K-ONE 기술 문서 #5
OPNFV에서의 Fault Management 관련 프로젝트 현황

Document No. K-ONE #5
Version 1.0
Date 2016-04-22
Author(s) 김민식

K-ONE

ONK
OpenNetworkingKorea

■ 문서의 연혁

버전	날짜	작성자	비고
초안 - 0.1	2016. 03. 01	김민식	초안 작성
0.2	2016. 03. 12	김민식	그림 추가
0.3	2016. 03. 24	김민식	데모 내용 추가
0.4	2016. 04. 11	김민식	유즈케이스 내용 수정
1.0	2016. 04. 22	김민식	최종 점검(오타, 형식)

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

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

기술문서 요약

ETSI NFV그룹에서는 기존에 네트워크 기능들을 하드웨어로 구현한 방식과 달리 컴퓨팅, 서버, 네트워크 기능들을 추상화하여 소프트웨어로 구현한 기술을 정의하고 있다. Network Function Virtualization(NFV)[1]는 하드웨어와 소프트웨어를 분리함으로써 네트워크의 유연성을 높일 수 있고 기존의 고가의 하드웨어 장비가 아닌 일반 범용 서버에서 구현하기 때문에 총소유 비용을 절감할 수 있다.[2] NFV의 목적은 산업적인 표준을 정의하여 네트워크 오퍼레이터들에게 새로운 서비스를 유연하고 확장성 있게 제공할 수 있는 환경을 정의하고 가속화시키는 것이다. NFV구조[3]는 크게 Virtual Network Function(VNF), VNF 인프라, NFV 오케스트레이션으로 구성되어 있고 각 모듈별로 인터페이스를 통해 총체적인 시스템을 관리하게 된다. NFV에서 정의된 표준 레퍼런스 구조와 인터페이스를 실제로 구현하고 가속화하기 위해서 오픈소스 소프트웨어 플랫폼인 Open Platform for NFV(OPNFV)[4] 프로젝트가 발표되었다. OPNFV의 목적은 NFV 솔루션을 개발하고 사업화를 촉진하여 NFV에서 정의한 요구사항들을 만족하는 프레임워크를 생성하는 것이다.

OPNFV는 현재 구현되어 있는 오픈소스 프로젝트와의 차이점을 정의하고 필요한 요구사항들을 직접 개발하고 있다. 현재 OPNFV가 정의하고 개발하고 있는 범위는 인프라 레벨이다. 네트워크 서비스를 제공할 때 중요한 요구사항중 하나가 결함에 대한 빠른 처리이기 때문에 인프라에 대한 모니터링 기능은 필수적이다. 하지만 현재 인프라 플랫폼인 오픈스택에서는 OPNFV에서 정의하고 있는 모니터링 요소들을 전부 모니터링 하지 못하고 성능에 관한 요구사항도 만족시키지 못하고 있다.

본 기술문서에서는 위 문제점을 보안하고 해결하기 위해 제안된 OPNFV Doctor 프로젝트[5]의 현황과 데모 시나리오에 대한 자세한 소개를 한다. Doctor 프로젝트에서의 데모 시나리오를 테스트할 때 주의점과 OpenStack 프로젝트의 설정방법[6]을 소개하여 테스트 환경을 구축할 수 있도록 해준다.

Contents

K-ONE #5. OPNFV에서의 Fault Management 관련 프로젝트 현황

그림 목차

1. Introduction

OPNFV Doctor프로젝트의 목표는 NFVI 결함에 대한 관리와 사용자에게 제공되는 네트워크 서비스에 대한 고가용성을 지원하는 것이다. Doctor프로젝트에서 제공하는 중요한 기능은 VIM으로부터 사용할 수 없는 리소스에 대해서 즉각적인 알림을 받고 NFVI에 깔려있는 VNF들의 결함에 대한 복구를 진행하는 것이다. 이 프로젝트는 NFVI 결함관리나 유지에 대해서 개발하고 있는 상위 프로젝트의 빠진 부분이나 요구사항들을 정의한다. 또 정의한 요구사항들을 가지고 연관된 오픈소스 프로젝트(OpenStack)에서 직접 코드개발에 참여하고 있다. ETSI NFV에서 정의하고 있는 NFVI 결함 관리나 유지보수에 대해서도 필요한 VIM Northbound 인터페이스를 정의하고 있다.

