K-ONE 기술 문서 #20
OPNFV/OpenDaylight SFC 및 OpenStack Tacker 기반의 VNFFG 구성 알고리즘

Document No. K-ONE #20
Version 1.0
Date 2017-05-12
Author(s) 서동은, 이재욱, 백호성

K-ONE

ONK
OpenNetworkingKorea

■ 문서의 연혁

버전	날짜	작성자	내용
초안 - 0.1	2017. 03. 15	서동은, 이재욱, 백호성	초안 작성
0.2	2017. 03. 31	서동은, 이재욱, 백호성	내용 추가
0.3	2017. 04. 21	서동은, 이재욱, 백호성	기능 테스트 및 코드 분석 작성
1.0	2017. 05. 12	서동은, 이재욱, 백호성	검토

본 문서는 2016년도 정부(미래창조과학부)의 재원으로 정보통신 기술진흥센터의 지원을 받아 수행된 연구임 (No. B0190-16-2012, 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. B0190-16-2012, Global SDN/NFV OpenSource Software Core Module/Function Development)

기술문서 요약

본 고는 OPNFV/OpenDaylight SFC 및 OpenStack Tacker 기반의 VNFFG 구성 알고리즘에 대한 기술문서이다. 본 고는 다음과 같이 구성된다. 1장은 VNF Forwarding Graph를 위한 오픈소스 프로젝트들에 대해 알아보고, 2장에서는 OpenStack Tacker 프로젝트의 NFV Orchestration 기능에 대해 자세히 알아본다. 3장에서는 OpenStack Tacker의 기타 기능에 대하여 알아보고, 4장에서는 OpenStack Tacker 기반의 VNFFG 구성 테스트를 수행한다. 마지막으로 5장에서는 OpenStack Tacker 기반의 VNFFG 구성 알고리즘을 제안한다.

Contents

K-ONE #20. OPNFV/OpenDaylight SFC 및 OpenStack Tacker 기반의 VNFFG 구성 알고리즘

그림 목차

K-ONE #20. OPNFV/OpenDaylight SFC 및 OpenStack Tacker 기반의 VNFFG 구성 알고리즘

1. VNF Forwarding Graph를 위한 오픈 소스 프로젝트

최근 네트워크가 다양한 서비스를 제공하면서 보안 유지나 성능 향상 등을 위해 Firewall, Deep Packet Inspection (DPI), Network Address Translation (NAT) 등과 같은 미들박스 (Middlebox), 즉 서비스 기능에 대한 의존성이 높아지고 있다. 또한, 특정 플로우 (flow)가 여러 네트워크 서비스 기능을 필요로 하는 경우에는, 서비스들이 해당 플로우에 대하여 논리적인 순서에 따라 적용될 수 있는 합리적인 방법이 필요하다. 즉, 여러 요구사항에 따라 통신 사업자의 목적과 정책에 맞추어 서비스 기능들을 하나의 논리적인 연결로 순서화하는 서비스 기능 체이닝 기술이 주목 받고 있으며, 이에 대한 구조 및 프로토콜 표준화, 실제 구현에 관련된 다양한 연구가 학계, 산업계를 불문하고 활발하게 이루어지고 있다.

이러한 서비스 기능 체이닝은 Software Defined Networking (SDN) 기술 및 Network Function Virtualization (NFV) 기술을 통하여 더욱 효과적이고 유연하게 제공될 수 있다. SDN은 기존 네트워크 장비의 제어 평면과 데이터 평면을 분리시켜 논리적으로 중앙 집중된 컨트롤러를 통해 네트워크 제어 기능을 제공한다. 따라서, SDN 기술을 통해 전체 네트워크 뷰를 유지하며, 요구 사항 및 가변적인 네트워크 상황에 따라 동적으로 서비스 기능 체인을 구성할 수 있다. 한편, NFV 기술이란 고가의 네트워크 장비에 대한 투자비용 (CAPEX) 및 운용비용 (OPEX) 문제를 해결하기 위하여, 다양한 네트워크 서비스 기능들을 범용의 표준 서버 하드웨어에서 실행될 수 있는 소프트웨어로 구현하여 운용하는 것을 의미한다. 관련 표준화 노력의 일환으로, 통신 분야 표준화 단체인 ETSI (European Telecommunications Standards Institute)는 전 세계 주요 통신사업자들과 NFV ISG (Industry Specification Group)을 출범하고, 활발한 표준화 활동을 진행하고 있다.

현재 오픈소스 커뮤니티 기반의 SDN 컨트롤러 개발 프로젝트인 OpenDaylight [1]에서는 서비스 체이닝을 제공하기 위하여, 하위 프로젝트로 SFC 프로젝트를 유지하며 활발한 개발을 진행 중에 있다. OpenDaylight SFC 프로젝트는 IETF SFC [2] 작업 그룹에서 제안한 SFC 구조를 기반으로 개발을 진행중에 있다. 또한, ETSI NFV ISG [3] 에서 제안한 NFV 구조를 기반으로 NFV 레퍼런스 플랫폼을 개발하고 있는 프로젝트인 OPNFV (Open Platform for NFV) [4] 에서도 하위 프로젝트로 SFC 프로젝트가 존재하며 서비스 체인이 기술 개발을 진행하고 있다. 한편, 기존의 OpenDaylight SFC 프로젝트에서는 OPNFV 플랫폼 상에서의 가상화된 네트워크 기능들 (즉, Virtual Network Function (VNF))에 대한 서비스 기능 체이닝을 고려하지 않고 있기 때문에, OPNFV SFC 프로젝트에서는 OpenDaylight SFC 프로젝트를 업스트림 프로젝트로 하여 추가적으로 필요한 인터페이스 및 기능들에 대한 정의 및 개발을 진행 중이다. 다음으로 1.1과 1.2에서는 OpenDaylight 와 OPNFV에서 개발되고 있는 각각의 SFC 프로젝트에 대해 분석한다. 그리고 1.3에서는 OpenStack에서 개발되고 있는 Tacker 프로젝트에 대해 분석한다.

1.1. OpenDaylight SFC

서비스 기능 체이닝 기술이 중요한 요구사항으로 부상되고 있음에 따라 세계적으로 SDN에서 서비스 체이닝에 대한 다양한 형태의 개념 검증 및 프로토타이핑이 진행되고 있다. 대표적으로 오픈 소스 기반의 SDN 컨트롤러인 OpenDaylight 의 Beryllium 버전에서는 서비스 체이닝에 대한 소스 코드를 포함하고 있다. 1.1에서는 OpenDaylight SFC 프로젝트에서 서비스 기능 체이닝 연구 동향 및 구조에 대해 분석한다.

1.1.1. OpenDaylight 프로젝트

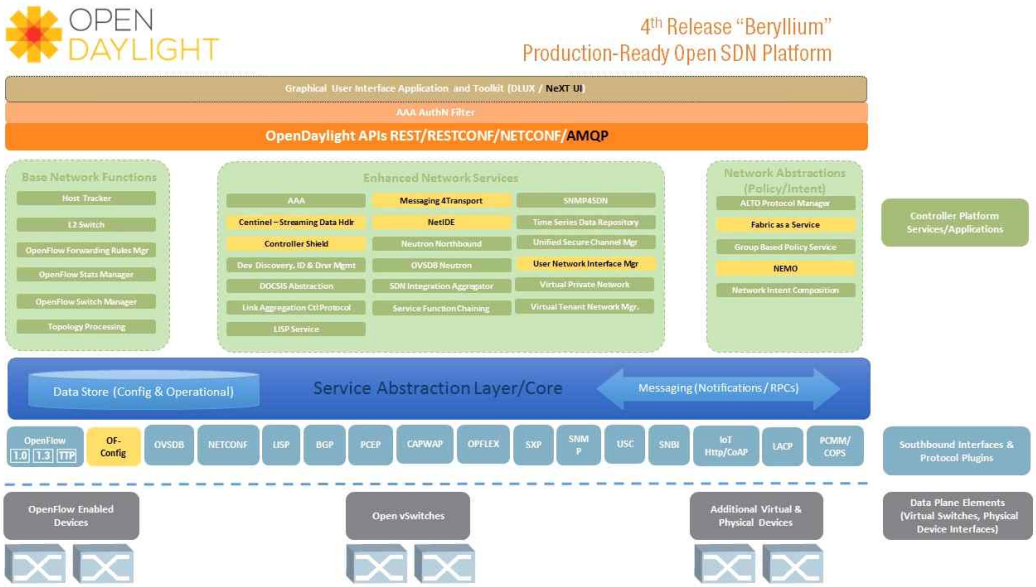

그림 1. OpenDaylight 구조

위의 그림은 SDN 기반 오픈 소스 컨트롤러인 Opendaylight 프로젝트의 버전인 Beryllium의 논리적인 구조를 나타낸다. Opendaylight은 다양한 네트워크 서비스들을 모듈화된 OSGi 플러그인 형태로 제공한다. 각 모듈은 Opendaylight 프로젝트에 참여하고 있는 다양한 벤더들에 의해 구현된다. 먼저 가장 상위 계층인 Network Apps & Orchestration계층은 네트워크 제어 및 모니터링을 할 수 있는 네트워크 애플리케이션들로 구성되어 있다. 예를 들어, 네트워크 관리자는 DLUX UI 애플리케이션을 통해 SFC 서비스를 이용할 수 있다. 중간 계층에 위치한 Controller Platform 계층은 다양한 네트워크 서비스들 (네트워크 토폴로지 매니저, 스위치 매니저, SFC, GBP 등) 및 OpenFlow, LISP, SNMP, NETCONF 등의 다양한 Southbound 프로토콜들을 플러그인 형태로 네트워크 애플리케이션에게 제공한다.

하위 계층의 Physical & Virtual Network Devices 계층은 네트워크의 모든 종단점을 연결하는 물리적인 혹은 가상화된 스위치와 라우터 등으로 구성된다. 서비스 체이닝 및 Group Based Policy 또한 Opendaylight 프로젝트의 하위 프로젝트로써 플러그인 형태로 개발 중에 있다.

1.1.2. OpenDaylight SFC 프로젝트

Opendaylight 의 SFC 개발 프로젝트에서는 IETF SFC 작업 그룹에서 제안하는 SFC 구조를 기반으로 하여 개발을 진행하고 있다. 1.1.2에서는 해당 구조에서 등장하는 핵심 용어에 대한 설명을 나타낸다.

- Service Function Chain (SFC) : 서비스들의 논리적인 순서를 나타낸다.
- Service Function Path (SFP) : 주소를 갖는 서비스 인스턴스들의 순서를 나타낸다. 특정 SFP는 유일한 SFP 식별자를 통해 구별된다.
- Service Index (SI) : 특정 트래픽 (플로우 혹은 패킷)이 받아야 할 서비스 인스턴스들의 갯수를 나타내며 SFP의 길이에서 서비스를 받을 때 마다 1 씩 감소한다.
- Network Service Header (NSH) [5] : NSH에는 SFP 식별자 정보와 SI 정보가 담겨 있다. NSH를 통해 서비스 평면이라 불리는 오버레이 네트워크를 구성한다.
- Service Function (SF) : 미들박스 기능을 제공하는 요소이다.
- Service Function Forwarder (SFF) : SFP를 따라 스위칭 역할을 수행하는 요소이다. 서비스 오버레이 네트워크의 스위치로 동작한다.
- Service Classifier (SC) : 유입되는 트래픽을 분류하여 NSH 인캡슐레이션을 수행하는 요소이다.

그림 2는 서비스 기능 체이닝의 논리적인 구조를 나타낸다. SC는 미리 정의된 정책에 기반하여 유입된 외부 트래픽이 어떠한 SFP를 따르게 될 지를 정하고 결정된 SFP에 대한 식별자 정보를 NSH에 추가한다. NSH 인캡슐레이션이 된 패킷들은 경로상의 첫 번째 SFF로 전달된다. SFF는 SFP 식별자, SI, 다음 서비스 홉 (NH: Next Hop)으로 구성된 서비스 라우팅 테이블을 유지하여, NSH 패킷이 도달하면 NSH의 SFP 식별자와 SI에 매칭되는 NH로 패킷을 포워딩 한다.

그림 3은 Opendaylight SFC의 동작 과정을 나타낸다. 박스 부분은 Opendaylight 컨트롤러의 기능을 나타낸다. 먼저 Opendaylight SFC 웹 UI를 통해 SFP를 구성,

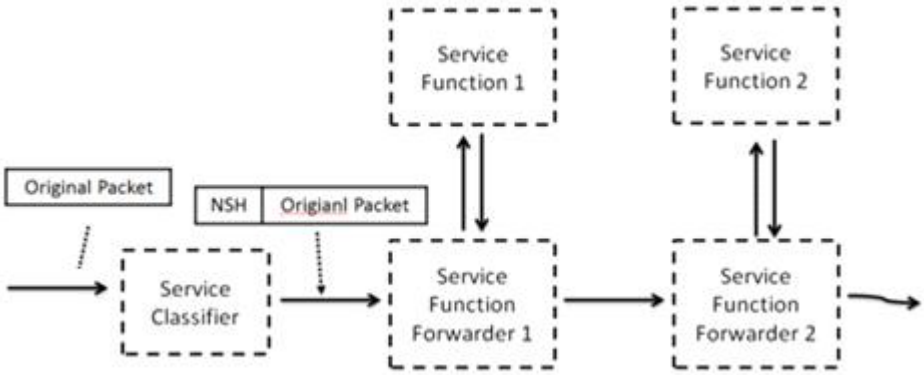

그림 2. OpenDaylight SFC 논리적 구조

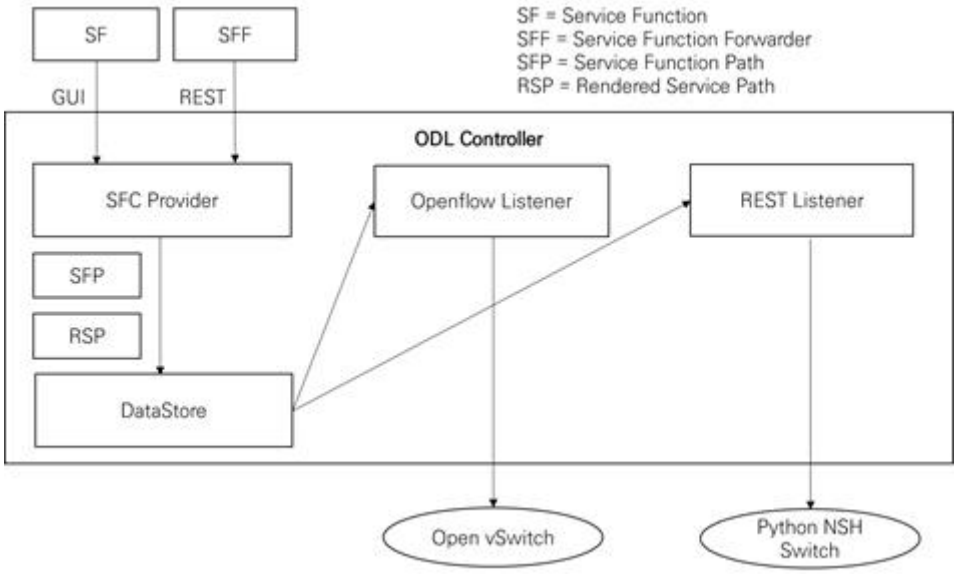

그림 3. OpenDaylight SFC의 동작과정

컨트롤러의 SFC 저장소에 저장한다. SFC 저장소에 등록된 SBI 플러그인들은 SFP 정보를 전달받고, 이를 기반으로 데이터 평면에 위치한 SFF의 서비스 라우팅 테이블을 조작한다. 이 때 OpenFlow 프로토콜 혹은 REST 프로토콜 등으로 SFF를 조작할 수 있다.

1.2. OPNFV SFC

ETSI ISG 산하에 NFV라는 표준 그룹은 NFV 기술 분야에서 통신사업자 및 산업체가 요구하는 산업 규격을 정의한다. ETSI ISG가 제안한 규격을 기반으로 OPNFV(Open Platform for Network Functions Virtualization)는 NFV에 대한 오픈 플랫폼을 개발하고 있다. 아래 그림은 NFV기술을 실현하기 위한 프레임워크를 나

타내는 그림이다.

NFV 프레임워크가 만족해야하는 구조적인 특징은 다음과 같다. Virtualized Network Functions (VNF)를 관리하고 조율할 수 있는 EMS, OSS/BSS와 같은 시스템이 필요하다. 그리고 VNF는 서로 다른 하이퍼바이저 위에서 동작하고 자유롭게 컴퓨팅 자원에 접근이 가능해야 한다. VNF를 하드웨어적으로 구성할 수 있는 요건이 만족된다면, Virtualized Infrastructure Manager (VIM)을 통해 물리적인 하드웨어 자원을 추상화하여 VNF의 자원의 상태 및 가상 네트워크를 구성한다. Orchestrator 은 VIM의 도움으로 가상화된 네트워크 상태를 바탕으로 VNF Forwarding Graph (VNFFG)를 구성하며, 정책 및 결정을 내리는 역할을 하고, 그리고 컴퓨팅, 저장소, 네트워크 기능을 지원하는 물리적인 하드웨어 자원을 가상화하여 VNF를 실행시킬 수 있도록 지원하는 역할을 하는 Network Function Virtualization Infrastructure (NFVI) 구성 요소와 네트워크 서비스의 자원을 조율한다.

1.2.1. OPNFV 프로젝트

OPNFV는 네트워크 장비 및 기능을 가상화하기 위한 NFV의 오픈 플랫폼이다. 네트워크 기술이 빠르게 발전함에 따라 고가의 네트워크 장비에 대한 투자비용과 운용비용 문제를 해결하기 위해 네트워크 및 장비를 가상화하기 위한 NFV 기술에 대한 연구가 활발해졌고, NFV 플랫폼 및 솔루션 개발에 대한 필요성이 대두되었다. 이에 따라 최근 Linux Foundation이 공개 소프트웨어 기반의 OPNFV 프로젝트가 발표하였다. OPNFV에서 개발하려고 하는 부분은 각 네트워크의 요소 자체의 기능을 개발하는 것보다 각 네트워크 요소들 간의 인터페이스가 서로 간에 호환이 잘 될 수 있도록 지원하는 것이다. 즉, 모든 통신 산업 개발자들이 NFV를 개발하는데 있어서 각 요소들이 쉽게 장착될 수 있도록 호환성을 제공하는 데 목적을 둔다. 다음 장에는 OPNFV 진영에서 개발진행중인 SFC 프로젝트에 대해 분석한다.

