivers (OpenStack Ceilometer & Monasca) as well as external monitoring tools like Zabix.

Proposed Architecture and Features

Problem description

ETSI MANO architecture describes to monitor the VNF to take appropriate action such as fault management, performance management. Monitoring became an important aspect in MANO architecture. Currently, Tacker provides a very minimal support for checking the liveliness of VNF elements by means of ping or curl which helps to recover the element in case it is unreachable. But Tacker does not support monitoring of the CPU/memory usage of VNF elements. Further, it is necessary for Tacker to monitor all VNF resources as well. The reason is that the failure of VNFs happen too diversely.

Proposed framework

The scope of this spec focused on:

Designing a generic monitoring framework. Whereby, an alarm-based monitoring driver in Tacker is designed to collect alarms/events triggered by the low-level designs (Ceilometer, Monasca, custom driver). In this spec, the alarm-based monitoring driver can completely monitor any resources in OpenStack that Ceilometer can support. In real implementation, this spec aims to leverage Ceilometer to monitor CPU/memory usage inside VNF.

Defining Monitoring Policy using the TOSCA Policy format. The monitoring policy can apply to a single VDU or multiple VDUs.

Adding support for inserting Ceilometer Alarms into the HOT template to allow Ceilometer to trigger scaling in Heat resource groups.

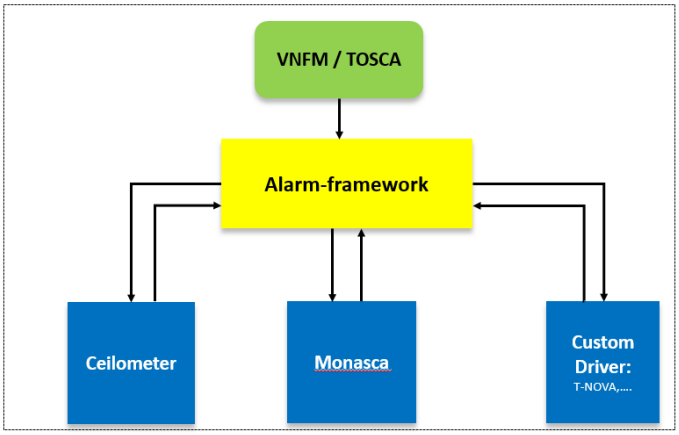


Figure 45 Alarm Monitoring Framework

alarm_url will be created by webhook in Tacker as the following:

```
v1.0/vnfs/<vnf-uuid>/<monitoring-policy-name>/<action-name>/2w3r40-34c2d2
```

Where: monitoring-policy is the name of monitoring policy which is described in VNFD.

action-name is the name of action which is described in VNFD as well. Multiple actions could be supported in monitoring policy. By changing action-name, the appropriate action will be invoked and then the alarm-based monitoring driver will process this action. In above example, action-name is *'vdu1_scaling_policy'*. Whereby, when the monitoring driver receives triggers from Ceilometer, it will invoke scaling action and trigger scaling automatically. The detailed scaling mechanism using the monitoring driver is defined by the scaling spec.

params contains the information related to alarm-actions. For example, it can be used for user authentication. Whereby, Webhook handler will generate randomly a key. This helps to make sure that we have a unique url for each alarm. Alarm url will be stored in Tacker db and only these unique callbacks will be used.

The expression showm below is an example of alarm url which contains user authentication

```
v1.0/vnfs/<vnf-uuid>/<monitoring-policy-name>/<action-name>/2w3r40-34c2d2
```

Here, monitoring-policy-name is the name of monitoring policy and threshold is a value which user wants to update.

Installation

Sample TOSCA with monitoring policy

The following example shows monitoring policy using TOSCA template. The target (VDU1) of the monitoring policy in this example need to be described firstly like other TOSCA templates in Tacker[34].

```
policies:
 - vdu1_cpu_usage_monitoring_policy:
     type: tosca.policies.tacker.Alarming
     triggers:
       resize_compute:
           event_type:
               type: tosca.events.resource.utilization
               implementation: ceilometer
           metrics: cpu_util
           condition:
               threshold: 50
               constraint: utilization greater_than 50%
               period: 65
               evaluations: 1
               method: avg
               comparison_operator: gt
           actions: [respawn]
```

Figure 46 other TOSCA templates in Tacker

Alarm framework already supported the some default backend actions like scaling, respawn, log, and log_and_kill.