현재 OPNFV에서 VIM은 OpenStack을 사용하고 있지만 특정 NFVI의 결함을 찾아내기에는 부족함이 있다. Doctor프로젝트는 EPC[7]에 있는 S-GW와 MME같은 VNF들이 끊김없이 잘 운용될 수 있도록 NFVI에서 필요한 요소들의 결함을 찾고 Consumer(VNFM or NFVO)에게 알리는 기능과 구조를 정의하고 있다. 또한 VIM은 Consumer로부터 maintenance에 관련된 명령을 받아야 한다. 예를 들어서 NFVI에 있는 하이퍼바이저의 버전을 업데이트하는 경우에 하이퍼바이저에 속해 있는 VM을 정책에 맞게 처리해야 한다. 이런 maintenance와 관련된 명령어가 필요하고 VIM에서는 이 명령어를 처리할 수 있어야 한다.

2. Usecase and Scenario

Doctor 프로젝트는 NFV의 표준문서의 요구사항에 따른 실제 구현을 가속화하는 프로젝트이기 때문에 Telecom 서비스를 한 유즈케이스로 설명하고 있다. 실제 Telecom 서비스는 높은 요구사항을 가지고 있고 서비스를 안전하게 운영하기 위해서 플랫폼을 추가하여 고가용성을 높이는 매커니즘을 수행한다. 이런 고가용성을 만족하기 위해서는 모니터링을 통해 최대한 빠르게 오류를 찾고 곧 바로 알림 메시지를 전달하여 서비스에 최대한 적게 영향을 주어야 한다. Doctor프로젝트에서는 VIM/NFVI 환경안에서 빠르게 오류를 탐지하고 복구하여 고가용성을 제공하는 구조를 제안한다.

2.1. Fault management

Doctor 프로젝트에서 VIM은 OpenStack으로 구현하였고 이 VIM은 NFVI와 Consumer와의 인터페이스를 가지고 있다. Consumer들은 NFVI에 각자 자기 소유의 VM들을 가지고 있고 여러 VM을 통해 서비스를 제공하고 있다. 만약 Consumer가 소유하고 있는 VM에게 영향을 끼치는 오류가 NFVI에 일어나게 된다