1.2.2. OPNFV SFC 프로젝트

OPNFV SFC 프로젝트는 오픈 소스 기반의 SDN 컨트롤러인 OpenDaylight을 활용하여 OPNFV 플랫폼 상에서 관리하는 VNF들 간에 서비스 체인을 구성할 수 있는 기능을 구현하는 것을 목표로 한다. 한편, Opendaylight의 버전인 Beryllium에서는 IETF SFC 구조를 기반으로 Opendaylight 컨트롤러가 관리하는 데이터 평면상에 서비스 체인을 구성하는 기능을 제공하고 있다. 기존의 OpenDaylight SFC에서는 OPNFV 플랫폼 상에서의 서비스 체이닝을 고려하고 있지 않기 때문에, OPNFV SFC에서는 Opendaylight SFC 프로젝트를 업스트림 프로젝트로 하여 추가적으로 필요한 인터페이스 및 기능들에 대한 정의 및 개발을 계획하고 있다.

아래 그림은 OPNFV SFC프로젝트의 초기 구상도를 나타낸다. VIM (Virtualized Infrastructure Manager)에서 네트워크 자원에 대한 관리는 Opendaylight 컨트롤러를 사용하고, 컴퓨팅 자원에 대한 관리는 OpenStack을 사용함을 알 수 있다. VNFM (VNF Manager)로는 OpenStack Tacker가 사용된다. 한편, 노란색 박스로 표현된 Opendaylight SFC, SFF, SC는 OPNFV SFC 프로젝트에서 개발을 진행하는 컴포넌트들을 나타낸다. Opendaylight SFC는 OpenDaylight 컨트롤러의 SFC 플러그인을 의미하며 VNFM으로부터 전달받은 서비스 체인 요구사항을 데이터 평면에 렌더링하기 위해 데이터 평면의 SC 및 SFF를 조작한다. 먼저 SC는 미리 정의된 정책에 따라 유입되는 트래픽을 분류하여 NSH 인캡슐레이션을 수행하는 요소이다. 이 때, Opendaylight 의 GBP (Group-based Policy) 프로젝트에서 정의하는 EPG (End-point Group) 형식으로 트래픽을 분류한다. 그림 5의 예시에서는 클라이언트들 및 서버들이 각각 GBP EPG1 그리고 GBP EPG2로 정의된다. 이 때, 정의된 정책은 EPG1과 EPG2사이의 트래픽들에 대해서 VNF1 및 VNF2로 정의되는 서비스 체인을 지나도록 하는 것이다. 왼쪽의 SC는 이에 따라 VM으로부터 들어오는 트래픽들 중 소스가 EPG1이고 목적지가 EPG2인 트래픽에 대해 VNF1->VNF2에 해당하는 SFP 식별자 정보를 담은 NSH 인캡슐레이션을 수행한다. 오른쪽 SC는 VM으로부터 들어오는 트래픽들 중 소스가 EPG2이고 목적지가 EPG1인 트래픽에 대해 VNF2->VNF1에 해당하는 SFP 식별자 정보를 담은 NSH 인캡슐레이션을 수행한다. 인캡슐레이션이 끝나면 SFP상의 첫 번째 SF가 연결된 SFF에게로 패킷을 전달한다. 한편 SFF는 SFP를 따라 스위칭 역할을 수행하는 요소로 Opendaylight SFC는 SFF의 서비스 라우팅 테이블에 VNF1->VNF2 그리고 VNF1->VNF2로 정의되는 서비스 체인에 대한 <SFP 식별자, SI> 엔트리와 이에 대한 NH들을 설치한다. SFF는 SC로부터 NSH 인캡슐레이션된 패킷을 받으면 매칭되는 엔트리의 NH으로 패킷을 전달한다. 이러한 일련의 과정을 거침으로써 EPG1->EPG2로의 트래픽은 VNF1->VNF2 순서로 서비스를 받게 되고 EPG2->EPG1로의 트래픽은 VNF2->VNF1 순서로 서비스를 받게 된다.

1.3. OpenStack Tacker

본 장에서는 클라우드 컴퓨팅 오픈 소스 프로젝트 OpenStack 프로젝트의 하위 프로젝트인 Tacker 프로젝트에 대해 설명한다. Tacker 프로젝트는 위의 그림에서 VNFO와 NFVO 역할을 타겟으로 하는 프로젝트이다. VNF를 고려하여 서비스 기능 체이닝을 구성하기 위해서는 NFV 환경에서 VNF를 관리하는 OpenStack과 같은 플랫폼이 필요하다. 1.3.1에서는 OpenStack 프로젝트에 대해 분석하고, 1.3.2에서는 OpenStack의 하위 프로젝트인 Tacker 프로젝트에 대해 분석한다.

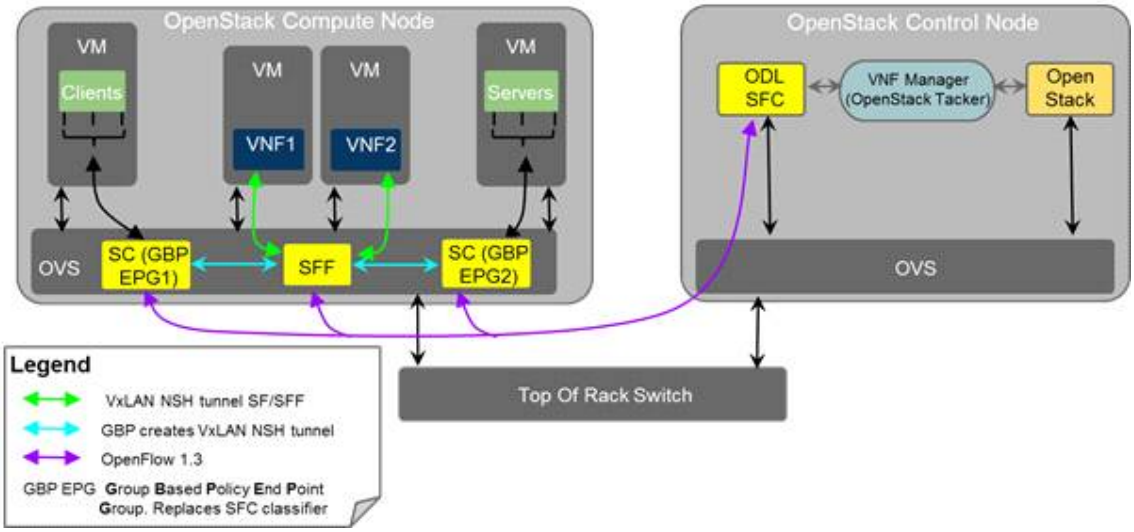

그림 4. OPNFV SFC 초기구상도

1.3.1. OpenStack 프로젝트

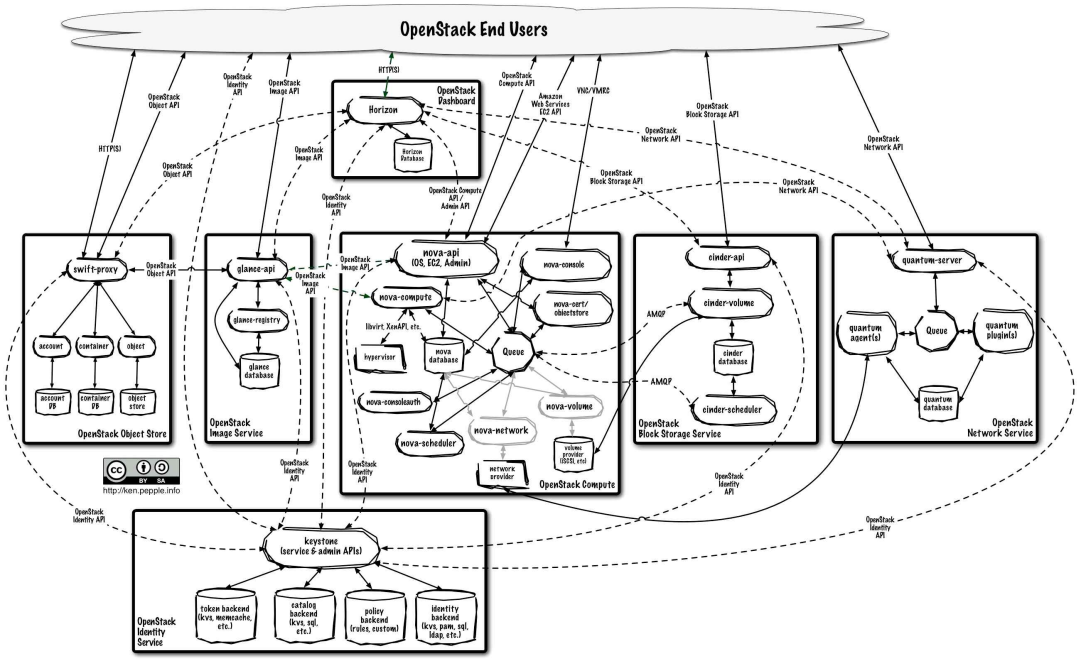

그림 5. OpenStack 개념적 구조

OpenStack은 클라우드 컴퓨팅 오픈 소스 프로젝트로, 2010년 미 항공우주국과 락스페이스사가 오픈스택을 설립하였고, 2012년 9월 비영리 단체인 오픈스택 파운데이션 (OpenStack Foundation) 으로 다시 출범하게 되었다. 첫 번째 버전인 오스틴 (Austin) 을 시작으로, 2016년 10월 뉴튼 (Newton) 버전이 릴리즈되었다. 매 해 6개

월 주기로 2번씩 새로운 버전이 릴리즈 되고 있다. OpenStack의 개념적인 구조는 아래 그림과 같다. 클라우드 컴퓨팅 시스템을 구축하기 위한 것으로, 크게 컴퓨트 서비스를 Nova로, 오브젝트 스토리지 서비스는 Swift, 운영체제 이미지 관리를 위한 서비스는 Glance로 구성되어 있다. 또한 위의 Nova, Swift, Glance와 같은 서비스들의 인증을 담당하는 Keystone, 서비스들을 쉽게 사용하기 위해 사용자들에게 대시보드를 제공하는 Horizon, 오케스트레이션 서비스르 제공하는 Heat, 미터링 서비스 Ceilometer도 OpenStack의 버전업에 따라 새롭게 서비스가 추가되었다. 그리고 다음 장에서 Kilo 버전부터 새롭게 추가된 Tacker 프로젝트에 대해서 분석한다.

1.3.2. OpenStack Tacker 프로젝트

본 장에서는 OpenStack의 Kilo 버전부터 새롭게 추가된 Tacker 프로젝트에 대해 분석한다. Tacker 프로젝트는 NFV 환경에서 VNF를 관리하는 VNF Manager 기능과 NFV 서비스들을 관장하고 관리하는 NFV Orchestrator 기능을 지원하기 위한 프로젝트이다. 즉, Tacker 프로젝트에서 제공하는 기능은 아래 그림에서 빨간색 박스 부분이다. 가상화된 리소스를 바탕으로 VNF를 관리하고 구성하는 역할을 담당하는 VNFM과 NFV 환경에서 네트워크의 전체적인 조율 및 관리에 대한 기능을 수행하는 NFVO로 구성된다. VNFM과 NFVO에서 VNF의 배치와 네트워크 서비스를 관리하는 데는 Network Service Descriptor (NSD)와 VNF 카탈로그에 저장되어 있는 Virtualized Network Function Descriptor (VNFD)를 기반으로 이루어진다.

Tacker의 구조는 아래 그림과 같다. VIM 부분의 VNF 생성을 위한 부분과 관리를 위한 드라이버 모니터링을 위한 드라이버가 존재하며, 현재 ping과 HTTPping 그리고 alarm-based monitoring이 있다. NFVO에서는 포워딩그래프와 네트워크 서비스 오케스트레이션, 다중 VIM 환경에서의 VNF 배치. 리소스 체크 및 할당 위한 VNFM으로 구성되어 있다. Tacker는 ETSI에서 논의되고 있는 기본 형태를 기반으로 제안하고 있기 때문에 특정 벤더에 종속되지 않고 멀티 벤더로부터 VNF를 배치하는 것이 가능하다.

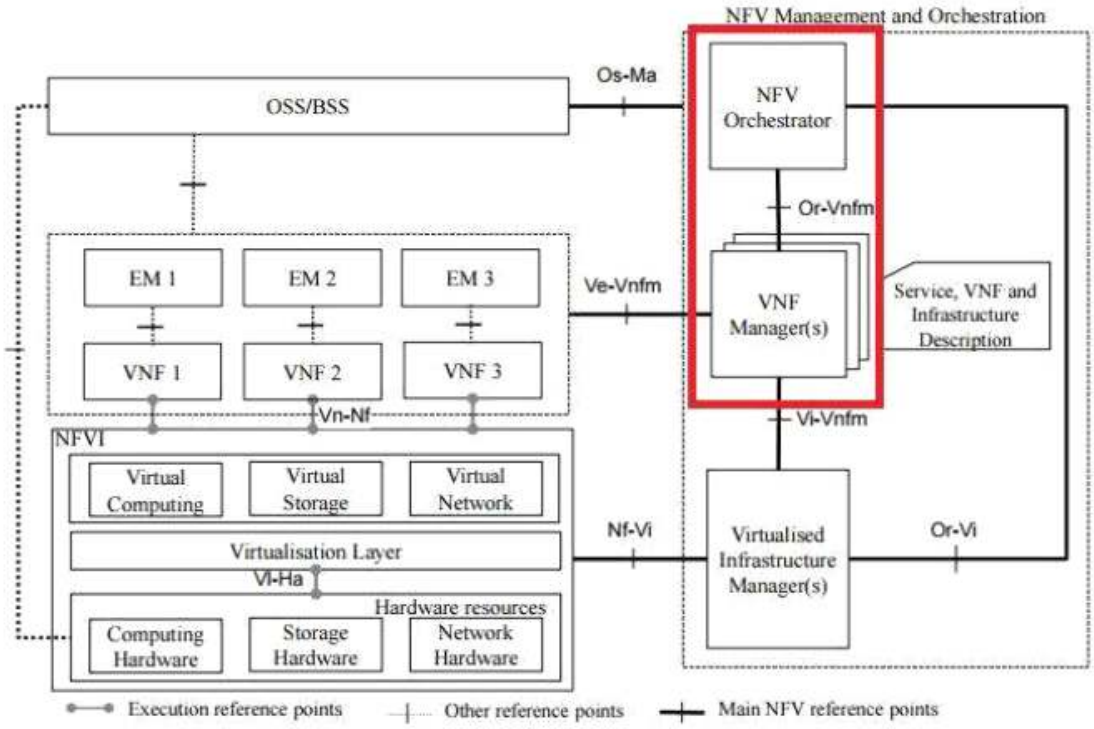

그림 6. OpenStack Tacker target 기능

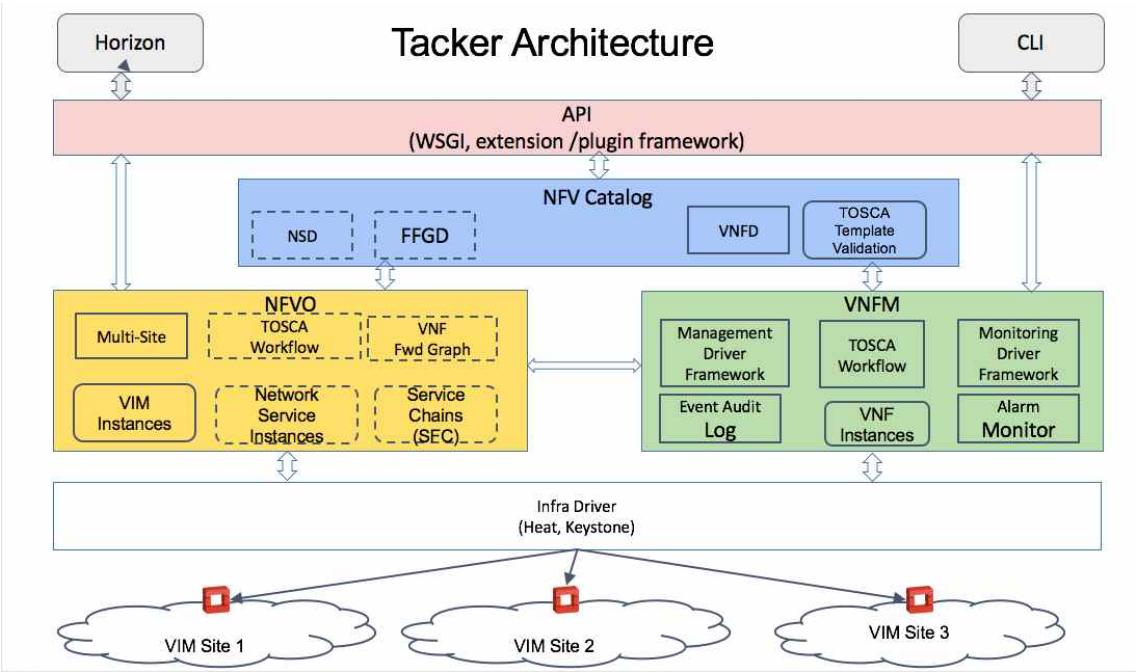

그림 7. Tacker의 구조

2. OpenStack Tacker의 NFV Orchestration 기능

2.1. VNF Manager

Tacker 프로젝트는 NFV Orchestration 기능을 타겟으로 한 프로젝트로 대표적으로 VNF manager 기능이 있다. VNF manager의 기능은 다음과 같다. OpenStack과 같은 VIM 위에서 VNF의 lifetime을 관리한다. 즉, VNF의 생성 및 삭제, 모니터링, 스케일링을 통해 전반적으로 VNF를 관리한다. VNF를 배치하는 과정은 다음과 같다. Tacker 프로젝트는 template 기반으로 VNF를 생성 및 관리하기 때문에, Horizon 혹은 Command line을 통해 Tacker VNFD catalog에 TOSCA VNFD template을 온보딩 시켜야 한다. VNFD에는 기본적으로 VNFD의 id, 기본정보, vdu 정보, virtual link 정보, 모니터링 및 오토 스케일링 관련 파라미터 등이 포함되어 있다. 이를 통해, VNF의 생성뿐만 아니라, 생성이후에 필요한 관리 기능에 대하 정의 및 모니터링 이후의 액션에 대한 정의도 포함되어 있어 유용하다.