Tacker users could change the desired action as described in the above example. Until now, the backend actions could be pointed to the specific policy which is also described in TOSCA template like scaling policy. The integration between alarming monitoring and scaling was also supported by Alarm monitor in Tacker:

Setup environment

If OpenStack Devstack is used to test alarm monitoring in Tacker, OpenStack Ceilometer and Aodh plugins will need to be enabled in local.conf:

enable_plugin ceilometer https://git.openstack.org/openstack/ceilometer master

enable_plugin aodh https://git.openstack.org/openstack/aodh master

How to monitor VNFs

How to setup alarm configuration

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ aodh alarm list
+--------------------------------------+-----------+------------------------------------------------------------------+-------------------+---------
| alarm_id                             | type      | name                                                             | state             | severity
+--------------------------------------+-----------+------------------------------------------------------------------+-------------------+---------
| dd0484e4-948d-4f81-a962-7909c401a2d6 | threshold | tacker.vnfm.infra_drivers.openstack.openstack_OpenStack-ad1e228e- | insufficient data | low
|                                      |           | df91-43b4-ae7c-e69472609e21-vdu_hcpu_usage_scaling_out-lit7ep6yflby |                   |
| 1b2af378-9705-4f1d-b6c8-9c6787e5b685 | threshold | tacker.vnfm.infra_drivers.openstack.openstack_OpenStack-ad1e228e- | insufficient data | low
|                                      |           | df91-43b4-ae7c-e69472609e21-vdu_lcpu_usage_scaling_in-valvn5lrpgte |                   |
+--------------------------------------+-----------+------------------------------------------------------------------+-------------------+---------
```

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ aodh alarm show dd0484e4-948d-4f81-a962-7909c401a2d6
+---------------------------+------------------------------------------------------------------------------------------------------------------
| Field                     | Value
+---------------------------+------------------------------------------------------------------------------------------------------------------
| alarm_actions             | [u'http://pinedcn:9890/v1.0/vnfs/ad1e228e-df91-43b4-ae7c-e69472609e21/vdu_hcpu_usage_scaling_out/SP1-out/tfs4t2fy']
| alarm_id                  | dd0484e4-948d-4f81-a962-7909c401a2d6
| comparison_operator       | gt
| description               | utilization greater_than 50%
| enabled                   | True
| evaluation_periods        | 1
| exclude_outliers          | False
| insufficient_data_actions | None
| meter_name                | cpu_util
| name                      | tacker.vnfm.infra_drivers.openstack.openstack_OpenStack-ad1e228e-df91-43b4-ae7c-e69472609e21-vdu_hcpu_usage_scaling_out-lit7ep6
| ok_actions                | None
| period                    | 600
| project_id                | 55a8f56767f541c78a1d271e7b0ca3a7
| query                     | metadata.user_metadata.vnf_id = SG1
| repeat_actions            | True
| severity                  | low
| state                     | insufficient data
| state_timestamp           | 2016-12-13T18:20:05.111069
| statistic                 | avg
| threshold                 | 50.0
| time_constraints          | []
| timestamp                 | 2016-12-13T18:20:05.111069
| type                      | threshold
| user_id                   | 63db82cc378c412fa60803ec72e33ffe
+---------------------------+------------------------------------------------------------------------------------------------------------------
```