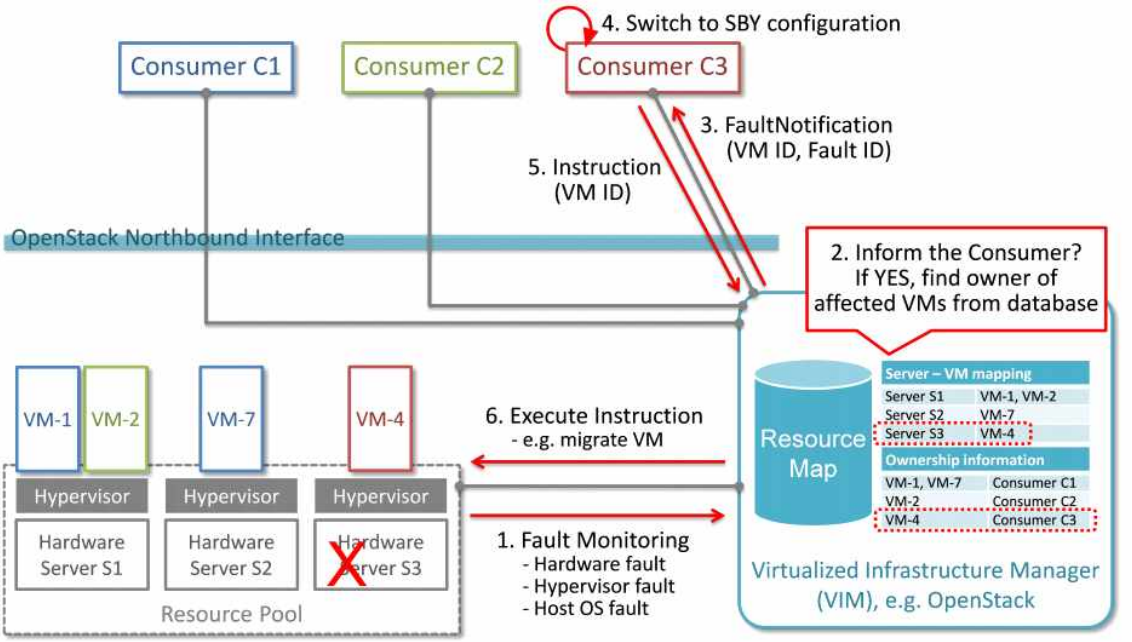

그림 1. Fault management/recovery use case

면 그림 1과 같이 Consumer는 VIM으로부터 NFVI의 이상이 생겼다는 정보를 전달받아 Fault management 유즈케이스가 시작이 된다. Fault management를 실제로 제공하기 위해서 VIM은 최대한 빠르게 결함을 감지해야 한다. 결함을 감지하기 위해서 VIM은 Controller에게 해당 결함을 알려주는 모니터링 기능이 있어야 한다. Doctor 프로젝트에서는 오픈스택 자체에 있는 모니터링 기능과 써드파티 모니터링 툴(e.g. Zabbix[8])의 비교를 통해 Doctor 프로젝트의 요구사항과의 차이점을 정의하고 어떤 모니터링 툴을 사용할 지를 결정중이다. 모니터링 툴을 통해 NFVI 레벨의 오류를 검출하고 오류정보를 기반으로 VIM이 가지고 있는 리소스 테이블을 가지고 오류로 인해 영향을 받는 VM을 찾는다. 그리고 해당 VM을 소유하고 있는 Consumer를 찾아 오류가 났음을 알리게 된다. 오류정보를 받은 Consumer는 ACT-STBY 매커니즘을 사용하여 STBY에 있는 노드를 ACT로 바꾸고 새로운 STBY 노드를 새로 설정하게 된다. 이 설정과정은 VNFM/NFVO의 역할이기 때문에 Doctor 프로젝트는 오류가 났다는 정보를 알려주는 인터페이스까지만 정의한다. STBY 설정이 끝나고 나서 Consumer는 오류에 대한 처리 명령을 내리게 되고 VIM에 있는 미리 선언된 정책에 따라 명령을 수행하게 된다.

또한, VIM 안에는 모니터링을 통해 얻은 이벤트를 가지고 분석하여 결함을 예방할 수 있는 모듈이 있다. 이 결함을 예상하는 모듈은 실제 OPNFV에서 Prediction[9]이라는 프로젝트에서 담당하고 있다. 간단한 예로써 만약 하드웨어 서버의 온도가 올라가게 되면 NFVI에서는 먼저 이 정보를 Prediction 모듈로 보내게 된다. Prediction 모듈은 저장된 정보들을 해석해 오류를 예상하게 되고 NFVI 오류에 영향을 받는 VM을 소유한 Consumer에게 결함사실을 알려 결함을 사전에 처리하게 되는 것이다.

2.2. NFVI Maintenance

현재 환경에서 네트워크 오퍼레이터들은 해당 서버의 하드웨어 장비를 주기적으로 점검한다. 만약 인프라를 가상환경으로 옮기게 되면 하드웨어 뿐만 아니라 하이퍼바이저나 호스트 OS와 같은 요소도 유지점검할 필요가 있다. 이렇게 점검이 되는 요소들은 점검이 끝날 때까지 서비스를 제공하지 못한다. Doctor 프로젝트에서는 이런 점검이 일어날 때에 어떻게 대처해야 될 것인지에 대해서 구조를 정의하고 있다. 먼저 그림2를 보면 해당 시나리오가 관리자(Administrator)로부터 시작됨을 알 수 있다. 관리자가 해당 NFVI의 점검이 필요할 경우 VIM에게 알리게 된다. VIM은 해당 정보를 가지고 어떤 VM이 영향을 받는지를 판단하여 해당 VM을 소유하고