VNFD template을 온보딩하는 명령어는 다음과 같다.

```
tacker vnfd-create --vnfd-file <yaml file path> <VNFD-NAME>
```

VNFD template 의 간단한 예제로 다음과 같다. 아래 코드는 http monitoing 기능을 하는 VNFD template이다. VNFD template을 온보딩 시킨 후 VNF를 deploy 시키는 방법은 두가지로 다음과 같다.

1. Tacker에서 가장 일반적으로 쓰이는 방법으로 Tacker Catalog에 온보딩되어 있는 VNFD template을 이용하여 VNF를 생성한다. 이 방법을 통해 VNF를 생성시키는 명령어는 다음과 같다.

```
tacker vnf-create --vnfd-name <VNFD-FILE-NAME> <VNF-NAME>
```

즉, 미리 온보딩 되어있는 VNFD template을 이용하여 VNF를 생성한다.

2. TOSCA template으로 Tacker Catalog에 온보딩 시키지 않고 직접적으로 VNF를 생성한다. 이 방법은 NFV Catalog가 Tacker에 밖에 위치하여 Tacker가 단지 NFV workflow engine으로만 사용될 때 주로 쓰이는 방법이다. 이 방법을 통해 VNF를 생성하는 명령어는 다음과 같다.

```
tacker vnf-create --vnfd-template <VNFD-FILE-NAME> <VNF-NAME>
```

즉, VNF를 생성할 때 참조할 VNFD template을 NFV Catalog에서 직접 가져온다.

```
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
      capabilities:
        nfv_compute:
          properties:
            num_cpus: 1
            mem_size: 2 GB
            disk_size: 20 GB
      properties:
        image: ubuntu
        availability_zone: nova
        mgmt_driver: noop
        config: |
          param0: key1
          param1: key2
        monitoring_policy:
          name: http_ping
          parameters:
            retry: 5
            timeout: 10
            port: 8000
          actions:
            failure: respawn

    CP1:
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        management: true
        order: 0
        anti_spoofing_protection: false
      requirements:
```

그림 8. http monitoring VNFD template (1/2)

```
    CP2:
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        order: 1
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL2
        - virtualBinding:
            node: VDU1

    CP3:
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        order: 2
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL3
        - virtualBinding:
            node: VDU1

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

    VL2:
      type: tosca.nodes.nfv.VL
      properties:
        network_name: net0
        vendor: Tacker

    VL3:
      type: tosca.nodes.nfv.VL
      properties:
        network_name: net1
        vendor: Tacker
```

그림 9. http monitoring VNFD template (2/2)

현재 VNF Manager에 구현되어 있는 명령어는 다음과 같다.

vnfd-list는 tacker의 Catalog에 온보딩 되어있는 vnfd의 리스트를 나타낸다. Onboarding 되어있지 않고 직접 VNF를 생성하는 데 사용된 VNFD list를 보기 위해서는 '–template-source all'flag를 사용하여야 한다.

다음으로, VNFM의 상태를 나타내는 명령어로는 vim-list, vnf-list, vnf-show, vnfd-show가 있다. 그리고 생성된 VNF 및 VNFD를 제거할 수 있고 command는 다음과 같다.

```
tacker vnf-delete <VNF_ID/NAME>
tacker vnfd-delete <VNFD_ID/NAME>
```

2.2. VNFFG Manager

VNFFG Manager는 Tacker project에 새롭게 추가된 기능이다. NFV 진영에서는 단순히 VNF를 생성하고 관리하는 단계를 넘어, 생성된 VNF를 SFC 기술을 이용하여 플로우의 요구사항에 따라 VNF를 논리적으로 연결하여 그래프를 구성하는 단계까지를 필요로 한다. 최종적으로, 단순히 논리적이고 추상적으로 구성된 그래프를 오버레이 네트워크 아래의 물리적인 네트워크에 실질적으로 렌더링하는 것이다.

현재 Tacker project에서는 단일 경로의 SFC만 지원 가능한 상태이다. 이 문제를 해결하기 위해서, 다중 경로 그래프를 여러 개의 단일 경로 SFC로 파싱하여 더 확장된 VNFFG를 생성할 수 있도록 한다. 이 feature는 체인 최적화 문제로 고려할 사항이 많아서 다음 버전에서 다루어질 것이다.

이 기능은 Tacker project에서 새롭게 추가되었기 때문에, Tacker Client, Horizon에도 추가된 기능에 맞게 추가되는 부분이 있다. 수정되는 부분은 다음과 같다.

1. tacker-horizon에 VNFFG 탭이 추가되었다. 이 탭을 통해서 미리 생성된 VNF를 이용하여 그래프를 생성할 수 있다. 이 부분은 TOSCA VNFFGD를 이용한다.
2. tacker client에서 tacker server에 CRUD VNFFG call을 통과하도록 하여야 한다.
3. tacker server는 NFVO 확장 부분과 VNFFG 기능과 자원을 통합하기 위한 플러그인 부분의 업데이트가 필요하다.
4. VNFFG를 위한 드라이버가 필요하다. 현재 SFC를 생성하기 위한 드라이버로는 뉴트론 기반의 networking-sfc와 OpenDaylight SFC가 있다. 이후에 VNFFG 플러그인을 지원할 드라이버는 networking-sfc 드라이버가 될 것이다. OpenDaylight SFC 드라이버는 networking-sfc의 별도의 spec으로 networking-sfc로의 드라이버로 처리될 것이다. 구조는 아래 그림과 같다.

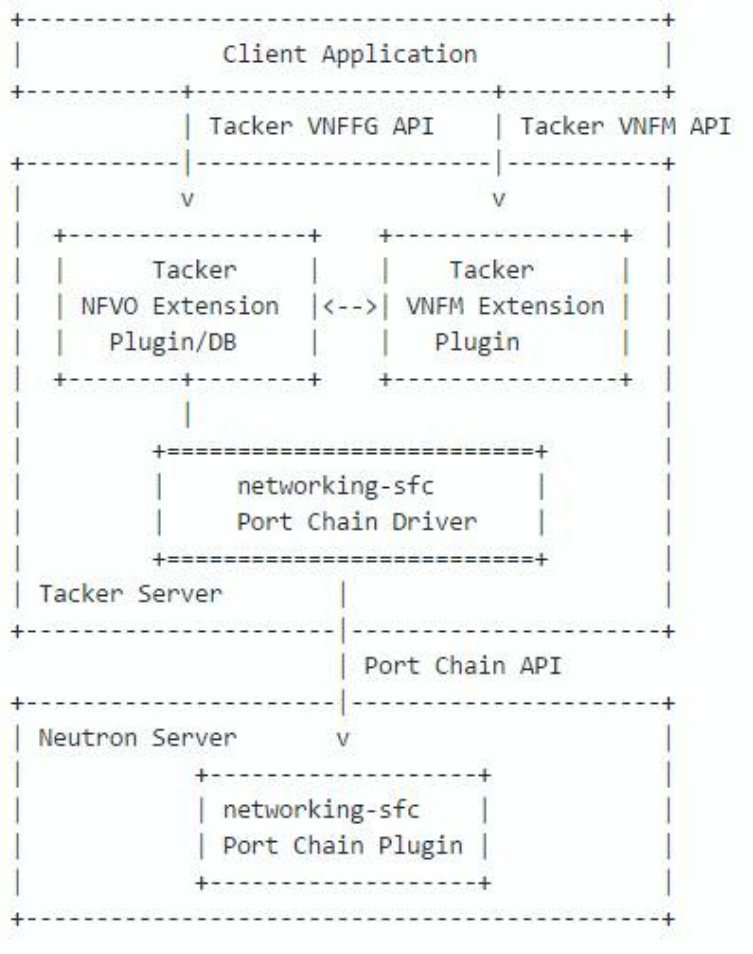

그림 10. networking-sfc 드라이버가 추가된 tacker 구조

데이터 모델에서 추가되는 부분은 다음과 같다. 'vnffgd', 'vnffgd_nfp', 'vnffg', 'vnffg_nfp', 'vnffg_chain','vnffg_classifier' 테이블들의 생성이 추가된다. 각 테이블의 기능을 간단히 설명하면 다음과 같다. 'vnffgd'테이블은 VNFFG 인스턴스 생성 관련 속성과 'vnffg_nfp' 테이블에 저장된 관련 NFP 및 'vnffg_chain'및 'vnffg_classifier' 테이블에서 생성 된 연관된 SFC 및 분류 자에 대한 참조를 갖는다.

또한 VNFD template은 VNFM에서 아래 코드와 같이, TOSCA 속성을 추가해야 한다. 아래의 속성은 VNFFG가 SFC provider에게 VNF가 Network Service Header (NSH) 를 지원하는지 그리고 패킷을 전송할 때, 어떤 encapsulation을 사용하는지에 대해 나타낼 수 있다. 여기서, NSH는 홉 단위로 SFC 홉에 대한 정보를 전달하는 IETF 프로토콜이다. 즉, NSH 헤더를 이용하여 체인을 통과하는 각 패킷에 추가되고 해당 체인에 대한 속성을 보유하여 패킷이 체인의 다음 VNF에 도착하면 VNF는 패킷이 속한 체인을 결정할 수 있고 패킷이 이전에 몇 개의 노드들을 통과하였는지에 대해 알 수 있다.

```
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]
```

그림 11. VNFM NSH TOSCA template

다음으로, REST API의 변화 부분은 다음과 같다. VNFFG를 생성하기 위해서는 VNFFGD가 생성되어야 한다. VNFFGD를 생성하는 방법은 TOSCA template scheme을 따른다. 포맷은 그룹에 정의되어 있는 하나 혹은 그 이상의 VNFFG가 요구된다.

VNFFG의 생성의 기본적인 방법은 미리 생성된 VNFFGD를 인스턴스화하여 수행된다. VNFFG 생성의 기본 동작은 추상화된 VNF type을 선택하는 것이다. VNFFGD에는 경로에 사용 된 forwarder 리스트를 포함하는 하나 이상의 Network Forwarding Path (NFP) 가 존재한다. VNFFG 생성시 NFVO 플러그인은 VNFM을 쿼리하여 각 NFP에 해당하는 VNFD에서 존재하는 사용 가능한 VNF 인스턴스를 찾는다. 현재는 VNFD에 둘 이상의 같은 type의 VNF가 존재할 경우 어떤 VNF를 사용할 지는 랜덤하게 선택된다. VNF는 다중 경로로는 선택 될 수 있지만 여러 VNFFG에 중복되게는 선택되지 못한다. 그리고 '--vnf-mapping'를 이용하여 그래프에서 사용할 VNFM을 통해 이미 생성 된 VNF 인스턴스를 지정하는 기능을 수행할 수 있다. 이것은 <VNFD>:<VNF Instance ID/NAME>로 맵핑이 된다. 예를 들어, 아래 VNFFGD TOSCA template 예제 코드를 보면 "Forwarding_path1"을 사용하는 경우 VNF1 및 VNF3 VNFD가 포함된다. 따라서 해당 VNFD, VNF1Test 및 VNF3Test에서 인스턴스가 두 개 이상 생성 된 경우 해당 VNF 인스턴스를 구체적으로 사용하도록 NFVO에 알리기 위해 '--vnf-mapping VNF1 : VNF1Test, VNF3 : VNF3Test' 로 맵핑이 된다.

가상 포트에 맵핑되어야 하는 요구 사항으로 CP는 특정 forwarder에 대해 VNFD에 정의되어야한다. CP는 체인의 일부로 '포워딩'기능을 갖는 것으로 정의되어야 한다. VNFD의 논리적 CP는 VNF 인스턴스의 neutron 포트로 맵핑되고, VNFFG는

주어진 CP에 대한 neutron port id를 얻기 위해, VNFM에게 query한다. 그리고 VNFM은 Heeat 드라이버를 호출하여 정보를 찾고, 이것은 VNFM에 새롭게 구현되어야 할 기능이다. 추가적으로, '--symmetrical'를 이용하여 VNFFG에서 대상으로 나열된 경로에 대해 역방향 경로를 자동으로 구성할 수 있다. 다른 방법으로, VNFFGD에서도 정의 될 수 있지만, 편의상 '--symmetrical'을 통해 구성한다.

```
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]
```

그림 12. VNFFGD TOSCA template exmaple

VNFFG의 사용법은 다음과 같다. 먼저 Tacker를 이용하여 VNFFG를 생성하기 위해서는 networking-sfc 프로젝트를 설치하여야 한다. OpenStack에서 VNFFG는 SFC

와 Classifier를 생성하기 위해 Neutron networking-sfc를 이용하기 때문이다. VNFFG의 생성법은 VNFD를 통해 VNF를 생성하는 것과 비슷하다. VNFFGD를 이용하여 VNFFG를 생성한다. 즉, VNFFGD를 생성한 후, Tacker 명령어를 이용하여 VNFFG를 생성한다. VNFFG에 대한 명령어는 다음과 같다.

VNFFGD를 생성하기 위한 명령어는 다음과 같다. VNFFGD를 생성하기 위해서는 VNFFGD에 리스트업 되어 있는 VNFD type에 대해 VNF 인스턴스가 생성되어 있어야 한다.

```
tacker vnffgd-create --vnffgd-file <vnffgd file> <vnffgd name>
```

VNFFGD의 예제코드는 아래 그림과 같다.

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Sample VNFFG template

topology_template:
  description: Sample VNFFG template

  node_templates:

    Forwarding_path1:
      type: tosca.nodes.nfv.FP.Tacker
      description: creates path (CP12->CP22)
      properties:
        id: 51
        policy:
          type: ACL
          criteria:
            - network_src_port_id: 640dfd77-c92b-45a3-b8fc-22712de480e1
            - destination_port_range: 80-1024
            - ip_proto: 6
            - ip_dst_prefix: 192.168.1.2/24
        path:
          - forwarder: VNFD1
            capability: CP12
          - forwarder: VNFD2
            capability: CP22
```

그림 13. VNFFGD 예제 코드 (1/2)

```
28    groups:
29      VNFFG1:
30        type: tosca.groups.nfv.VNFFG
31        description: HTTP to Corporate Net
32        properties:
33          vendor: tacker
34          version: 1.0
35          number_of_endpoints: 5
36          dependent_virtual_link: [VL12,VL22]
37          connection_point: [CP12,CP22]
38          constituent_vnfs: [VNFD1,VNFD2]
39        members: [Forwarding_path1]
```

그림 14. VNFFGD 예제 코드 (2/2)

VNFFG를 생성하는 명령어는 다음과 같다. 미리 인스턴스화 되어 있는 vnffgd를 이용하여 VNFFG를 생성하는 것이다.

```
tacker vnffg-create --vnffgd-name <vnffgd name> \
        --vnf-mapping <vnf mapping> --symmetrical <boolean>
```

```
tacker vnffg-create --vnffgd-name <vnffgd name> \
  --param-file <param file> --vnf-mapping <vnf mapping> \
  --symmetrical <boolean>
```

VNFFG를 나타내는 명령어는 다음과 같다. 이 명령어를 통해 나온 결과의 예제는 아래 그림과 같다.

```
tacker vnffg-show <vnffg name>
```

생성된 체인을 보는 명령어는 다음과 같다. 이를 통해 나타나는 특성들은 아래 그림과 같다.

```
tacker chain-show <chain id>
```

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

그림 15. VNFFG 결과 예제

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

그림 16. chain-show 결과 예제

Classifier 자체를 나타내는 명령어는 다음과 같다. 이를 통해 나타나는 특성들에 대한 예제는 아래 그림과 같다.