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ aodh alarm show 1b2af378-9705-4f1d-b6c8-9c6787e5b685
+---------------------------+------------------------------------------------------------------------------------------------------------------
| Field                     | Value
+---------------------------+------------------------------------------------------------------------------------------------------------------
| alarm_actions             | [u'http://pinedcn:9890/v1.0/vnfs/ad1e228e-df91-43b4-ae7c-e69472609e21/vdu_lcpu_usage_scaling_in/SP1-in/nw39j0ap']
| alarm_id                  | 1b2af378-9705-4f1d-b6c8-9c6787e5b685
| comparison_operator       | lt
| description               | utilization less_than 10%
| enabled                   | True
| evaluation_periods        | 1
| exclude_outliers          | False
| insufficient_data_actions | None
| meter_name                | cpu_util
| name                      | tacker.vnfm.infra_drivers.openstack.openstack_OpenStack-ad1e228e-df91-43b4-ae7c-e69472609e21-vdu_lcpu_usage_scaling_in-valvn5l
| ok_actions                | None
| period                    | 600
| project_id                | 55a8f56767f541c78a1d271e7b0ca3a7
| query                     | metadata.user_metadata.vnf_id = SG1
| repeat_actions            | True
| severity                  | low
| state                     | insufficient data
| state_timestamp           | 2016-12-13T18:20:05.105414
| statistic                 | avg
| threshold                 | 10.0
| time_constraints          | []
| timestamp                 | 2016-12-13T18:20:05.105414
| type                      | threshold
| user_id                   | 63db82cc378c412fa60803ec72e33ffe
+---------------------------+------------------------------------------------------------------------------------------------------------------
```

How to trigger alarms

As shown in the above Ceilometer command, alarm state is shown as "insufficient data". Alarm is triggered by Ceilometer once alarm state changes to "alarm". To make VNF instance reach to the pre-defined threshold, some simple scripts could be used.

Note: Because Ceilometer pipeline set the default interval to 600s (10 mins), in order to reduce this interval, users could edit "interval" value in /etc/ceilometer/pipeline.yaml file and then restart Ceilometer service.

Another way could be used to check if backend action is handled well in Tacker:

curl -H "Content-Type: application/json" -X POST -d '{"alarm_id": "35a80852-e24f-46ed-bd34-e2f831d00172", "current": "alarm"}' http://pinedcn:9890/v1.0/vnfs/a0f60b00-ad3d-4769-92ef-e8d9518da2c8/vdu_lcpu_scaling_in/SP1-in/yl7kh5qd

Then, users can check Horizon to know if vnf is respawned. Please note that the url used in the above command could be captured from "ceilometer alarm-show command as shown before. "key" attribute in body request need to be captured from the url. The reason is that key will be authenticated so that the url is requested only one time[35].

Result

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ tacker vnf-list
+--------------------------------------+-------------+-------------+--------------------------------------+-----------------+
| id                                   | name        | description | mgmt_url                             | status          |
+--------------------------------------+-------------+-------------+--------------------------------------+-----------------+
| 6d7ef686-4302-47e2-bf6a-             | vnf-respawn | Demo example |                                     | PENDING_CREATE  |
| 9f2c1e212dbe                         |             |             |                                      |                 |
| ad1e228e-df91-43b4-ae7c-             | vnf-scale   | Demo example | {"VDU1": ["192.168.120.9",           | PENDING_SCALE_IN |
| e69472609e21                         |             |             | "192.168.120.13"], "VDU2":           |                 |
|                                      |             |             | ["192.168.120.3", "192.168.120.7"]} |                 |
+--------------------------------------+-------------+-------------+--------------------------------------+-----------------+
```

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ tacker vnf-list
+--------------------------------------+-------------+-------------+--------------------------------------+---------------+
| id                                   | name        | description | mgmt_url                             | status        |
+--------------------------------------+-------------+-------------+--------------------------------------+---------------+
| 6d7ef686-4302-47e2-bf6a-9f2c1e212dbe | vnf-respawn | Demo example |                                     | PENDING_CREATE |
| ad1e228e-df91-43b4-ae7c-e69472609e21 | vnf-scale   | Demo example | {"VDU1": ["192.168.120.13"], "VDU2": | ACTIVE        |
|                                      |             |             | ["192.168.120.7"]}                   |               |
+--------------------------------------+-------------+-------------+--------------------------------------+---------------+
```

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ tacker vnf-list
+--------------------------------------+------------+-------------+--------------------------------------+-------------------+
| id                                   | name       | description | mgmt_url                             | status            |
+--------------------------------------+------------+-------------+--------------------------------------+-------------------+
| 6d7ef686-4302-47e2-bf6a-             | vnf-respawn | Demo example |                                     | PENDING_CREATE    |
| 9f2c1e212dbe                         |            |             |                                      |                   |
| ad1e228e-df91-43b4-ae7c-             | vnf-scale  | Demo example | {"VDU1": ["192.168.120.13"],         | PENDING_SCALE_OUT |
| e69472609e21                         |            |             | "VDU2": ["192.168.120.7"]}           |                   |
+--------------------------------------+------------+-------------+--------------------------------------+-------------------+
```

```
(venv) pine@pinedcn:/opt/stack/tacker/samples/tosca-templates/vnfd$ tacker vnf-list
+--------------------------------------+------------+-------------+--------------------------------------+---------------+
| id                                   | name       | description | mgmt_url                             | status        |
+--------------------------------------+------------+-------------+--------------------------------------+---------------+
| 6d7ef686-4302-47e2-bf6a-9f2c1e212dbe | vnf-respawn | Demo example |                                     | PENDING_CREATE |
| ad1e228e-df91-43b4-ae7c-e69472609e21 | vnf-scale  | Demo example | {"VDU1": ["192.168.120.13",          | ACTIVE        |
|                                      |            |             | "192.168.120.3"], "VDU2":            |               |
|                                      |            |             | ["192.168.120.7", "192.168.120.10"]} |               |
+--------------------------------------+------------+-------------+--------------------------------------+---------------+
```

4.2. Support Senlin auto-scaling policy in heat-translator

This is implementation of translation of Senlin scaling policy from TOSCA template to HOT template in Openstack/Heat-Translator project. Below table shows the TOSCA template and corresponding HOT template[35][36].

TOSCA	**HOT**
tosca_definitions_version: tosca_simple_yaml_1_0 description: > Template for deploying servers based on policies. imports: - custom_types/senlin_cluster_policies.yaml topology_template: node_templates: my_server_1: type: tosca.nodes.Compute capabilities: host: properties: num_cpus: 2 disk_size: 10 GB mem_size: 512 MB os: properties: # host Operating System image properties architecture: x86_64	heat_template_version: 2016-04-08 description: > Template for deploying servers based on policies. parameters: {} resources: my_server_1: type: OS::Senlin::Profile properties: type: os.nova.server-1.0 properties: flavor: m1.medium image: rhel-6.5-test-image networks: - network: net0 cluster_scaling_scale_out: type: OS::Senlin::Policy properties: bindings: - cluster: get_resource: my_server_1_cluster type: senlin.policy.scaling-1.0 properties:

```
            type: Linux
            distribution: RHEL
            version: 6.5
    my_port_1:
      type: tosca.nodes.network.Port
      requirements:
        - link:
            node: my_network_1
        - binding:
            node: my_server_1
    my_network_1:
      type: tosca.nodes.network.Network
      properties:
        network_name: net0
  policies:
    - cluster_scaling:
        type: tosca.policies.Scaling.Cluster
        description: Cluster node autoscaling
        targets: [my_server_1]
        triggers:
          scale_out:
            description: trigger
            event_type:
              type: tosca.events.resource.cpu.utilization
              metrics: cpu_util
              implementation: Ceilometer
            condition:
              constraint: utilization greater_than 50%
              adjustment:
                type: CHANGE_IN_CAPACITY
                number: 1
              event: CLUSTER_SCALE_OUT
  my_server_1_cluster:
    type: OS::Senlin::Cluster
    properties:
      profile:
        get_resource: my_server_1
      min_size: 2
      max_size: 10
      desired_capacity: 3
  my_server_1_scale_out_receiver:
    type: OS::Senlin::Receiver
    properties:
      action: CLUSTER_SCALE_OUT
      cluster:
        get_resource: my_server_1_cluster
      type: webhook
  scale_out_alarm:
    type: OS::Aodh::Alarm
    properties:
      meter_name: cpu_util
      alarm_actions:
      - get_attr:
        - my_server_1_scale_out_receiver
        - channel
        - alarm_url
      description: Cluster node
```

period: 60 evaluations: 1 method: average action: scale_out: type: SCALE_OUT implementation: Senlin.webhook properties: min_instances: 2 max_instances: 10 default_instances: 3 increment: 1	autoscaling evaluation_periods: 1 repeat_actions: True period: 60 statistic: avg threshold: 50 comparison_operator: gt outputs: {}