```
tacker classifier-show <classifier id>
```

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

그림 17. classifier-show 결과 예제

VNFFGD에 정의되어 있는 특성들은 아래 그림과 같다. 즉, id, name, VNFFGD에 대한 설명이 나타나 있는 description, VNFFGD template이 나타나 있는 attributes 그리고 VNFFGD를 실행할 프로젝트 id인 tenant_id로 구성되어 있다.

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 VNFFGD
attributes	dict	RW, All	None	template/ dict	VNFFGD template
tenant_id	string	RW, All	None (required)	string	project id to launch VNFFGD

그림 18. VNFFGD 특성

VNFFGD에 대한 REST call은 아래 그림과 같다. VNFFGD를 생성하고 삭제하는 create_vnffgd, delete_vnffgd와 특정 VNFFG id를 return하는 show_vnffgd 그리고 VNFFGD의 리스트를 나타내는 list_vnffgds가 있다.

REST Calls	Type	Expected Response	Body Data Schema	Description
create_vnffgd	post	200 OK	schema 1	Creates VNFFGD
delete_vnffgd	delete	200 OK	None	Deletes VNFFG by name or ID
show_vnffgd	get	200 OK	None	Returns output of specific VNFFG ID, including associated chains and classifiers
list_vnffgds	get	200 OK	None	Returns list of configured VNFFGD Names/IDs

그림 19. VNFFGD REST Call

3. OpenStack Tacker의 기타 기능

Tacker 프로젝트에서는 2 장에서 기술한 VNFFG 구성 기능과 함께 다중 사이트의 VIM을 관리하는 Multi-site VIM, VNF를 관리하기 위한 Monitoring Framework 그리고 VNF 배치를 위한 Enhanced Placement Awareness 등의 기능을 제공한다. 본 장에서는 Multi-site VIM, Monitoring Framework, Enhanced Placement Awareness 기능을 분석한다.

3.1. Multi-site VIM

Liberty 버전까지는 아래 그림과 같이 Tacker 서버에서는 하나의 VIM만 관리 가능하였으며, 해당 VIM을 통해서 VNF를 배포 할 수 있었다. 하지만, Mitaka 버전부터는 Multisite VIM 기능이 추가 되면서, 여러 사이트에 존재하는 VIM을 Tacker 서버에서 동시에 제어 및 관리가 가능하게 되었다.

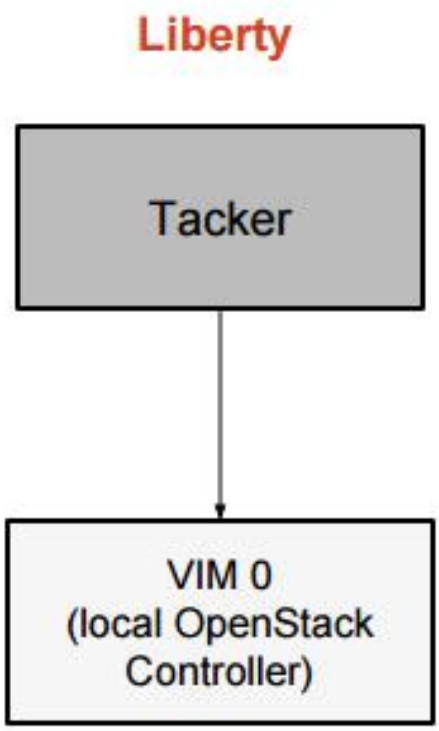

그림 20. Single site vim

Multi-site VIM은 NFV Orchestrator의 기능으로 추가 되었으며, 해당 기능을 통해 아래 그림과 같이 하나의 Tacker 서버가 여러 사이트 내에 존재하는 VIM을 제어 / 관리 한다. 예를 들어, Tacker 서버가 Site 1, Site 2에 VNF1과 VNF2를 각각 배포

하려고 할 때, Multi-site VIM 기능을 통해 각 사이트로 VNF들을 배포한다. 또한, 각 사이트들의 Openstack의 버전과 무관하게 Tacker 서버에서 통합적으로 관리 가능하다.

본 내용에서는 Multi-site VIM의 스팩 문서를 통해 구조를 알아보고, 해당 기능의 사용법에 대해 분석한다. 스팩 문서와 사용법은 [MV-S]과 [MV-U]를 참고 하였다.

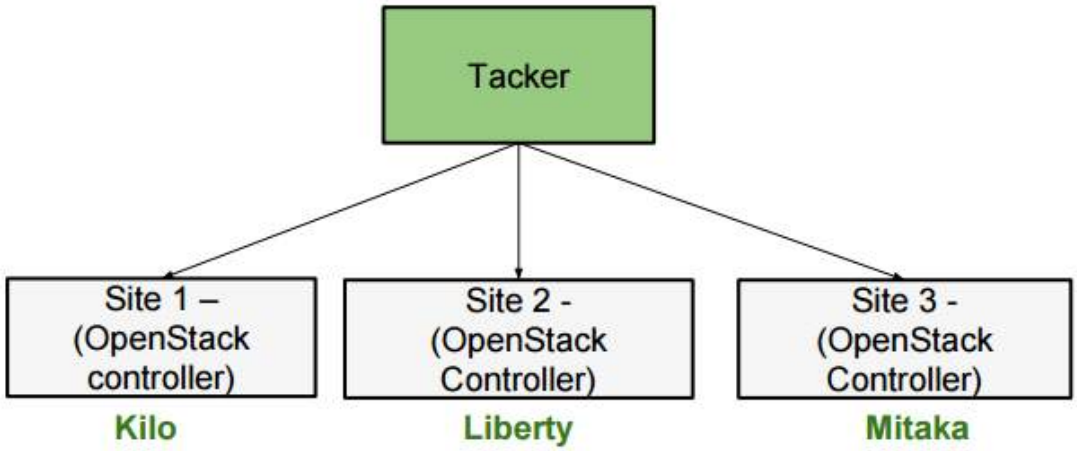

그림 21. Multi site VIM

3.1.1. Multisite VIM supprot in Tacker [6]

고객들에게 보다 빠른 네트워크 서비스를 제공하기 위해 통신사들은 VNF을 고객들과 가까이 배치하려고 한다. 즉, VNF들은 보다 빠른 네트워크 서비스 제공을 위해 여러 사이트에 배치된다. 이와 같은 상황에서, 하나의 VNF manager가 한 사이트내의 VNF들만 관리 가능하다며, 사이트의 개수만큼의 VNF manager가 필요한 단점이 존재한다. 반면에 하나의 VNF manager를 통해 통합 관리하기 위해서 각 사이트의 클라우드 시스템의 변경이 필요하다며, 통합적 관리를 위해 클라우드 시스템들을 모두 변경해야하는 단점이 존재한다. 이에 따라, 여러 사이트에 존재하는 클라우드 시스템의 수정 없이 배치된 VNF들을 통합 관리 및 제어하는 기능이 통신사들에게는 필수적이며, VNF 배치를 위한 최적의 VIM 선택 시, 아래와 같은 고려사항이 존재한다.

- NFV Orchestrator는 VFN 배치 시, 사용자에게 Service Level Agreement (SLA)을 보장하는 VIM 중에서 최적의 VIM을 선택해야 한다.
- Heartbeat를 통해 VIM들의 생존 여부와 성능을 모니터링 기능이 필요하다.
- 지리적 중복성 (geo-redundancy)과 같은 고객의 요구상황에 맞는 정책기반의 VIM 선택이 제공되어야 한다.

- 최적의 VNF 배치를 위해 VIM들의 자원사용량 정보를 알아야한다.

이러한 요구사항을 충족하기 위해 Multisite VIM 기능이 제안되었으며, 해당 기능을 구현하기 위해 변경된 사항은 아래와 같다.

- API

확장된 NFVO에서는 Tacker API v1에 추가되었다. 해당 API는 NFVO 인터페이스 계층을 정의하고, VIM의 REST API들을 표현한다. NFVO 인터페이스 계층에는 nfvo-plugin이 추가 되었으며, 해당 plugin은 Openstack의 디폴트 VIM 드라이버로 지원하다. 또한, 새로운 CRUD 동작을 통해 새로운 VIM을 추가하고, VIM의 유효성과 정보 등을 데이터베이스에 저장되도록 변경되었다. 아래 그림과 같이 NFVO를 통해 여러 사이트 VIM에 접근하고, VNFM을 통해 VNF을 배포 및 관리한다.

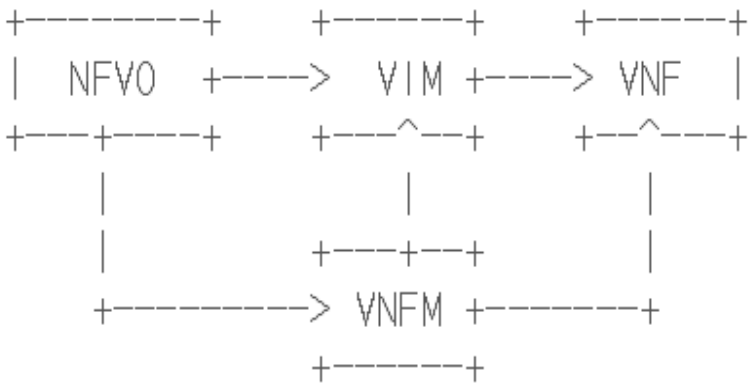

그림 22. API change

- Identity

Tacker 서버는 사용자가 제공한 인증 정보를 기반으로 관리 할 VIM들의 keystone 클라이언트를 동적으로 구성한다. Tacker 서버는 인증 정보를 통해 관리할 VIM에 대해 유효성이 판단한다. 판단 결과가 유효할 경우, VIM 등록과정동안 VIM의 인증정보와 가용영역이 Tacker 데이터베이스에 저장된다. VNF 배포 작업(vnf-create)시에는 먼저 배포 될 VIM의 인증정보를 검색하고, 해당 VIM에 VNF가 배포된다.

VIM의 인증 정보들은 암호화되고 'vimauth' 테이블에 저장된다. 개별 VIM의 암호된 정보를 복호화하기 위해서는 fernet 키가 생성되어야하며, 해당 키는 관리자만 접근 할 수 있는 루트 파일 시스템 경로에 저장된다. 해당 키는 VNF 배포 작업(vnf-create)시에 암호화된 VIM의 정보를 복호화 할 때 사용된다.

- VIM id in VNF Creation

사용자가 VNF 배포를 Tacker 서버에 요청 할 때, 사용자가 특정 VIM ID를 제

공해줌으로써, Tacker 서버는 특정 VIM내에 VNF를 배포 할 수 있다. 반면에, 사용자가 VIM ID를 제공해주지 않으면, Tacker 서버는 기본 VIM (default VIM)에 VNF를 배포한다. VIM ID 뿐만 아니라, 특정 사이트 이름을 통해 해당 사이트에 VNF를 배포 할 수 있으며, 특정 사이트를 나타내는 가용성 영역은 (Availability zone)은 VNFD의 placement_policy 속성에 정의가능하다.

- python-tackerclient and horizon dashboard

tacker vim-register, vim-show, vim-list, vim-update, vim-delete 기능과 같은 클라이언트 변경사항들이 VIM CRUD 기능에 포함된다. 일반적으로, 새로운 VIM 등록 명령은 아래와 같다.

```
tacker vim-register --name VIM3 --config-file ~/vim3.yaml
```

그림 23. VIM 등록 명령어

--config-file의 내용은 아래 그림과 같이 구성되며, 등록 할 VIM에 대한 정보들이 기입된다.

```
auth_url: http://10.10.10.13:5000
username: nfv_user
password: tacker_pw
tenant_name: nfv
```

그림 24. VIM 등록을 위한 config 파일

VIM의 업데이트 명령은 아래와 같다.

```
tacker vim-update --vim-id VIM3 --config-file ~/vim_update.yaml
```

그림 25. VIM 업데이트 명령어

업데이트를 위한 —config-file의 내용 아래와 같으며, 업데이트 할 정보들이 기입된다.

```
username: new_user
password: 123456
user_domain_id: default
```

그림 26. VIM 업데이트 파일

해당 업데이트 통해 VIM3의 ID를 갖는 VIM에 new_user 이름의 사용자가 추가된다. 또한, VNF 배포 명령인 tacker vnf-create에 -vim-id 와 -region-name 옵션을 추가함으로써, 특정 VIM ID 혹은 사이트 이름을 갖는 VIM에 VNF 배포를

가능하도록 하였다. 뿐만 아니라, horizon dashboard에 NFVO 탭을 새로 만들어 Tacker 서버에서 관리하는 (등록된) VIM리스트를 dashboard를 통해 확인 할 수 있도록 하였다.

- Error handling
  - 새로운 VIM 등록이 성공하지 못 할 시에는 오류의 타입과 해당 오류의 원인이 출력되어 진다.
  - Tacker 서버에서 VNF 배포하는 동안 VIM ID가 유효하지 않을 시에는, 해당 오류 메시지가 클라이언트와 horizon dashboard에 출력된다.
  - 등록 된 VIM으로 접근이 불가 할 때, VIM의 상태가 error 상태로 변경되어 지고, 접근 불가 유무는 VIM의 keystone과 heat 서비스를 통해 이루어진다.

Multi-site VIM기능을 제공하기 위해 가정된 사항은 다음과 같다. 각 사이트에 존재하는 Openstack들은 고유한 L2 하위 도메인 (개별 공급 네트워크)을 가지고 있거나, 혹은 중복되는 주소가 없는 하나의 큰 L2 도메인 (공유 공급 네트워크)를 가지고 있다고 가정한다. 또한, 각 VIM들은 OpenStack의 핵심 서비스인 nova, neutron, cinder, glance, horizon, idntity 그리고 heat 서비스를 제공한다고 가정하며, 이 기능들은 버전에 관계 VIM 등록을 허용되지만 Kilo 버전이후부터 지원 가능하다.

아래 나열된 멀티 사이트 배포와 관련된 기능은 추후 제공되어 질 것이다.

- 자체 테넌트내에서 사용자를 위한 역할 기반의 VNF 생성기능
- VIM들을 통한 자원 활용 및 관리
- Multi-site 환경에서 SFC 지원
- VMware, Xen, KVM과 같이 Openstack 기반이 아닌 클라우드 환경 지원

새로운 NFVO 확장 기능이 Tacker API v1에 도입되면서 VIM리소스에 대한 내용은 아래와 같다.

/vim

Attribute Name	Type	Access	Default Value	Validation/ Con^ersion	Description
id	string (UUID)	RO, All	generated	N/A	identity
name	string	RW, All	''	string	human+readable name
description	string	RW, All	''	string	description of template
auth_url	string	RW, All	''	string	identity service endpoint
auth_attr	string	RW, All	None	string	tenant_name, user-name, password, etc
placement_attr	dict	RO, All	None	dict	VIM region and availability zones
tenant_id	string	RO, All	N/A	string	project id of VIM
type	string	RW, All	openstack	string	driver implementing VIM specific logic

그림 27. Tacker API v1에서 VIM 정보

3.1.2. Multisite VIM Usage [7]

Multisite VIM을 사용하기 전에 아래와 같다.

- 원격 OpenStack 사이트에 새로운 'nfv' 프로젝트와 관리자 권한의 'nfv' 사용자를 생성한다.
- 관리 및 패킷 인/아웃을 위한 neutron netowrks를 생성한다.

Tacker를 통한 새로운 OpenStack VIM 등록은 아래와 같다

```
$ tacker vim-register --description 'OpenStack Liberty' --config-file vim_config.yaml Site1
Created a new vim:
+----------------+---------------------------------------------------------------------------------------------------------+
| Field          | Value                                                                                                   |
+----------------+---------------------------------------------------------------------------------------------------------+
| auth_cred      | {"username": "nfv_user", "password": "***", "project_name": "nfv", "user_id": "", "user_domain_name": "default", "auth_url": |
|                | "http://10.18.161.165:5000/v3", "project_id": "", "project_domain_name": "default"}                  |
| auth_url       | http://10.18.161.165:5000/v3                                                                            |
| description    | OpenStack Liberty                                                                                       |
| id             | 3f3c51c5-8bda-4bd3-adb3-5ae62eae65c3                                                                    |
| name           | Site1                                                                                                   |
| placement_attr | {"regions": ["RegionOne", "RegionTwo"]}                                                                 |
| tenant_id      | 8907bae480c0414d98c3519acbad1b06                                                                        |
| type           | openstack                                                                                               |
| vim_project    | {"id": "", "name": "nfv"}                                                                               |
+----------------+---------------------------------------------------------------------------------------------------------+
```

그림 28. 새로운 OpenStack VIM 등록

vim_config.yaml 파일은 VIM의 정보를 담고 있으며 아래와 같다.

```
auth_url: 'http://localhost:5000'
username: 'nfv_user'
password: 'devstack'
project_name: 'nfv'
```

그림 29. vim_config.yaml

auth_url은 원격 OpenStack 사이트의 keystone 서비스 인증 URL을 지칭한다.

다수의 VIM을 Tacker 서버에 등록 시, 반드시 기본 VIM (Default VIM)은 반드시 등록해야한다. 기본 VIM은 vnf-create 중에 선택 변수인 –vim-id가 없는 경우에 사용되며, 기본 vim을 등록하려면 manual install에 설명된 단계를 참조한다.

아래의 그림은 등록된 VIM위에 VNF를 생성하는 예제를 나타낸다.

```
$ tacker vnf-create --description 'Openwrt VNF on Site1' --vnfd-id c3cbf0c0-a492-49e3-9541-945e49e7ed7e --vim-name Site1 openwrt_VNF
Created a new vnf:
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| description    | Openwrt tosca template               |
| id             | 159ed8a5-a5a7-4f7a-be50-0f5f86603e3a |
| instance_id    | 7b4ab046-d977-4781-9f0c-1ee9dcce01c6 |
| mgmt_url       |                                      |
| name           | openwrt_VNF                          |
| placement_attr | {"vim_name": "Site1"}                |
| status         | PENDING_CREATE                       |
| tenant_id      | 8907bae480c0414d98c3519acbad1b06     |
| vim_id         | 3f3c51c5-8bda-4bd3-adb3-5ae62eae65c3 |
| vnfd_id        | c3cbf0c0-a492-49e3-9541-945e49e7ed7e |
+----------------+--------------------------------------+
```

그림 30. 특정 VIM에 VNF 생성

-vim-id / -vim-name 인수는 vnf-create 명령의 선택 사항이다. 기본적으로 -vim-id / -vim-name을 지정하지 않으면 기본 vim에 VNF가 생성이 된다. 기본 vim의 경우 vim-register 명령으로 vim 등록시 –is-default 옵션을 통해 기본 vim으로 tacker에 등록 가능하다. -vim-id / -vim-name 인수에 특정 vim의 id나 name을 지칭하면 해당 vim에 VNF가 생성된다.

Tacker 서버에서는 등록 된 'username', 'password', 'project_name' 및 'ids'와 같은 VIM의 인증 정보를 업데이트 할 수 있다. 아래의 명령을 통해 Tacker내의 등록

된 VIM의 인증 정보를 업데이트 한다.

```
$tacker vim-update VIM0 --config-file update.yaml
```

그림 31 . VIM 정보 업데이트 명령어

update.yaml 파일은 아래와 같다.

```
username: 'new_user'
password: 'new_pw'
```

그림 32. update.yaml 파일

Tacker 서버 등록된 VIM0라는 이름의 VIM의 username과 password를 update.yaml에 정의 된 것처럼 사용자이름과 비밀번호가 new_user와 new_pw로 업데이트 된다.

3.2. Tacker Monitoring Framework

VNF의 상태를 모니터링 하는 기능은 VNF를 관리하는데 있어 필수적인 기능이다. Tacker에서는 Monitoring Framework를 통해 VNF의 상태를 모니터링 하고, 이를 기반으로 VNF를 관리한다.

3.2.1. Monitoring Framework for VNF Manager [8]

Monitoring Framework for VNF Manager는 liberty버전에 제안된 기능이다. 제안된 기능을 통해 Tacker 서버에서는 VNF 관리를 위한 모니터링 기능을 제공한다. 또한, liberty이전에 제공되던 ping을 통해 단순히 VNF 인스턴스의 생존 유무확인 뿐만 아니라 더 복잡한 모니터링 기능 추가확정을 목표로 하고 있다.

Monitoring Framework을 구현하기 위해 변경사항은 다음과 같다. 기존의 관리 및 인프라 드라이버 (Management and Infrastructure Driver)와 유사한 드라이버 모델을 통해, 간단한 모니터링과 외부 모니터링 기능 수행을 통해 모니터링 기능이 향상된다. 따라서, 기존의 관리 드라이버를 복제하여 모니터링 기능을 모듈화 함으로써, 모니터링 방법을 쉽게 추가 시킬 수 있다. 해당 방식을 구현하기 위해, tacker/vm/drivers내에 mon_driver을 생성하고, 기존의 ping 모니터링 기능을 새 모듈화된 드라이버로 이동시킨다. 비록 기존의 모니터링 프레임워크를 구조적 변경 없이 모니터링기능을 추가함으로써 확장시킬 수 있지만, 제안한 드라이버를 사용하면 OpenStack에 존재하는 Monasca나 Ceilometer와 같은 모니터링 프로젝트를 쉽게 추가 할 수 있다.

이러한 모니터링 기능은 VNF 별로 정의가능하다. Tacker 서버에서 VNF생성 할

때 참조하는, VNFD를 통해 정의 할 수 있다. 모니터링 기능이 정의된 TOSCA 양식은 아래와 같다.