Integrate Senlin resource to Tacker for VDU management

This spec invents a method to manage VDU scaling for Openstack/Tacker project using Senlin resources. Tacker provides feasible functions of managing VNF which include auto-scaling. Currently auto-scaling function is provided by Heat AutoScalingGroup. But Heat AutoScalingGroup is not support a feasible enough feature for VDU auto-scaling. That is because Heat can only make the instruction of auto-scaling to VDU clusters but is lack of ability of managing the VDU cluster. For example, when VDU auto-scaling failed or VDU went to ERROR status, the only way to recover it is using Heat stack-update, but stack-update usually has a low possibility to recover the stack. Senlin [37] is an OpenStack project which provides clustering service. It defines the concepts of Profile, Cluster, Node, Policy, Receiver, etc. which is fit for the use case of VDU cluster management in Tacker. For example when auto-scaling failed user can use Senlin commands to delete the failed nodes directly. Senlin provides powerful policies (placement policy, deletion policy, etc.) which can be used to make VDU auto-scaling much more intelligent than Heat autoscaling group. If user wants to scale in a cluster, senlin deletion policy (which can added to tacker later) can decide which nodes should be deleted firstly (the elder ones or the young ones) and the nodes which are in ERROR status will always be deleted at first.

For alarming management, Senlin does not only support Ceilometer, but also message service (Zaqar), and even some other monitoring tools defined by user himself, as long as those tools support webhook. Besides scaling function, Senlin also provides HA policy (will be introduced to Tacker in future) which can be used to support HA function for VDUs. By using senlin to manage the VDU cluster, the VDU cluster's health status are always checked by senlin, if the cluster is not healthy, for example, some nodes go to error status, senlin will send an event to notify user. After all it is reasonable to integrate Senlin into Tacker to manage VDU cluster.

To use Senlin, necessary Senlin entities like Profile, Cluster, Policies, etc. should be created by Heat first. All the Senlin resources have been defined in HOT template already. By integrating Senlin with tacker, after all the Tacker resources translated into a HOT template file, the HOT file will be stored in Senlin profile, then Senlin will pass this template to Heat to create VDU, CP VL, etc. The whole workflow is like:

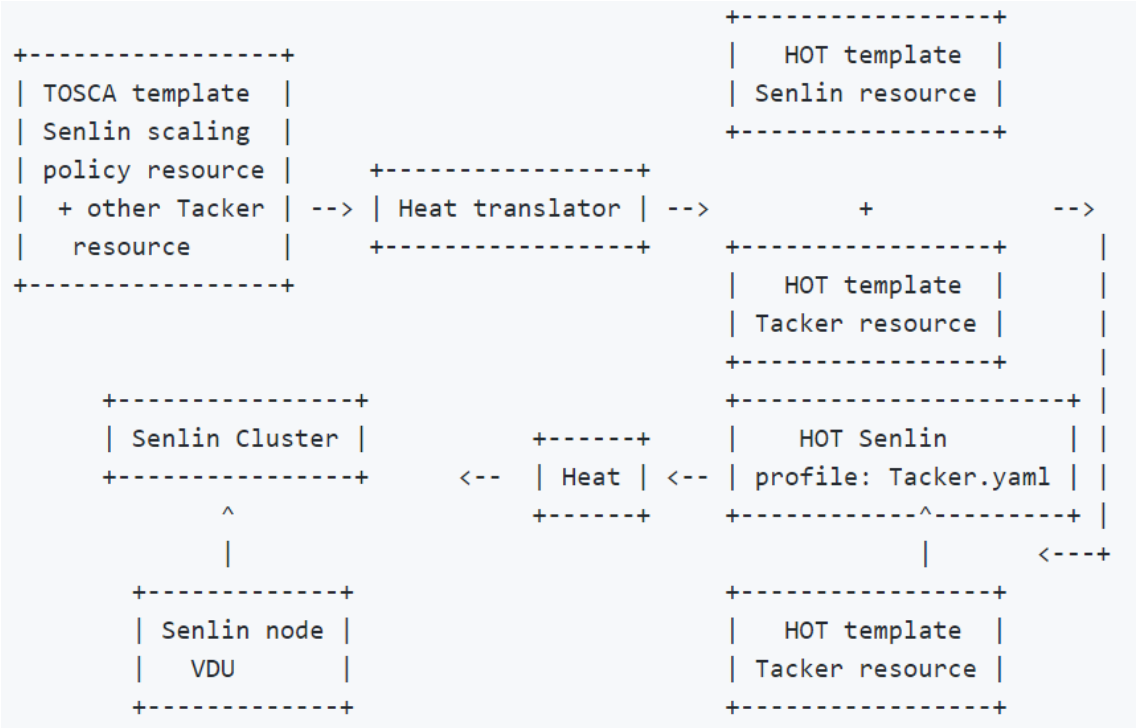


Figure 47 Whole workflow for HA

Because Tacker supports TOSCA format template, it is necessary to define all the resources(both Tacker and Senlin resources)in one TOSCA template. Then the TOSCA template will be translated by tosca-parser and heat-translator to HOT template which will be used to

deploy VDU cluster. Heat-translator has already supported most of the resources translation for Tacker and Senlin, So what is needed to be done is TOSCA template integration for Tacker and Senlin, and adding translation support for some resources in heat-translator. And in Tacker additional jobs like parsing the monitoring property of VDU needs to be done.

TOSCA template example for VDU auto-scaling management

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Demo example

metadata:
  template_name: sample-tosca-vnfd

topology_template:
  node_templates:

    VDU1:
      type: tosca.nodes.nfv.VDU.Tacker
      properties:
        image: cirros-0.3.4-x86_64-uec
        flavor: m1.tiny
        availability_zone: nova
        metadata: {metering.vnf: SG1}
        monitoring_policy:
          name: ping
          parameters:
            monitoring_delay: 45
            count: 3
            interval: 1
            timeout: 2
          actions:
            failure: respawn

    CP1:
```

```
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        management: true
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL1
        - virtualBinding:
            node: VDU1

  VL1:
    type: tosca.nodes.nfv.VL
    properties:
      network_name: net1
      vendor: Tacker

policies:
  - cluster_scaling:
      type: tosca.policies.Tacker.Scaling
      description: Cluster node autoscaling
      driver: Senlin(Or some name else to distinguish Heat driver)
      targets: [VDU1]
      properties:
        min_instances: 2
        max_instances: 10
        default_instances: 3
        increment: 1

  - vdu_cpu_usage_monitoring_policy:
        type: tosca.policies.tacker.Alarming
        triggers:
            vdu_hcpu_usage_scaling_out:
                event_type:
                    type: tosca.events.resource.cpu.utilization
                    implementation: Ceilometer
```

```
                metrics: cpu_util
                condition:
                    threshold: 50
                    constraint: utilization greater_than 50%
                    period: 60
                    evaluations: 1
                    method: avg
                    comparison_operator: gt
                metadata: SG1
                actions: [cluster_scaling]
```

The TOSCA template above does not introduce new resource type, only some attributes of the policies resource are different from the existing scaling policy supported by Tacker now. The 'driver' attribute is added to distinguish the Heat-autoscaling-group driver and Senlin driver. User can switch the auto-scaling backend by configuring different drivers. This TOSCA template will be translated to HOT template. There will be two HOT template created after the translation. One contains all the resources related to VDU which is like what is done in Tacker now. This HOT template will be referenced by Senlin profile when Senlin resources are created by Heat. Another HOT template only contains senlin related resources which will be passed to Heat for resource creation first. After Senlin resources 'profile' and 'cluster' are created, three senlin node(the number of nodes depends on the desired_capacity of cluster property in line 196) will be created according to the senlin profile. The senlin node actually is the VDU, it will be created during the node creation. After that the VDU nodes belong to a cluster. There is a receiver which is a webhook pointing to the cluster, if the resource usage triggers the alarm limit, the webhook will be executed to start scaling the VDU cluster. The scaling obeys the scale-in and scale-out policies attached to the cluster.

HOT template for Senlin resources