```
vduN:
  monitoring_policy:
    <monitoring-driver-name>:
      monitoring_params:
        <param-name>: <param-value>
        ...
      actions:
        <event>: <action-name>
        ...
    ...
```

그림 33. 모니터링 TOSCA 양식

드라이버는 monirot_drivers 디렉토리 구조에 존재해야하며, 불러올 수 있는 클래스여야 한다. 또한, setup.cfg 파일이 존재해야하며, 해당 파일은 Tacker 서버가 초기화 될 시 읽어져야 한다. 뿐만 아니라, 모니터링 쓰레드는 기본값이 30초로 지정된 global boot_wait configured time이후에 VNF와 VDU를 모니터링 하며, 드라이버는 기본값이 10초로 설정된 check_intvl interval time 간격으로 호출되어 모니터링을 수행한다. 사용되어지는 boot_wait 지연시간과 check_intvl 주기는 향후에 template으로 설정 가능하게 변경 될 예정이다.

```
vdu1:
  monitoring_policy:
    ping:
      actions:
        failure: respawn

vdu2:
  monitoring_policy:
    http-ping:
      monitoring_params:
        port: 8080
        url: ping.cgi
      actions:
        failure: respawn

    acme_scaling_driver:
      monitoring_params:
        resource: cpu
        threshold: 10000
      actions:
        max_foo_reached: scale_up
        min_foo_reached: scale_down
```

그림 34. 모니터링 기능을 위한 TOSCA 예제

3.2.2. Alarm-based monitoring driver to Tacker [9]

기존의 존재하는 모니터링 모듈을 사용하여, Tacker 서버에서 모니터링 기능을 지원하기 위한 알람기반의 모니터링 드라이버가 구현되었다. 아래의 그림은 알람기반의 모니터링 구조를 나타낸다. 기존의 OpenStack에서 모니터링 기능을 제공하는 Ceilometer나 Monasca 혹은 Custom 모니터링 기능을 통해 Tacker 서버에서 VNF를 모니터링 한다.

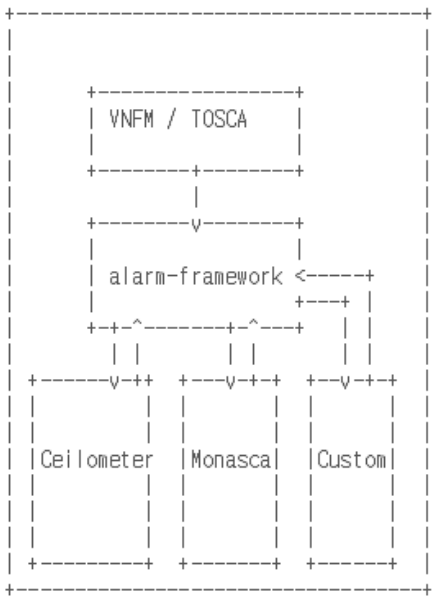

그림 35 알람기반 모니터링

3.2.3. Enhanced Placement Awareness [10]

OpenStack Tacker에서는 NUMA 토폴로지, SR-IOV, Huge pages, 그리고 CPU-Pining 등과 같이 컴퓨트 노드의 기능을 활용하는 VNF의 요구 사항을 지원하는 TOSCA VNFD template을 사용한다. VNFD template을 이용하여 성능이 높고 대기 시간이 짧은 VNF의 Enhanced Platform Awareness (EPA) 배치를 가능하게 한다. 컴퓨트 노드는 고성능 VNF 배포를 위한 EPA 배치를 지원하기 위해 BIOS, Hypervisor 및 OpenStack에서 구성해야한다. 아래 표는 BIOS, Hypervisor 및 OpenStack의 다양한 기능에 필요한 구성을 나타낸다.

	BIOS	Hypervisor	OpenStack
NUMA Topology	X		X
SR-IOV	X	X	X
HyperThreading	X		
Huge Pages		X	
CPU Pinning		X	X

그림 36. 필요구성 요소

참고 : BIOS에서 NUMA 토폴로지, SR-IOV 및 하이퍼 스레딩을 활성화하려면 서버 및 NIC 공급 업체의 구성 안내서를 참조. 또한 Hypervisor 토폴로지가 지원되는지 확인하려면 Hypervisor 설명서를 확인.

우분투 내의 /etc/default/grub에 snippet 파일은 아래와 같은 기능을 하게 한다.

- VM들이 사용하고 있는 커널 프로세스로부터 CPU isolation (isolcpus 참조)
- 대용량 메모리 페이지 예약 (default_hugepagesz, hugepagesz and hugepages 참조)
- SR-IOV 가상 기능 활성화 (intel_iommu 참조)

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=8-19
default_hugepagesz=1G hugepagesz=1G hugepages=24"
GRUB_CMDLINE_LINUX="intel_iommu=on"
```

중요 : 위의 내용은 서버의 하이퍼바이저 및 하드웨어 아키텍처에 따라 다를 수 있으니, Hypervisor 설명서를 참조.

아래의 테이블은 컴퓨트 노드에서 EPA 배치기능을 활성화하기 위해 필요한 OpenStack 파일들을 나타낸다.

	nova.conf	ml2_conf.ini	ml2_conf_sriov.ini
NUMA Topology	X		
SR-IOV	X	X	X
CPU Pinning	X		

그림 37 . EPA 배치기능을 위한 필요 Openstack 파일

컴퓨트 노드에서 NUMA 토폴로지 기능을 활성화하기 위해서는 nova.conf파일의 scheduler_default_filters에 NUMATopologyFilter를 추가해야한다. 반면에, SR-IOV 기능을 화성화하기 위해서는 컴퓨트 노드 뿐만 아니라 컨트롤 노드에서도 설정을 해야한다. 해당 SR-IOV를 설정하기 위한 과정은 다음 링크를 통해 확인 할 수 있다. http://docs.openstack.org/networking-guide/config-sriov.html

CPU pinning 기능은 컴퓨트 노드에서 nova.conf 구성을 통해 활성화된다. 아래의

항목과 같이 nova.conf를 구성한다.

```
[DEFAULT]
vcpu_pin_set = 8-19
cpu_allocation_ratio = 1.0
[libvirt]
virt_type = kvm
```

그림 38 . nova.conf

- 컴퓨트 노드를 통한 availability zone 생성

고성능 VNF 배포를 위해 컴퓨트 노드가 준비 된 후에 다음과정은 availability zone 생성이다. 해당 과정에서는 VNF 배포를 위해 식별 된 컴퓨트 노드에서 'aggregate-list'와 availability zone을 생성한다. 아래 명령은 aggregate-list와 availability zone 그리고 컴퓨트 노드 추가하는 예사를 보여준다.

```
openstack aggregate create --zone NFV-AZ NFV-AGG

openstack aggregate add host NFV-AGG <NFV_HOST1>

openstack aggregate add host NFV-AGG <NFV_HOST2>
```

그림 39. availability zone 생성

- Specifying Availability Zone for VDU in VNFD template

아래 VNFD 템프릿의 snippet을 나타낸다. 해당 snippet은 availability_zone 등록 정보를 사용하여 VDU 등록 정보의 일부로 작성된 EPA 가용 영역을 지정한다.

```
vdu1:
  type: tosca.nodes.nfv.VDU.Tacker
  capabilities:
    nfv_compute:
      properties:
        disk_size: 10 GB
        mem_size: 2048 MB
        num_cpus: 2
        mem_page_size: large
  properties:
    availability_zone: NFV-AZ
    image: cirros
```

그림 40. 특정 가용성 영역을 위한 VNFD template

4. OpenStack Tacker 기반의 VNFFG 구성 알고리즘 제안

4.1. 기존의 VNFFG 구성 알고리즘

앞서 기술한 바와 같이 Tacker의 VNFFG 기능은 트래픽이 지나야하는 VNF들을 정의하고, 해당 트래픽을 요구하는 VNF들을 지나도록 조율하고 관리하는 것을 목적으로 한다. 즉, Tacker 서버로부터 추상화된 VNFFG TOSCA에서 정의된 VNF목록 등의 VNFFG 관련된 정보를 서비스 기능 체인 (SFC) 및 트랙픽을 분류하는 Classifier에 제공해야한다. 정보를 받은 SFC는 트래픽에 요구하는 VNF들의 목록을 구성하고, Classifier은 다수의 트래픽을 분류하는 역할을 한다.

VNF가 VNFD로 정의되는 것과 같이 VNFFG는 VNFFG Descriptor로 정의된다. VNFFGD를 생성한 후에 VNFFG는 Tacker 서버에 의해 인스턴스화 된다.

VNFFG 생성 관련된 명령어는 다음과 같다.

- VNFFGD 생성

• tacker vnffgd-create --vnffgd-file <vnffgd file> <vnffgd name>

미리 작성된 vnffgd file을 참조하여 vnffgd name의 이름으로 VNFFGD를 온보드하는 명령어이다.

- VNFFG 생성

• tacker vnffg-create --vnffgd-name <vnffgd name> --vnf-mapping <vnf mapping> --symmetrical <boolean> <vnffg name>

온보딩된 VNFFGD를 참조하여 vnffg name의 이름으로 VNFFG를 생성하는 명령어이다. --vnf-mapping 옵션을 통해 특정 VNF를 지칭하여 VNFFG를 생성 가능하며, 해당 옵션을 사용하지 않으면 임의의 VNF를 선택하여 VNFFG를 생성한다.

- SFC 관련 명령어
  - **tacker nfp-list**
    - networking forwarding path (nfp)의 리스트를 보여준다.
  - **tacker nfp-show <nfp id>**
    - nfp id를 갖는 특정 nfp의 속성을 보여준다.
  - **tacker chain-list**
    - 생성된 chain의 리스트를 보여준다.
  - **tacker chain-show <chain id>**

    - chain id를 갖는 특정 chain의 속성을 보여준다.
- **tacker classifier-list**
    - classifier의 리스트를 보여준다.
- **tacker classifier-show <classifier id>**
    - classifier id를 갖는 특정 classifier의 속성을 보여준다.

4.2. VNFFG 소스코드 분석

앞서 기술한 명령어들 어떻게 코드상으로 구현되었는지 확인하기 위해 Tacker 내의 VNFFG 생성 소스코드를 분석한다. 소스 코드는 VNFFG 생성 관련 함수들이 정의된 vnffg_db.py 파일을 분석한다. 해당 파일은 tacker/db/nfvo 폴더 내에 존재하며, 분석하는 vnffg_db.py 파일은 4월 1일 기준의 master 브랜치의 파일이다.

아래의 그림은 VnffgPluginDbMixin 클래스 중 일부를 나타낸다. 해당 클래스에서는 VNFFGD 와 VNFFG 관련 tacker CLI 명령어에 따른 호출 함수들을 정의하고 있다. VnffgPluginDbMixin 클래스의 퍼블릭 함수는 다음과 같다.

- VNFFG 관련 함수
    - def create_vnffg
    - def get_vnffg
    - def get_vnffgs
    - def update_vnffg
    - def delete_vnffg
- VNFFGD 관련 함수
    - def create_vnffgd
    - def get_vnffgd
    - def get_vnffgds
    - def delete_vnffgd
- SFC 관련 함수
    - def get_classifier
    - def get_classifiers
    - def get_nfp
    - def get_nfps
    - def get_sfc
    - def get_sfcs

본 소스 코드 분석에서는 VNFFG 생성시 호출되는 create_vnffg 함수를 중점적으로 분석한다. create_vnffg 함수 분석을 통해 새로운 VNFFG 알고리즘을 구상할 예정이다.

```
188 class VnffgPluginDbMixin(vnffg.VNFFGPluginBase, db_base.CommonDbMixin):
189
190     def __init__(self):
191         super(VnffgPluginDbMixin, self).__init__()
192
193     def create_vnffg(self, context, vnffg):
194         vnffg_dict = self._create_vnffg_pre(context, vnffg)
195         sfc_instance = str(uuid.uuid4())
196         fc_instance = str(uuid.uuid4())
197         self._create_vnffg_post(context, sfc_instance,
198                                 fc_instance, vnffg_dict)
199         self._create_vnffg_status(context, vnffg_dict)
200         return vnffg_dict
201
202     def get_vnffg(self, context, vnffg_id, fields=None):
203         vnffg_db = self._get_resource(context, Vnffg, vnffg_id)
204         return self._make_vnffg_dict(vnffg_db, fields)
205
206     def get_vnffgs(self, context, filters=None, fields=None):
207         return self._get_collection(context, Vnffg, self._make_vnffg_dict,
208                                     filters=filters, fields=fields)
209
210     def update_vnffg(self, context, vnffg_id, vnffg):
211         vnffg_dict = self._update_vnffg_pre(context, vnffg_id)
212         self._update_vnffg_post(context, vnffg_id, constants.ACTIVE, vnffg)
213         return vnffg_dict
214
215     def delete_vnffg(self, context, vnffg_id):
216         self._delete_vnffg_pre(context, vnffg_id)
217         self._delete_vnffg_post(context, vnffg_id, False)
218
```

그림 41. vnffg_db.py 소스코드 (1/3)

```
219     def create_vnffgd(self, context, vnffgd):
220         template = vnffgd['vnffgd']
221         LOG.debug(_('template %s'), template)
222         tenant_id = self._get_tenant_id_for_create(context, template)
223 
224         with context.session.begin(subtransactions=True):
225             template_id = str(uuid.uuid4())
226             template_db = VnffgTemplate(
227                 id=template_id,
228                 tenant_id=tenant_id,
229                 name=template.get('name'),
230                 description=template.get('description'),
231                 template=template.get('template'))
232             context.session.add(template_db)
233 
234         LOG.debug(_('template_db %(template_db)s'),
235                   {'template_db': template_db})
236         return self._make_template_dict(template_db)
237 
238     def get_vnffgd(self, context, vnffgd_id, fields=None):
239         template_db = self._get_resource(context, VnffgTemplate,
240                                          vnffgd_id)
241         return self._make_template_dict(template_db, fields)
242 
243     def get_vnffgds(self, context, filters=None, fields=None):
244         return self._get_collection(context, VnffgTemplate,
245                                     self._make_template_dict,
246                                     filters=filters, fields=fields)
247 
248     def delete_vnffgd(self, context, vnffgd_id):
249         with context.session.begin(subtransactions=True):
250             vnffg_db = context.session.query(Vnffg).filter_by(
251                 vnffgd_id=vnffgd_id).first()
252             if vnffg_db is not None:
253                 raise nfvo.VnffgdInUse(vnffgd_id=vnffgd_id)
254 
255             template_db = self._get_resource(context, VnffgTemplate,
256                                              vnffgd_id)
257             context.session.delete(template_db)
```

그림 42 . vnffg_db.py 소스코드 (2/3)

```
259         def get_classifier(self, context, classifier_id, fields=None):
260             classifier_db = self._get_resource(context, VnffgClassifier,
261                                                classifier_id)
262             return self._make_classifier_dict(classifier_db, fields)
263
264         def get_classifiers(self, context, filters=None, fields=None):
265             return self._get_collection(context, VnffgClassifier,
266                                         self._make_classifier_dict,
267                                         filters=filters, fields=fields)
268
269         def get_nfp(self, context, nfp_id, fields=None):
270             nfp_db = self._get_resource(context, VnffgNfp, nfp_id)
271             return self._make_nfp_dict(nfp_db, fields)
272
273         def get_nfps(self, context, filters=None, fields=None):
274             return self._get_collection(context, VnffgNfp,
275                                         self._make_nfp_dict,
276                                         filters=filters, fields=fields)
277
278         def get_sfc(self, context, sfc_id, fields=None):
279             chain_db = self._get_resource(context, VnffgChain, sfc_id)
280             return self._make_chain_dict(chain_db, fields)
281
282         def get_sfcs(self, context, filters=None, fields=None):
283             return self._get_collection(context, VnffgChain,
284                                         self._make_chain_dict,
285                                         filters=filters, fields=fields)
286
```

그림 43. vnffg_db.py 소스코드 (3/3)

4.2.1. create_vnffg 함수 분석

create_vnffg 함수는 위의 소스코드에서 193줄부터 200줄까지로 정의된다. 해당 함수는 3가지 내부함수들을 (_create_vnffg_pre, _create_vnffg_post, _create_vnffg_status) 호출한다. 이 중 _create_vnffg_pre 함수는 가장 먼저 호출되는 함수로써, 참조하는 VNFFGD를 기반으로 VNFFG를 구성하고 구성된 VNFFG를 출력하는 함수이다. 해당 함수가 VNFFG를 구성하는데 핵심 함수이다. 아래의 소스코드는 _create_vnffg_pre 함수를 나타내고, 해당 함수는 vnffg_db.py 파일에 정의되어 있다.

```
288     def _create_vnffg_pre(self, context, vnffg):
289         vnffg = vnffg['vnffg']
290         LOG.debug(_('vnffg %s'), vnffg)
291         tenant_id = self._get_tenant_id_for_create(context, vnffg)
292         name = vnffg.get('name')
293         vnffg_id = vnffg.get('id') or str(uuid.uuid4())
294         template_id = vnffg['vnffgd_id']
295         symmetrical = vnffg['symmetrical']
296 
297         with context.session.begin(subtransactions=True):
298             template_db = self._get_resource(context, VnffgTemplate,
299                                              template_id)
300             LOG.debug(_('vnffg template %s'), template_db)
301             vnf_members = self._get_vnffg_property(template_db,
302                                                    'constituent_vnfs')
303             LOG.debug(_('Constituent VNFs: %s'), vnf_members)
304             vnf_mapping = self._get_vnf_mapping(context, vnffg.get(
305                                                 'vnf_mapping'), vnf_members)
306             LOG.debug(_('VNF Mapping: %s'), vnf_mapping)
307             # create NFP dict
308             nfp_dict = self._create_nfp_pre(template_db)
309             vnffg_db = Vnffg(id=vnffg_id,
310                              tenant_id=tenant_id,
311                              name=name,
312                              description=template_db.description,
313                              vnf_mapping=vnf_mapping,
314                              vnffgd_id=template_id,
315                              attributes=template_db.get('template'),
316                              status=constants.PENDING_CREATE)
317             context.session.add(vnffg_db)
318 
```