```
heat_template_version: 2016-04-08

description: >
  This template demonstrates creation of senlin resources for vm auto-scaling

resources:
  Senlin:
```

```
    type: OS::Senlin::Profile
    properties:
      type: os.nova.server-1.0
      properties:
        template: tacker.yaml

  Senlin_cluster:
    type: OS::Senlin::Cluster
    properties:
      desired_capacity: 3
      min_size: 2
      max_size: 10
      profile: {get_resource: Senlin}

  Senlin_scale_out_receiver:
    type: OS::Senlin::Receiver
    properties:
      action: CLUSTER_SCALE_OUT
      type: webhook
      cluster: {get_resource: Senlin_cluster}

  cluster_scaling_scale_out:
    type: OS::Senlin::Policy
    properties:
      type: senlin.policy.scaling-1.0
      bindings:
        - cluster: {get_resource: Senlin_cluster}
      properties:
        event: CLUSTER_SCALE_OUT
        adjustment:
          type: CHANGE_IN_CAPACITY
          number: 1

  scale_out_alarm:
    type: OS::Aodh::Alarm
```

```
    properties:
      meter_name: cpu_util
      statistic: avg
      period: 60
      evaluation_periods: 1
      threshold: 50
      repeat_actions: True
      alarm_actions:
        - {get_attr: [Senlin_scale_out_receiver, channel, alarm_url]}
      comparison_operator: gt

  Senlin_scale_in_receiver:
    type: OS::Senlin::Receiver
    properties:
      action: CLUSTER_SCALE_IN
      type: webhook
      cluster: {get_resource: Senlin_cluster}

  cluster_scaling_scale_in:
    type: OS::Senlin::Policy
    properties:
      type: senlin.policy.scaling-1.0
      bindings:
        - cluster: {get_resource: Senlin_cluster}
      properties:
        event: CLUSTER_SCALE_IN
        adjustment:
          type: CHANGE_IN_CAPACITY
          number: 1

  scale_in_alarm:
    type: OS::Aodh::Alarm
    properties:
      meter_name: cpu_util
      statistic: avg
```

```
        period: 60
        evaluation_periods: 1
        threshold: 50
        repeat_actions: True
        alarm_actions:
          - {get_attr: [Senlin_scale_in_receiver, channel, alarm_url]}
        comparison_operator: lt
```

HOT template for VDU