그림 44 . _create_vnffg_pre함수 소스코드 (1/2)

```
            nfp_id = str(uuid.uuid4())
            sfc_id = str(uuid.uuid4())
            classifier_id = str(uuid.uuid4())

            nfp_db = VnffgNfp(id=nfp_id, vnffg_id=vnffg_id,
                              tenant_id=tenant_id,
                              name=nfp_dict['name'],
                              status=constants.PENDING_CREATE,
                              path_id=nfp_dict['path_id'],
                              symmetrical=symmetrical)
            context.session.add(nfp_db)

            chain = self._create_port_chain(context, vnf_mapping, template_db,
                                            nfp_dict['name'])
            LOG.debug(_('chain: %s'), chain)
            sfc_db = VnffgChain(id=sfc_id,
                                tenant_id=tenant_id,
                                status=constants.PENDING_CREATE,
                                symmetrical=symmetrical,
                                chain=chain,
                                nfp_id=nfp_id,
                                path_id=nfp_dict['path_id'])

            context.session.add(sfc_db)

            sfcc_db = VnffgClassifier(id=classifier_id,
                                      tenant_id=tenant_id,
                                      status=constants.PENDING_CREATE,
                                      nfp_id=nfp_id,
                                      chain_id=sfc_id)
            context.session.add(sfcc_db)

            match = self._policy_to_acl_criteria(context, template_db,
                                                 nfp_dict['name'],
                                                 vnf_mapping)
            LOG.debug(_('acl_match %s'), match)

            match_db_table = ACLMatchCriteria(
                id=str(uuid.uuid4()),
                vnffgc_id=classifier_id,
                **match)

            context.session.add(match_db_table)

        return self._make_vnffg_dict(vnffg_db)
```

그림 45. _create_vnffg_pre함수 소스코드 (2/2)

_create_vnffg_pre 함수는 context와 vnffg 데이터를 입력으로 받는다. context의 경우 user_id, tenant_id 등이 포함되며, vnffg 데이터에는 tacker vnffg-create --vnffgd-name <vnffgd name> --vnf-mapping <vnf mapping> --symmetrical <boolean> <vnffg name> 명령어에 따른 데이터가 포함된다. 아래 그림은 vnffg 데이터의 예를 나타낸다.

```
['vnffg': {'description': 'dummy_vnf_description',
           'vnffgd_id': u'eb094833-995e-49f0-a047-dfb56aaf7c4e',
           'tenant_id': u'ad7ebc56538745a08ef7c5e97f8bd437',
           'name': 'dummy_vnffg',
           'vnf_mapping': {},
           'symmetrical': False}}
```

그림 46. vnffg 데이터 예시

해당 예시에는 --vnf-mapping 옵션을 사용하지 않은 경우를 보여준다. --vnf-mapping 옵션을 사용하는 경우 'vnf_mappin': {} 필드에 작성이 된다. vnffg 데이터와 context 데이터를 통해 vnffg, tenant_id, name, vnffg_id, template_id, symmtrical 변수를 초기화 시킨다. (289줄 ~ 295줄)

초기화된 변수를 통해 VNFFG template을 호출하고, 호출한 VNFFG template에 정의된 VNF 체인을 확인 후에 해당 VNF 체인에 맞는 VNF 인스턴스를 매핑한다. (298줄 ~ 306줄) 관련 함수들은 다음과 같다.

- _get_resource 함수를 통해 template_id를 갖는 VNFFG template을 호출한다.
- _get_vnffg_property 함수를 통해 호출한 VNFFG template에서 'constituent_vnfs' 필드값을 불러온다. 'constituent_vnfs' 필드값은 생성할 VNFFG가 요구하는 VNF들의 순서를 의미하는 값이다. 아래의 코드는 _get_vnffg_property 함수를 나타낸다.

```
@staticmethod
def _get_vnffg_property(template_db, vnffg_property):
    template = template_db.template['vnffgd']['topology_template']
    vnffg_name = list(template['groups'].keys())[0]
    try:
        return template['groups'][vnffg_name]['properties'][vnffg_property]
    except KeyError:
        raise nfvo.VnffgPropertyNotFoundException(
            vnffg_property=vnffg_property)
```

그림 47. _get_vnffg_property 함수

해당 함수는 입력받은 vnffg_property를 template으로부터 찾아서 출력해주는 것을 코드를 통해 확인 할 수 있다. 즉, 아래의 VNFFGD를 참조하여 예를 들면, constituent_vnfs: [VNFD1, VNFD2]로 정의하였기 때문에, 해당

디스크립터를 참조하여 생성되는 VNFFG는 VNFD1을 참조하여 생성된 VNF 인스턴스를 먼저 지나고, 그 후에 VNFD2을 참조하여 생성된 VNF 인스턴스를 지나는 그래프가 생성되어야 한다. 즉, _get_vnffg_property 함수를 통해 [VNFD1, VNFD2] 데이터가 출력된다.

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Sample VNFFG template

topology_template:
  description: Sample VNFFG template

  node_templates:

    Forwarding_path1:
      type: tosca.nodes.nfv.FP.Tacker
      description: creates path (CP12->CP22)
      properties:
        id: 51
        policy:
          type: ACL
          criteria:
            - network_src_port_id: 640dfd77-c92b-45a3-b8fc-22712de480e1
            - destination_port_range: 80-1024
            - ip_proto: 6
            - ip_dst_prefix: 192.168.1.2/24
        path:
          - forwarder: VNFD1
            capability: CP12
          - forwarder: VNFD2
            capability: CP22

  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: [VL12,VL22]
        connection_point: [CP12,CP22]
        constituent_vnfs: [VNFD1,VNFD2]
      members: [Forwarding_path1]
```

그림 48. VNFFGD 예시

- _get_vnf_mapping 함수를 통해 생성될 VNFFG의 체인에 맞는 VNF 인스턴스를 선택한다. 해당 함수의 주요 입력값으로는 vnf_mapping 데이터와 vnf_members 데이터가 존재한다. vnf_mapping 데이터의 경우 앞서 기술한바와 같이 vnffg 데이터에 포함된 값이며, tacker CLI 명령어에서 —vnf-mapping 옵션을 통해 채워진다. 따라서, _create_vnffg_pre함수에서는 vnffg의 ‘vnf_mappig’ 필드의 데이터를 _get_vnf_mapping 함수의 vnf_mapping 값으로 입력하는 것을 확인 할 수 있다. 한편, vnf_members 입력값은 앞서 기술한 _get_vnffg_property 함수를 통해 출력한 데이터를 입력으로 한다.

```
475     def _get_vnf_mapping(self, context, vnf_mapping, vnf_members):
476         """Creates/validates a mapping of VNFD names to VNF IDs for NFP.
477 
478         :param context: SQL session context
479         :param vnf_mapping: dict of requested VNFD:VNF_ID mappings
480         :param vnf_members: list of constituent VNFs from a VNFFG
481         :return: dict of VNFD:VNF_ID mappings
482         """
483         vnfm_plugin = manager.TackerManager.get_service_plugins()['VNFM']
484         new_mapping = dict()
485 
486         for vnfd in vnf_members:
487             # there should only be one ID returned for a unique name
488             try:
489                 vnfd_id = vnfm_plugin.get_vnfds(context, {'name': [vnfd]},
490                                                 fields=['id']).pop()['id']
491             except Exception:
492                 raise nfvo.VnffgdVnfdNotFoundException(vnfd_name=vnfd)
493             if vnfd_id is None:
494                 raise nfvo.VnffgdVnfdNotFoundException(vnfd_name=vnfd)
495             else:
496                 # if no VNF mapping, we need to abstractly look for instances
497                 # that match VNFD
498                 if vnf_mapping is None or vnfd not in vnf_mapping.keys():
499                     # find suitable VNFs from vnfd_id
500                     LOG.debug(_('Searching VNFS with id %s'), vnfd_id)
501                     vnf_list = vnfm_plugin.get_vnfs(context,
502                                                     {'vnfd_id': [vnfd_id]},
503                                                     fields=['id'])
504                     if len(vnf_list) == 0:
505                         raise nfvo.VnffgInvalidMappingException(vnfd_name=vnfd)
506                     else:
507                         LOG.debug(_('Matching VNFs found %s'), vnf_list)
508                         vnf_list = [vnf['id'] for vnf in vnf_list]
509                     if len(vnf_list) > 1:
510                         new_mapping[vnfd] = random.choice(vnf_list)
511                     else:
512                         new_mapping[vnfd] = vnf_list[0]
513                 # if VNF mapping, validate instances exist and match the VNFD
514                 else:
515                     vnf_vnfd = vnfm_plugin.get_vnf(context, vnf_mapping[vnfd],
516                                                    fields=['vnfd_id'])
517                     if vnf_vnfd is not None:
518                         vnf_vnfd_id = vnf_vnfd['vnfd_id']
519                     else:
520                         raise nfvo.VnffgInvalidMappingException(vnfd_name=vnfd)
521                     if vnfd_id != vnf_vnfd_id:
522                         raise nfvo.VnffgInvalidMappingException(vnfd_name=vnfd)
523                     else:
524                         new_mapping[vnfd] = vnf_mapping.pop(vnfd)
525         self._validate_vim(context, new_mapping.values())
526         return new_mapping
```

그림 49. _get_vnf_mapping 함수

- _get_vnf_mapping 함수의 경우 VNFD와 VNF를 관리하는 VNF Manager 모듈을 호출한다. (483줄) VNF Manager를 호출하는 이유는 정의한 VNFD가 온보드 되었는지, 해당 VNFD를 참조하여 생성된 VNF들이 존재하는지에 대한 정보가 필요하기 때문이다. 따라서, _get_vnf_mapping 함수는 vnf_members에 정의된 VNFD들이 존재하는지 확인을 한다. (489줄~494줄) 정의된 VNFD가 VNF Manager에서 관리 중이라고 하면, VNFD를 참조하여 생성된 VNF들을 호출한다. 이때, vnf_mapping 입력값이 존재한다면 (즉, 특정 VNF들을 CLI명령어를 통해 지칭하였다면), vnf_mapping에 정의된 VNF의 VNFD_ID를 VNF Manager로부터 확인하다. (515줄~520줄) 확인한 VNFD_ID 와 vnf_members에 정의된 ID와 비교하여, 같으면 해당 VNF 인스턴스를 선택한다.
- 이와 반대로, vnf_mapping 입력값이 존재하지 않는다면 (즉, 특정 VNF들을 CLI명령어를 통해 지칭하지 않았다면), vnf_members에 정의된 VNFD를 참조한 VNF 인스턴스를 선택해야한다. 우선 vnf_members에 정의된 VNFD를 참조하여 생성된 VNF 인스턴스 리스트를 VNF Manager를 통해 확인하다. (501줄) 이때, VNF 인스턴스가 한 개만 존재하는 경우 해당 인스턴스를 선택하고 (512줄), 한 개이상의 VNF 인스터스가 존재하는 경우 리스트내의 VNF 인스턴스 중 하나를 임의로 선택한다.

해당 과정을 통해 생성될 VNFFG가 지나야하는 VNF 인스턴스들이 구성되고, VNFFG 인스턴스가 생성되고, 업데이트 된다. (309줄~316줄) VNFFG 뿐만아니라, NFP, SFC, SFC Classifier 인스턴스가 생성 및 업데이트 된다.

Tacker VNFFG 기능의 경우 VNF 인스턴스간의 체이닝은 OpenStack내의 SFC 프로젝트인 networking-sfc만 지원하므로, networking-sfc에서 체이닝을 하는 방식인 포트간의 연결성을 통해 VNF간의 그래프가 구성된다. chain = self._create_port_chain(context, vnf_mapping, template_db, nfp_dict['name']) (331줄) 함수를 통해 체이닝을 위한 물리적 포트 ID리스트를 생성한다.

5. OpenStack Tacker 기반의 VNFFG 구성 테스트

앞서 언급하였듯이, OpenStack Tacker는 ETSI NFV 구조에서의 VNF Manager 및 NFVO 기능을 제공한다. 본 장에서는 OpenStack Tacker의 VNF Manager 기능 및 NFVO 기능 (즉, VNFFG Manager)을 검증하기 위한 테스트를 수행한다.

5.1. VNF 구성 테스트

OpenStack Tacker의 VNF Manager 기능을 검증하기 위해, VNF Descriptor를 통해 VNF 인스턴스를 생성하는 테스트를 수행한다.

5.1.1. 테스트 환경

5.1.1.1. Virtual Machine 환경

테스트는 VirtualBox를 통해 생성한 하나의 가상 머신 위에서 수행한다. 가상 머신은 다음의 환경으로 구축하였다.

- Operating System: Ubuntu 16.04
- CPU: 2 vCPU
- RAM: 6GB
- Storage: 32GB

5.1.1.2. Tacker devstack 설치 방법

생성한 가상 머신 위에 Tacker를 설치하기 위해 Tacker에서 제공하는 통합 설치 툴인 devstack을 활용한다. Tacker devstack은 bash shell script를 통해 Tacker를 포함하여 Tacker를 실행시키기 위해 필요한 모든 소프트웨어들을 자동으로 다운로드/설치한다. Tacker devstack의 다운로드/설치 방법은 다음과 같다.

- devstack (stable/newton버전) 다운로드:
$ git clone -b stable/newton https://git.openstack.org/openstack-dev/devstack
- local.conf 파일 다운로드 및 수정 다운로드: local.conf 예시 파일 (URL: https://github.com/openstack/tacker/blob/stable/newton/devstack/local.conf.example)을 devstack 디렉토리에 저장한다. 저장한 local.conf 파일에서 local.conf 파일에서 'HOST_IP'를 VM IP 주소로 변경한다. 또한, local.conf 파일에 'enable_plugin heat https://git.openstack.org/openstack/heat stable/newton'을 추가한다.
- devstack 설치 및 credential 설정
$ ./stack.sh
$ source openrc admin admin

5.1.1.3. VNF Descriptor

OpenStack Tacker에서는 Tosca 기반의 yaml template을 통해 VNF Descriptor를 제공한다. 본 테스트에서는 두 종류의 VNF Descriptor를 사용하였고 이를 각각 VNFD1 및 VNFD2라고 하면, 이는 다음과 같다.

```yaml
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Demo example

metadata:
  template_name: sample-tosca-vnfd1

topology_template:
  node_templates:
    VDU1:
      type: tosca.nodes.nfv.VDU.Tacker
      capabilities:
        nfv_compute:
          properties:
            num_cpus: 1
            mem_size: 512 MB
            disk_size: 1 GB
      properties:
        image: cirros-0.3.5-x86_64-disk
        availability_zone: nova
        mgmt_driver: noop
        config: |
          param0: key1
          param1: key2

    CP11:
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        management: true
        order: 0
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL11
        - virtualBinding:
            node: VDU1

```

그림 50 VNFD1의 tosca yaml template (1/2)

```
38      CP12:
39        type: tosca.nodes.nfv.CP.Tacker
40        properties:
41          order: 1
42          anti_spoofing_protection: false
43        requirements:
44          - virtualLink:
45              node: VL12
46          - virtualBinding:
47              node: VDU1
48
49      CP13:
50        type: tosca.nodes.nfv.CP.Tacker
51        properties:
52          order: 2
53          anti_spoofing_protection: false
54        requirements:
55          - virtualLink:
56              node: VL13
57          - virtualBinding:
58              node: VDU1
59
60      VL11:
61        type: tosca.nodes.nfv.VL
62        properties:
63          network_name: net_mgmt
64          vendor: Tacker
65
66      VL12:
67        type: tosca.nodes.nfv.VL
68        properties:
69          network_name: net0
70          vendor: Tacker
71
72      VL13:
73        type: tosca.nodes.nfv.VL
74        properties:
75          network_name: net1
76          vendor: Tacker
```

그림 51 VNFD1의 tosca yaml template (2/2)

```yaml
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Demo example

metadata:
  template_name: sample-tosca-vnfd1

topology_template:
  node_templates:
    VDU1:
      type: tosca.nodes.nfv.VDU.Tacker
      capabilities:
        nfv_compute:
          properties:
            num_cpus: 1
            mem_size: 512 MB
            disk_size: 1 GB
      properties:
        image: cirros-0.3.5-x86_64-disk
        availability_zone: nova
        mgmt_driver: noop
        config: |
          param0: key1
          param1: key2

    CP21:
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        management: true
        order: 0
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL21
        - virtualBinding:
            node: VDU1
```

그림 52 VNFD2의 tosca yaml template (1/2)

```
38      CP22:
39        type: tosca.nodes.nfv.CP.Tacker
40        properties:
41          order: 1
42          anti_spoofing_protection: false
43        requirements:
44          - virtualLink:
45              node: VL22
46          - virtualBinding:
47              node: VDU1
48
49      CP23:
50        type: tosca.nodes.nfv.CP.Tacker
51        properties:
52          order: 2
53          anti_spoofing_protection: false
54        requirements:
55          - virtualLink:
56              node: VL23
57          - virtualBinding:
58              node: VDU1
59
60      VL21:
61        type: tosca.nodes.nfv.VL
62        properties:
63          network_name: net_mgmt
64          vendor: Tacker
65
66      VL22:
67        type: tosca.nodes.nfv.VL
68        properties:
69          network_name: net0
70          vendor: Tacker
71
72      VL23:
73        type: tosca.nodes.nfv.VL
74        properties:
75          network_name: net1
76          vendor: Tacker
```

그림 53 VNFD2의 tosca yaml template (2/2)

5.1.2. 테스트 결과

테스트는 앞서 설명한 VNFD1 및 VNFD2를 온보딩 하는 과정, 그리고 각각의 VNFD를 통해 VNF 인스턴스를 생성하는 과정으로 구성된다.

5.1.2.1. VNFD 온보딩 테스트

먼저 VNFD 1에 대한 온보딩을 Tacker에서 제공하는 command line interface (CLI) (python-tackerclient)를 통해 다음과 같이 수행한다.