```
heat_template_version: 2016-04-08

description: >
  This template demonstrates a template for VDU

resources:
  VDU:
    type: OS::Nova::Server
    properties:
      image: cirros-0.3.4-x86_64-uec
      flavor: m1.tiny
      availability_zone: nova
      networks:
        - network: net1
```

Then how does the scaling feature work? Take this template for example, after all the Senlin resources(a cluster with three VDUs created on it, a receiver and scaling policy attached to the cluster, and also an alarm) are deployed by Heat, the scaling management can be left to Senlin completely. VDUs can be created and deleted automaticaly under the rules of the scaling policy according to the resource consumption. If users don't want to scale in/out VDUs automatically, they can also use 'tacker vnf-scale' command to control the scalability manually. The request will trigger senlin backend to execute the scale in/out actions. If user wants to auto-scale selective VDUs, they can simply add these VDUs information into the template, Senlin will adopt the nodes into Senlin's cluster and then control the scalability. This feature[2] is under implementation by Senlin team now.

5. Conclusion

In this document, we reviewed the High Availability with element technology such as monitoring driver, Clustering. In Monitoring part, we implemented Alarm monitoring driver which can trigger alarm when some parameter over the threshold. using this driver, we support availability for single service in one VM. Moreover, we implemented clustering service with senlin project. This is implementation of translation of Senlin scaling policy from TOSCA template to HOT template in Openstack/Heat-Translator project. This implementation could not support all clustering service, but, it is the one of the scenario of Clustering service for High availability. We will continue to research and implement architecture to support high availability services.

References

[1] Resiliency Requirements, ETSI GS NFV-REL 001, January, 2015

[2] Report on Models and Features for End-to-End Reliability, ETSI GS NFV-REL 003, April, 2016

[3] Report on Scalable Architectures for Reliability Management, ETSI GS NFV-REL 002, September, 2015

[4] ETSI,"Network Functions Virtualisation (NFV);Management and Orchestration;Or-Vi reference point - Interface and Information Model Specification",ETSI GS NFV-IFA 005 V2.1.1, April, 2016.

[5] ETSI,"Network Functions Virtualisation (NFV); Management and Orchestration; Vi-Vnfm reference point - Interface and Information Model Specification",ETSI GS NFV-IFA 006 V2.1.1, April, 2016.

[6] ETSI,"Network Functions Virtualisation (NFV);Management and Orchestration; Ve-Vnfm reference point - Interface and Information Model Specification",ETSI GS NFV-IFA 008 V2.1.1, October, 2016.

[7] High availability scenario analysis doc, OPNFV HA project, accessed at: https://wiki.opnfv.org/display/availability/High+Availability+For+OPNFV

[8] Clustering service Openstack Senlin, accessed at: https://wiki.openstack.org/wiki/Senlin

[9] (1, 2, 3) https://datatracker.ietf.org/doc/draft-ietf-sfc-nsh/?include_text=1

[10] http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf

[11] http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.pdf

[12] https://github.com/openstack/networking-sfc/blob/master/doc/source/api.rst

[13] https://github.com/opendaylight/sfc

[14] https://review.openstack.org/#/c/290771/

[15] Open source MANO (OSM) White paper: https://osm.etsi.org/images/OSM-Whitepaper-TechContent-ReleaseONE-FINAL.pdf

[16] OSM main website: https://osm.etsi.org/

[17] ONAP Wiki: https://wiki.onap.org/display/DW/Developer+Wiki

[18] Open Networking summit 2017: https://wiki.onap.org/display/DW/ONAP@ONS2017

[19] OpenBaton main website: https://openbaton.github.io/

[20] OPNFV Berlin summit: https://www.slideshare.net/OPNFV/summit-16-open-baton-overview

[21] https://docs.openstack.org/developer/networking-sfc/

[22] https://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining

[23] OpenDaylight SFC main page: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main

[24] OpenDaylight summit 2016: https://wiki.opendaylight.org/images/3/37/OpenDaylight-Summit-2016-OpenStack-SFC-Support.pdf

[25] OPNFV SFC wiki: https://wiki.opnfv.org/display/sfc/Service+Function+Chaining+Home

[26] https://www.slideshare.net/OPNFV/summit-16-service-function-chaining-demo-and-usage.

[27] https://wiki.openstack.org/wiki/Telemetry

[28] OpenStack Ceilometer: https://docs.openstack.org/developer/ceilometer/

[29] OpenStack Aodh: https://docs.openstack.org/developer/aodh/

[30] OpenStack Monasca main page: https://wiki.openstack.org/wiki/Monasca

[31] https://www.netways.de/fileadmin/images/Events_Trainings/Events/OSMC/2016/Slides/Rol and_Hochmuth_-_Monasca_-_Monitoring-as-a-Service__at-Scale__.pdf

[32] Zabix tutorial: http://www.zabbix.com/downloads/ZABBIX%20Manual%20v1.6.pdf.

[33] https://assets.nagios.com/downloads/nagioscore/docs/nagios-3.pdf

[34] http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.pdf

[35] https://docs.openstack.org/developer/heat/template_guide/openstack.html #OS::Aodh::Alarm

[36] https://wiki.openstack.org/wiki/Heat/AutoScaling

[37] Clustering service Openstack Senlin, accessed at: https://wiki.openstack.org/wiki/Senlin

K-ONE Technical Document

○ The technical contents of this document can be changed without a notification due to the progress of the project.

○ Refer website below for getting more information of this document.

(Homepage: http://opennetworking.kr/projects/k-one-collaboration-project/wiki, E-mail: k1@opennetworking.kr)

Written by K-ONE Consortium

Written in 20167/05