```
stack@stack-VirtualBox:~/devstack$ tacker vnfd-create --vnfd-file ./sample_descriptors/tosca-templates/vnffg
d/test_vnffg-vnfd1.yaml VNFD1
Passing infra_driver and mgmt_driver in the VNFD API is deprecated. infra_driver will be automatically deriv
ed from target vim type. mgmt_driver will be derived from TOSCA template values.
Created a new vnfd:
+--------------+--------------------------------------------------------------------------+
| Field        | Value                                                                    |
+--------------+--------------------------------------------------------------------------+
| created_at   | 2017-03-22 02:10:17.516247                                               |
| description  | Demo example                                                             |
| id           | 63e325ae-0f79-46ee-9910-12d7de463d0c                                     |
| infra_driver | heat                                                                     |
| mgmt_driver  | noop                                                                     |
| name         | VNFD1                                                                    |
| service_types | {"service_type": "vnfd", "id": "06f15129-5445-4679-ac8e-e2bf3573f2a2"} |
| tenant_id    | e8d0488104d5408aa5b1f93d40bf9a85                                         |
| updated_at   |                                                                          |
+--------------+--------------------------------------------------------------------------+
```

그림 54 VNFD1 on-boarding

다음으로 VNFD2에 대한 온보딩은 아래와 같이 수행한다.

```
stack@stack-VirtualBox:~/devstack$ tacker vnfd-create --vnfd-file ./sample_descriptors/tosca-templates/vnffg
d/test_vnffg-vnfd2.yaml VNFD2
Passing infra_driver and mgmt_driver in the VNFD API is deprecated. infra_driver will be automatically deriv
ed from target vim type. mgmt_driver will be derived from TOSCA template values.
Created a new vnfd:
+--------------+--------------------------------------------------------------------------+
| Field        | Value                                                                    |
+--------------+--------------------------------------------------------------------------+
| created_at   | 2017-03-22 02:11:24.137284                                               |
| description  | Demo example                                                             |
| id           | 393bf32b-c6f7-43e3-a335-3d00c9c88bcf                                     |
| infra_driver | heat                                                                     |
| mgmt_driver  | noop                                                                     |
| name         | VNFD2                                                                    |
| service_types | {"service_type": "vnfd", "id": "e4ec0bf0-757d-43f7-bbeb-fbf519138415"} |
| tenant_id    | e8d0488104d5408aa5b1f93d40bf9a85                                         |
| updated_at   |                                                                          |
+--------------+--------------------------------------------------------------------------+
```

그림 55 VNFD2 on-boarding

온보딩이 성공적으로 수행되었는지 확인하기 위해서 Tacker의 CLI 및 UI를 통해 온보딩 된 VNF Descriptor 리스트를 아래와 같이 확인한다. 그 결과, 앞서 온보딩을 수행한 VNFD1 및 VNFD2가 성공적으로 온보딩 되어있음을 확인할 수 있다.

```
stack@stack-VirtualBox:~/devstack$ tacker vnfd-list
+--------------------------------------+-------+-------------+-------------+-------------+
| id                                   | name  | description | infra_driver | mgmt_driver |
+--------------------------------------+-------+-------------+-------------+-------------+
| 03dde5c8-81d1-4d07-83d4-d14aa3efec25 | VNFD1 | Demo example | heat         | noop        |
| 3f63d901-5c60-484b-9990-d86a4f84cb7a | VNFD2 | Demo example | heat         | noop        |
+--------------------------------------+-------+-------------+-------------+-------------+
```

그림 56 CLI를 통한 온보딩 VNFD 리스트 확인

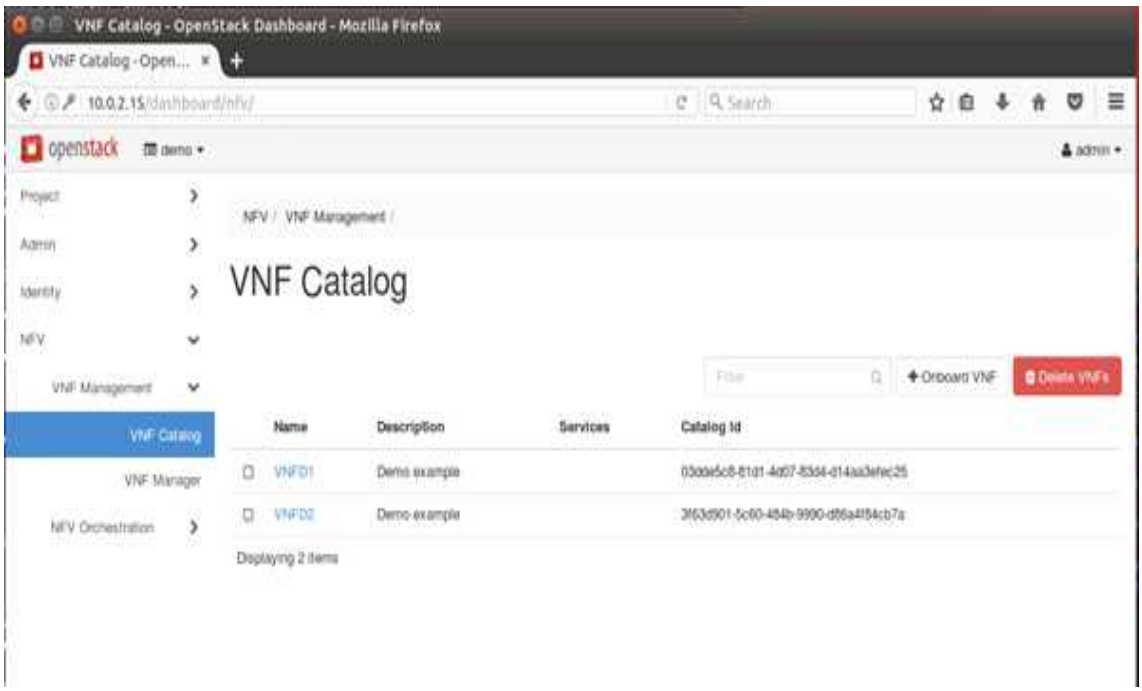

그림 57 Web UI를 통한 온보딩 VNFD 리스트 확인

5.1.2.2. VNF 인스턴스 생성 테스트

아래와 같이 Tacker CLI를 통해 앞서 온보딩을 수행한 VNFD1 및 VNFD2를 통해 실제 VNF 인스턴스를 생성한다. 이 때, CLI 옵션을 통해 VNFD1 및 VNFD2를 통해 생성한 인스턴스들의 이름을 각각 VNF1_VNFD1 그리고 VNF1_VNFD2가 되도록 설정한다.

```
stack@stack-VirtualBox:~/devstack$ tacker vnf-create --vnfd-name VNFD1 VNF1_VNFD1
Created a new vnf:
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| created_at     | 2017-03-22 02:11:44.452769           |
| description    | Demo example                         |
| error_reason   |                                      |
| id             | 6a609cab-066f-4629-96e7-aa2e0e5e314d |
| instance_id    | 1e88af4d-37e8-4f78-aca6-e70af4e1f35c |
| mgmt_url       |                                      |
| name           | VNF1_VNFD1                           |
| placement_attr | {"vim_name": "VIM0"}                 |
| status         | PENDING_CREATE                       |
| tenant_id      | e8d0488104d5408aa5b1f93d40bf9a85     |
| updated_at     |                                      |
| vim_id         | 26859e60-3879-4206-8143-9e7088997e5c |
| vnfd_id        | 63e325ae-0f79-46ee-9910-12d7de463d0c |
+----------------+--------------------------------------+
```

그림 58 VNFD1을 통한 VNF 인스턴스 생성

```
stack@stack-VirtualBox:~/devstack$ tacker vnf-create --vnfd-name VNFD2 VNF1_VNFD2
Created a new vnf:
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| created_at     | 2017-03-22 02:12:32.093666           |
| description    | Demo example                         |
| error_reason   |                                      |
| id             | 373300d1-282f-4c0b-a313-23803a411f5b |
| instance_id    | 09a754d3-e10b-44c7-9747-531ef102c4df |
| mgmt_url       |                                      |
| name           | VNF1_VNFD2                           |
| placement_attr | {"vim_name": "VIM0"}                 |
| status         | PENDING_CREATE                       |
| tenant_id      | e8d0488104d5408aa5b1f93d40bf9a85     |
| updated_at     |                                      |
| vim_id         | 26859e60-3879-4206-8143-9e7088997e5c |
| vnfd_id        | 393bf32b-c6f7-43e3-a335-3d00c9c88bcf |
+----------------+--------------------------------------+
stack@stack-VirtualBox:~/devstack$
```

그림 59 VNFD2를 통한 VNF 인스턴스 생성

VNF 인스턴스들이 성공적으로 생성되어 동작하고 있는지를 확인하기 위해 Tacker CLI 및 Web UI를 통해 아래와 같이 확인한 결과, VNF들의 상태가 Active 임을 확인할 수 있다.

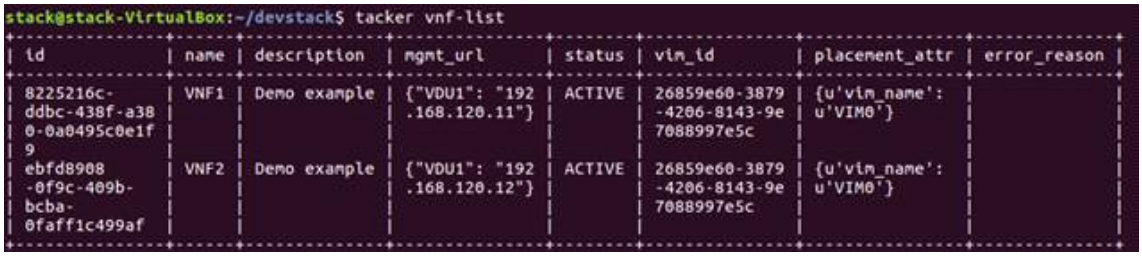

그림 60 CLI를 통한 VNF 인스턴스 리스트 확인

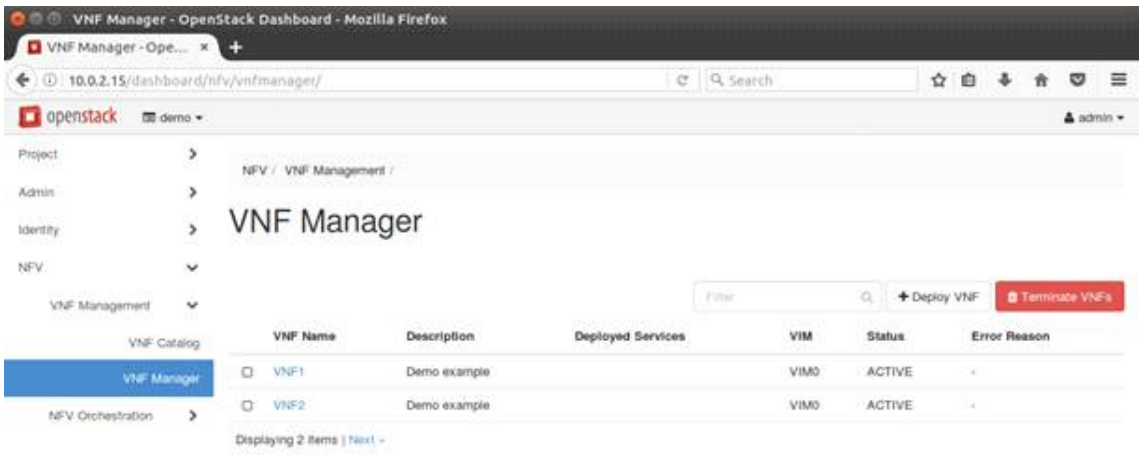

그림 61 Web UI를 통한 VNF 인스턴스 리스트 확인

5.2. VNFFG 구성 테스트

본 테스트에서는 OpenStack Tacker의 VNFFG Manager 기능을 검증하기 위해, VNFFG Descriptor를 통해 VNF 인스턴스를 생성하는 테스트를 수행한다.

5.2.1. 테스트 환경

VNFFG 구성 테스트 환경 (Virtual Machine 환경, devstack 등)은 앞의 풀 구성 테스트와 동일하므로 생략한다.

5.2.1.1. VNFFG Descriptor

VNFFG 구성 테스트에서는 포워딩 그래프에 포함될 VNF Descriptor들 및 이들의 연결관계를 참조하고 있는 VNFFG Descriptor가 사용된다. 테스트에 사용된 VNFFG Descriptor는 다음 그림과 같이 VNF 구성 테스트에서 사용한 VNFD1 및 VNFD2를 참조하고 있다.

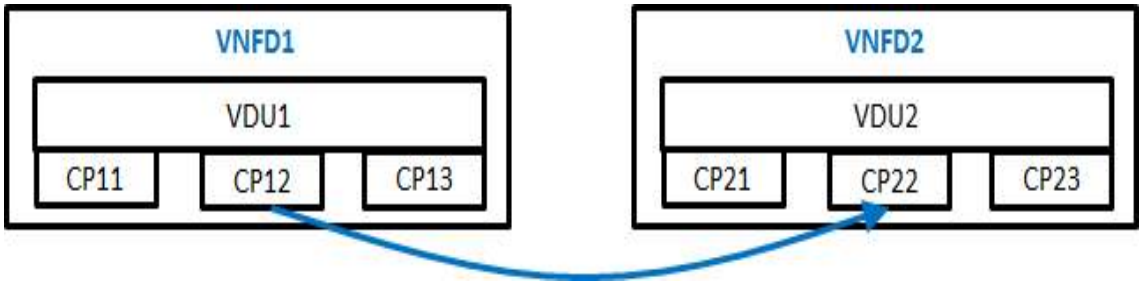

그림 62 테스트에 사용된 VNFFGD 도식화

다음은 테스트에서 사용된 VNFFGD의 tosca yaml template을 나타낸다. 이 때, netsor_src_port_id는 VNFD1의 CP12에 대한 port_id로 수정해 주어야 한다. 또한, policy type으로는 현재 acces control list (ACL) 만을 제공하며, criteria에 매칭이 되는 트래픽만 해당 체인을 부여받도록 한다.

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Sample VNFFG template

topology_template:
  description: Sample VNFFG template

  node_templates:

    Forwarding_path1:
      type: tosca.nodes.nfv.FP.Tacker
      description: creates path (CP12->CP22)
      properties:
        id: 51
        policy:
          type: ACL
          criteria:
            - network_src_port_id: 640dfd77-c92b-45a3-b8fc-22712de480e1
            - destination_port_range: 80-1024
            - ip_proto: 6
            - ip_dst_prefix: 192.168.1.2/24
```

그림 63 테스트에 사용된 VNFFGD의 tosca yaml template (1/2)

```
        path:
          - forwarder: VNFD1
            capability: CP12
          - forwarder: VNFD2
            capability: CP22

  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: [VL12,VL22]
        connection_point: [CP12,CP22]
        constituent_vnfs: [VNFD1,VNFD2]
      members: [Forwarding_path1]
```

그림 64 테스트에 사용된 VNFFGD의 tosca yaml template (2/2)

5.2.2. 테스트 결과

테스트는 앞서 설명한 VNFFGD를 온보딩 하는 과정, 그리고 VNFFGD를 통해 VNFFG 인스턴스를 생성하는 과정으로 구성된다.

5.2.2.1. VNFFG Descriptor 온보딩 테스트

먼저 VNFFGD에 대한 온보딩을 Tacker에서 제공하는 command line interface (CLI) (python-tackerclient)를 통해 다음과 같이 수행한다. 이 때, CLI 옵션을 통해 VNFFGD의 tosca yaml template 파일의 경로를 설정해주고, VNFFGD의 이름을 VNFFGD_VNFD1-CP12_VNFD2-CP22로 설정하였다.

그림 65 VNFFGD on-boarding

온보딩이 성공적으로 수행되었는지 확인하기 위해서 Tacker의 CLI 및 UI를 통해 온보딩 된 VNFFGD 리스트를 아래와 같이 확인한다. 그 결과, 앞서 온보딩을 수행한 VNFFGD_VNFD1-CP12_VNFD2-CP22가 성공적으로 온보딩 되어있음을 확인할 수 있다.

그림 66 CLI를 통한 온보딩 VNFFGD 리스트 확인

5.2.2.2. VNFFG 인스턴스 생성 테스트

아래와 같이 Tacker CLI를 통해 앞서 온보딩을 수행한 VNFFGD를 통해 실제 VNFFG 인스턴스를 생성한다. 이 때, CLI 옵션을 통해 인스턴스들의 이름을 VNFFGD로 설정하였다.

VNFFG 인스턴스들이 성공적으로 생성되어 동작하고 있는지를 확인하기 위해 Tacker CLI 및 Web UI를 통해 아래와 같이 확인한 결과, VNFFG의 상태가 Active 임을 확인할 수 있다.

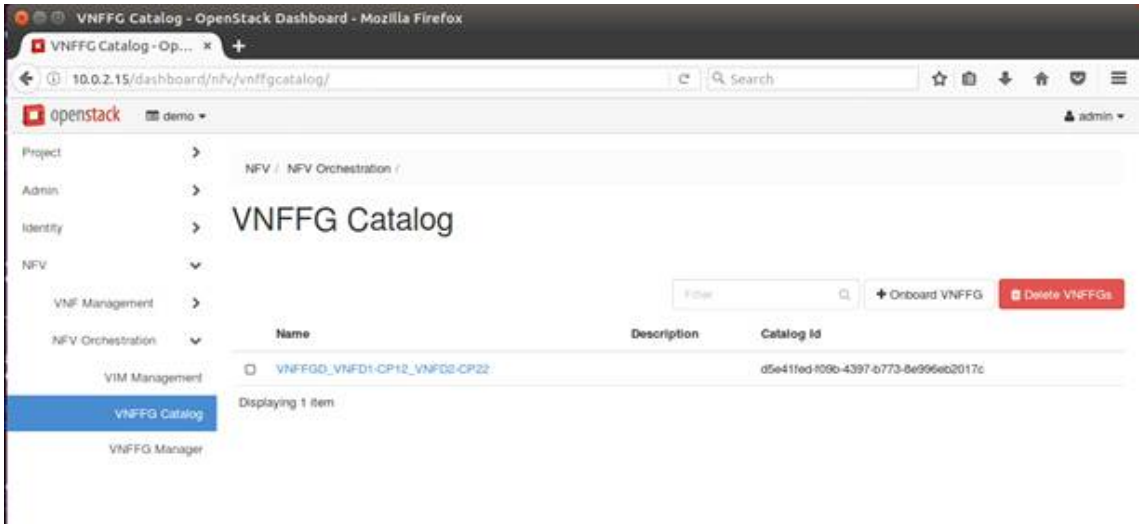

그림 67 Web UI를 통한 온보딩 VNFFGD 리스트 확인

```
stack@stack-VirtualBox:~/devstack$ tacker vnffg-create --vnffgd-name VNFFGD_VNFD1-CP12_VNFD2-CP22 VNFFG
Created a new vnffg:
+-----------------+--------------------------------------------------------------------------------+
| Field           | Value                                                                          |
+-----------------+--------------------------------------------------------------------------------+
| description     |                                                                                |
| forwarding_paths | 27b8390a-ca76-4a96-b824-4a5fb331d8ea                                          |
| id              | 32a5bd0d-e8a8-4a7d-ab98-eba2bf825bf6                                           |
| name            | VNFFG                                                                          |
| status          | PENDING_CREATE                                                                 |
| tenant_id       | e8d0488104d5408aa5b1f93d40bf9a85                                               |
| vnf_mapping     | {"VNFD2": "ebfd8908-0f9c-409b-bcba-0faff1c499af", "VNFD1": "8225216c-ddbc-     |
|                 | 438f-a380-0a0495c0e1f9"}                                                       |
| vnffgd_id       | d5e41fed-f09b-4397-b773-8e996eb2017c                                           |
+-----------------+--------------------------------------------------------------------------------+
```

그림 68 VNFFG 인스턴스 생성

5.3. VNFFG 구성 알고리즘 테스트

본 테스트에서는 OpenStack Tacker VNFFG Manager에서 VNFFG를 구성하는 기존 알고리즘에 대해서 확인해 보도록 한다. OpenStack Tacker VNFFG Manager에서는 VNF Descriptor들의 순서 집합을 참조하는 VNFFG Descriptor로부터 실제 VNF 인스턴스들을 참조하는 VNFFG 인스턴스를 생성할 시, 하나의 VNF Descriptor에 대해 여러 VNF 인스턴스가 존재할 경우, 특정 알고리즘을 통해 VNF 인스턴스를 선택하게 된다. 현재 구현되어 있는 방식은 Manual 방식 및 Random 방식으로, Manual 방식은 VNFFG Descriptor에 포함된 VNF Descriptor와 VNF 인스턴스 간의 매핑을 직접 명시하여 VNFFG 인스턴스를 구성하는 방식이고, Random 방식은 VNF 인스턴스들 중 Random하게 VNF 인스턴스를 선택하여 VNFFG 인스턴스를 구성하는 방식이다. 본 테스트에서는 Random 방식 기능을 검증해보도록 한다.

```
stack@stack-VirtualBox:~/devstack$ tacker vnf-list
+--------------------------------------+------+---------------+-----------------------------------------------------------+--------+--------------------------------------+-----------------------------------------------------------+--------------+
| id                                   | name | description   | mgmt_url                                                  | status | vim_id                               | placement_attr                                            | error_reason |
+--------------------------------------+------+---------------+-----------------------------------------------------------+--------+--------------------------------------+-----------------------------------------------------------+--------------+
| 8225216c-ddbc-438f-a380-0a0495c0e1f9 | VNF1 | Demo example  | {"VDU1": "192.168.120.11"}                                | ACTIVE | 26859e60-3879-4206-8143-9e7088997e5c | {u'vim_name': u'VIM0'}                                    |              |
| ebfd8908-0f9c-409b-bcba-0faff1c499af | VNF2 | Demo example  | {"VDU1": "192.168.120.12"}                                | ACTIVE | 26859e60-3879-4206-8143-9e7088997e5c | {u'vim_name': u'VIM0'}                                    |              |
+--------------------------------------+------+---------------+-----------------------------------------------------------+--------+--------------------------------------+-----------------------------------------------------------+--------------+
```

그림 69 CLI를 통한 VNFFG 인스턴스 리스트 확인

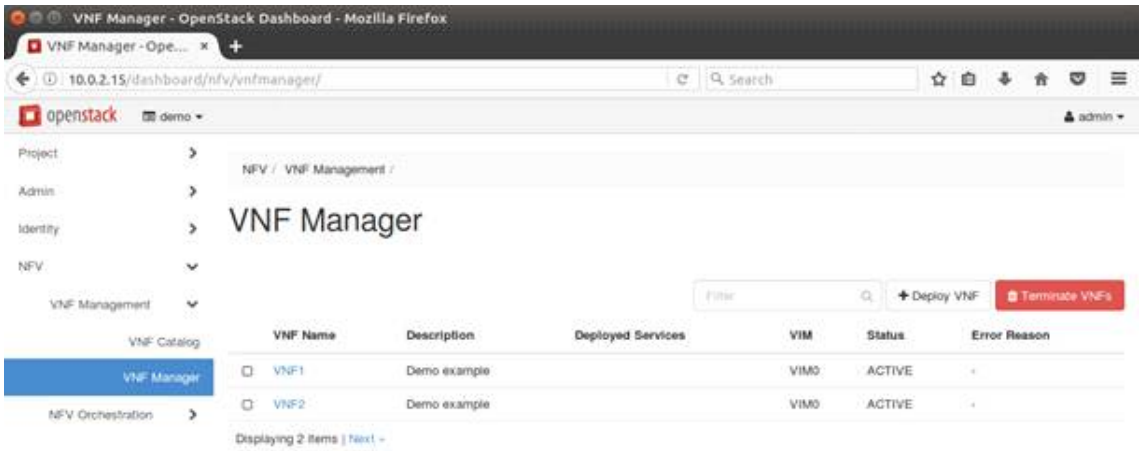

그림 70 Web UI를 통한 VNFFG 인스턴스 리스트 확인

5.3.1. 테스트 환경

VNFFG 구성 알고리즘 테스트 환경 및 테스트 시나리오는 아래 그림과 같다. 먼저, VNFD1 및 VNFD2로 명명된 2개의 VNF Descriptor들의 온보딩을 수행한다. 다음으로 VNFD1의 CP12로부터 VNFD2의 CP22로의 체인을 구성하는 VNFFGD1로 명명된 VNFFG Descriptor를 온보딩 한다.

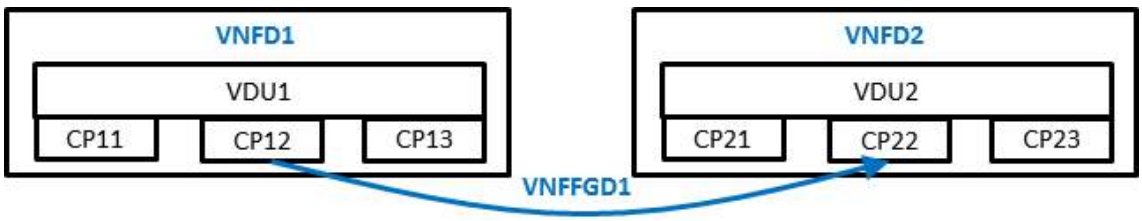

그림 71 VNFFG 구성 알고리즘 테스트 환경

다음으로, VNFFG 구성 알고리즘 테스트 시나리오를 알아본다. VNFD1 및 VNFD2에 대한 VNF 인스턴스를 각각 1개 및 3개 씩 생성한다. 여기서 VNFD1에 대한 VNF 인스턴스는 VNF1_VNFD1 그리고 VNFD2에 대한 VNF 인스턴스들은 VNF1_VNFD2, VNF2_VNFD2, VNF3_VNFD3으로 명명한 것을 알 수 있다. 여기서, VNFD2로부터 생성된 3개의 인스턴스들 중 Random하게 하나의 인스턴스가 선택되

어 VNFFG 인스턴스를 생성하는 것을 확인하도록 한다.

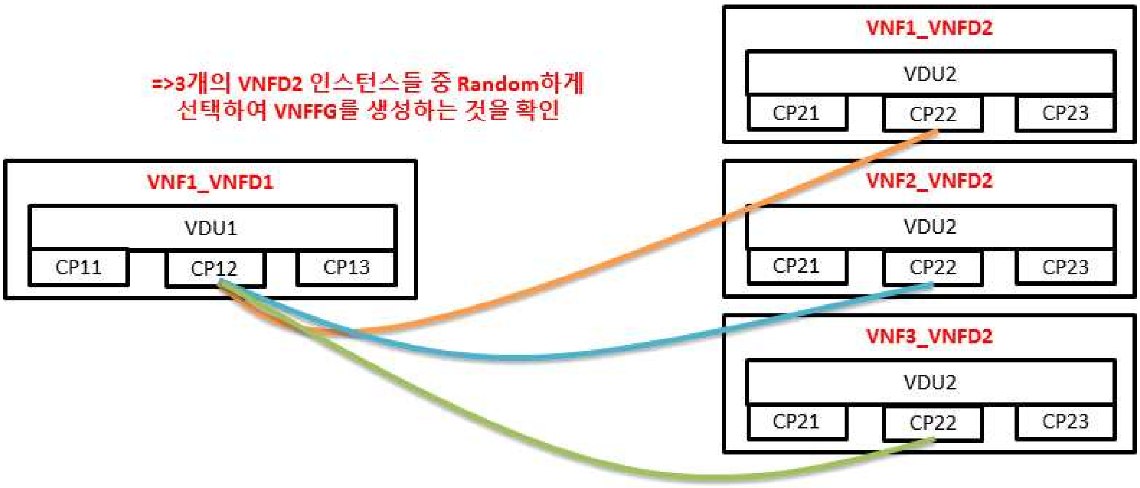

그림 72 VNFFG 구성 알고리즘 테스트 시나리오

5.3.2. 테스트 결과

VNFFG 랜덤 구성 알고리즘의 동작을 확인하기 위해, VNFFG 인스턴스를 생성하여, 생성된 VNFFG 인스턴스가 VNFD2로부터 생성된 3개의 인스턴스들 중 하나를 랜덤하게 선택하는 것을 확인하도록 한다. 먼저 아래 그림과 같이 VDU1의 CP11과 VDU2의 CP22를 연결하는 VNFFG 인스턴스를 생성한다.

```
stack@stack-VirtualBox:~/devstack$ tacker vnffg-create --vnffgd-name VNFFGD1_VDU1_CP12_VDU2_CP22 VNFFG1_VDU1_CP12_VDU2_CP22
Created a new vnffg:
+------------------+------------------------------------------------------------------------------------------------+
| Field            | Value                                                                                          |
+------------------+------------------------------------------------------------------------------------------------+
| description      |                                                                                                |
| forwarding_paths | d5b6ab90-053e-4b38-895a-9d28b935fb4f                                                           |
| id               | 3d9ad691-6a99-446d-b559-fafdab4e7d41                                                           |
| name             | VNFFG1_VDU1_CP12_VDU2_CP22                                                                     |
| status           | PENDING_CREATE                                                                                 |
| tenant_id        | 666c10a2c2f346a8973ba2624257226f                                                               |
| vnf_mapping      | {"VNFD2": "bcbefc46-2001-415c-99a5-d10445809a88", "VNFD1": "34944c47-350c-41e5-9c98-77932504266f"} |
| vnffgd_id        | b5103637-2197-4610-898f-db87592aaee4                                                           |
+------------------+------------------------------------------------------------------------------------------------+
```

그림 73 VNFFG 인스턴스 생성

아래와 같이 Tacker CLI를 통해 VNFFG 인스턴스가 성공적으로 생성된 것을 확인할 수 있다.

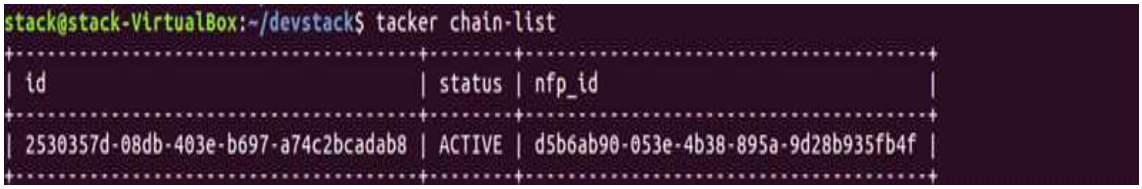

그림 74 VNFFG 인스턴스 생성 확인

다음으로 생성된 VNFFG 인스턴스의 실제 경로를 나타내는 Network Function Path (NFP)를 알아보도록 한다. 아래 그림과 같이, Tacker CLI를 통해 생성한 VNFFG 인스턴스의 NFP를 확인해 본 결과, VNF1_VNFD1 및 VNF3_VNFD2 인스턴스가 NFP에 포함되어있음을 확인할 수 있다. 즉, Random 알고리즘에 의해 VNFD2로부터 생성된 3개의 인스턴스들 중, 세번째 인스턴스 (VNF3_VNFD2)가 선택되었음을 확인할 수 있다.

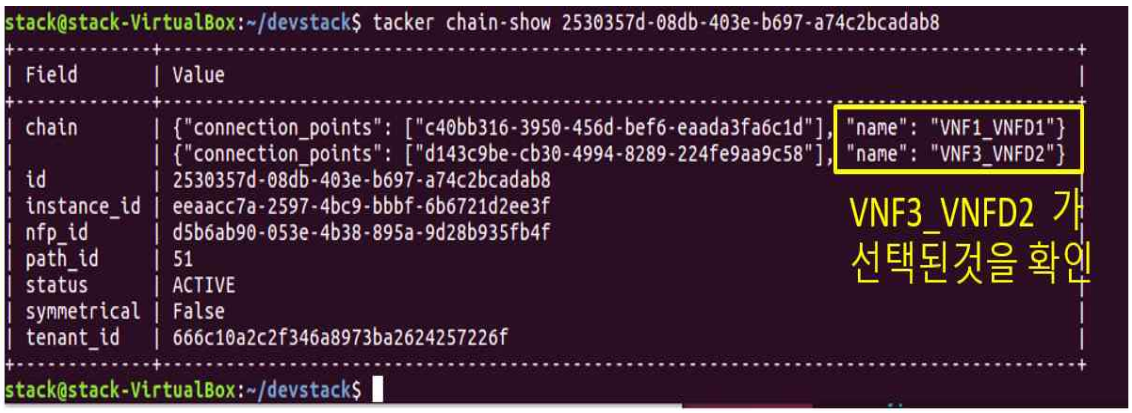

그림 75 VNFFG Random 구성 알고리즘 확인

6. 결 론

본 문서에서는 OPNFV/OpenDaylight SFC 및 OpenStack Tacker 기반의 VNFFG 구성 기능을 분석하였다. VNFFG 관련 소스 코드분석을 통해, VNFFG 생성 명령시 특정 VNF 인스턴스를 지칭하지 않을 시에 VNFD 순서에 맞는 VNF가 임의로 선택되는 것을 파악하였으며, 테스트를 통해 VNFD 순서에 맞게 VNF가 임의로 선택되는 것을 확인 하였다.

하지만, 이러한 임의의 VNF들의 조합 VNFFG를 구성하는 것은 네트워크 사업자의 다양한 요구사항을 충족하기 어려울 것으로 고려된다. 따라서, 네트워크 사업자의 요구에 맞는 VNFFG 구성 알고리즘이 필요하다.

우선 OpenStack Tacker에서 제공하는 Monitoring Framework 기능과 결합하여 부하분산적인 VNFFG 구성 알고리즘이 구현 가능 할 것으로 생각되며, 연구 진행 중에 있다. 하지만, Tacker에서 제공하는 알람기반의 Monitoring Framework의 경우 OpenStack에서 자원 모니터링기능을 제공하는 Ceilometer로부터 특정 이벤트에 대한 알람을 받아오는 형식이기 때문에, 주기적으로 혹은 VNFFG 구성 시에 VNF의 자원정보를 Ceilometer로부터 받아 올 수 있는 모듈 구현이 우선적으로 진행되어야 한다. 해당 모듈의 구현이 완료되면 VNF들의 부하를 기반으로 VNFFG를 구성하는 모듈을 구현함으로써 부하 분산적인 VNFFG 구성 알고리즘이 구현된다.

References

[1] OpenDaylight, [Online] Available : https://www.opendaylight.org/

[2] Halpern, J., Ed. and C. Pignataro, Ed., "Service Function Chaining (SFC) Architecture," RFC 7665, DOI 10.17487/RFC7665, October 2015, <http://www.rfc-editor.org/info/rfc7665>.

[3] ETSI ISG NFV, http://portal.etsi.org/portal/server.pt/community/NFV/367

[4] OPNFV, "Open Platform for Network Function Virtualization," [Online] Available : https://www.opnfv.org/

[5] Quinn, P. and U. Elzur, "Network Service Header", draft- ietf-sfc-nsh-04, March 2016.

[6] "Multisite VIM support in Tacker." [Online] Available : https://specs.openstack.org/openstack/tacker-specs/specs/mitaka/multi-site-feature.html

[7] "Multisite VIM Usage." [Online] Available : https://docs.openstack.org/developer/tacker/devref/multisite_vim_usage_guide.html

[8] "Tacker Monitoring Framework." [Online] Available : https://docs.openstack.org/developer/tacker/devref/monitor-api.html

[9] "Alarm monitoring framework." [Online] Available : https://docs.openstack.org/developer/tacker/devref/alarm_monitoring_usage_guide.html

[10] "Enhanced Placement Awareness Usage Guide." [Online] Available : https://docs.openstack.org/developer/tacker/devref/enhanced_placement_awareness_usage_guide.html

K-ONE 기술 문서

- 이 문서의 기술적인 내용은 프로젝트의 진행과 함께 별도의 예고 없이 변경될 수 있습니다.
- 본 문서와 관련된 문의 사항은 아래의 정보를 참조하시길 바랍니다.
  (Homepage: http://opennetworking.kr/projects/k-one-collaboration-project/wiki, E-mail: k1@opennetworking.kr)

작성기관: K-ONE Consortium
작성년월: 2017/05