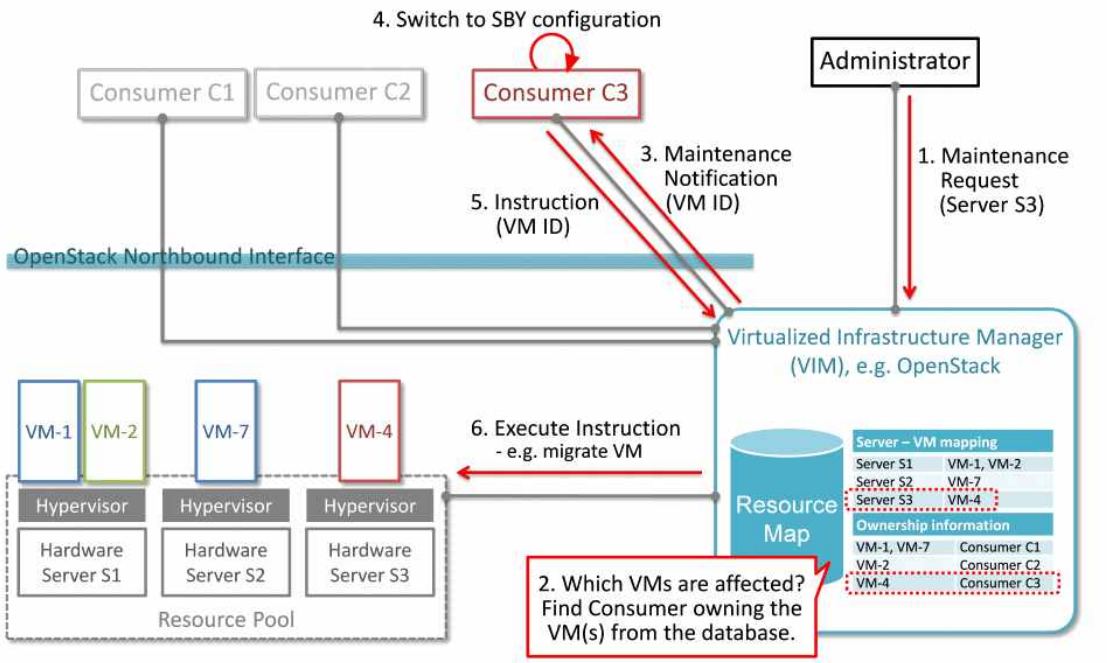

그림 2. Maintenance use case

있는 Consumer에게 점검이 있음을 알리게 된다. Comsumer는 즉각적으로 문제를 해결하기 위해 안전하게 STBY 설정이 끝내고 VIM에게 알려 점검에 대한 처리 명령을 내리게 된다.

3. Functional Overview

Doctor 프로젝트에서는 Chap 2에서 설명한 것과 같이 2개의 유즈케이스가 존재한다. 2개의 유즈케이스 모두 인프라에 문제가 발생했기 때문에 해당 인프라에서 돌아가고 있는 Virtual Network Function(VNF)나 어플리케이션에 영향을 끼치게 된다. 그러나 2개의 유즈케이스의 발생빈도가 다르기 때문에 해결책도 다르다. 인프라의 결함은 예상이 가능하지 않기 때문에 Consumer는 결함을 가능한 빨리 처리해야

할 것이다. 또 어플리케이션에서 고가용성 기능을 지원한다면 결함의 영향이 적을 것이고 충분한 네트워크 노드들이 미리 설치되어 있다면 어떤 리소스가 고장이 났을 때 문제없이 처리할 수 있다. 결함을 처리하는 유즈케이와는 반대로 미리 계획된 관리 유즈케이스의 경우에는 Comsumer에게 미리 알려줄 수 있고 해당 이벤트에 대해서 대비를 할 수 있기 때문에 문제를 최소화 시킬 수 있다. 또 다른 한가지 경우는 모니터링 툴을 사용하여 인프라에서 얻은 정보들을 가지고 결함을 예측하는 경우이다. 결함을 예측하게 된다면 결함이 나기전에 문제들을 해결할 수 있는 여유가 생기기 때문에 더욱더 효과적으로 결함을 처리할 수 있게 된다. Doctor 프로젝트는 결함을 처리함에 있는 결함예측 기능은 구현하지 않고 OPNFV에서 다른 프로젝트인 Prediction 프로젝트에서 모듈을 정의하고 구현한다.

4. Architecture Overview

NFV와 클라우드 플랫폼에서는 가상화된 자원들을 지원하고 이 자원들을 관리할 수 있는 기능을 제공한다. 그림 3은 NFVI를 중점으로 표현한 NFV 구조이다. NFVI는 가상 머신과 가상 네트워크 등을 지원하고 이 자원들을 사용해서 어플리케이션들을 실행시킨다. 이 네트워크 서비스를 제공하는 컴포넌트들을 NFV에서는 Virtual Network Functions(VNFs)라고 한다. VIM은 Consumer(NFVM/NFVO)에게 가상 자원들에 대한 정보를 제공하고 관리할 수 있도록 기능들을 제공한다. Doctor 프로젝트는 OpenStack을 VIM의 유력한 후보로 정했고 실제로 OpenStack에 있는 여러 프로젝트에서 정의한 모듈과 Doctor에서 정의한 명세를 비교하여 요구사항을 제안하고

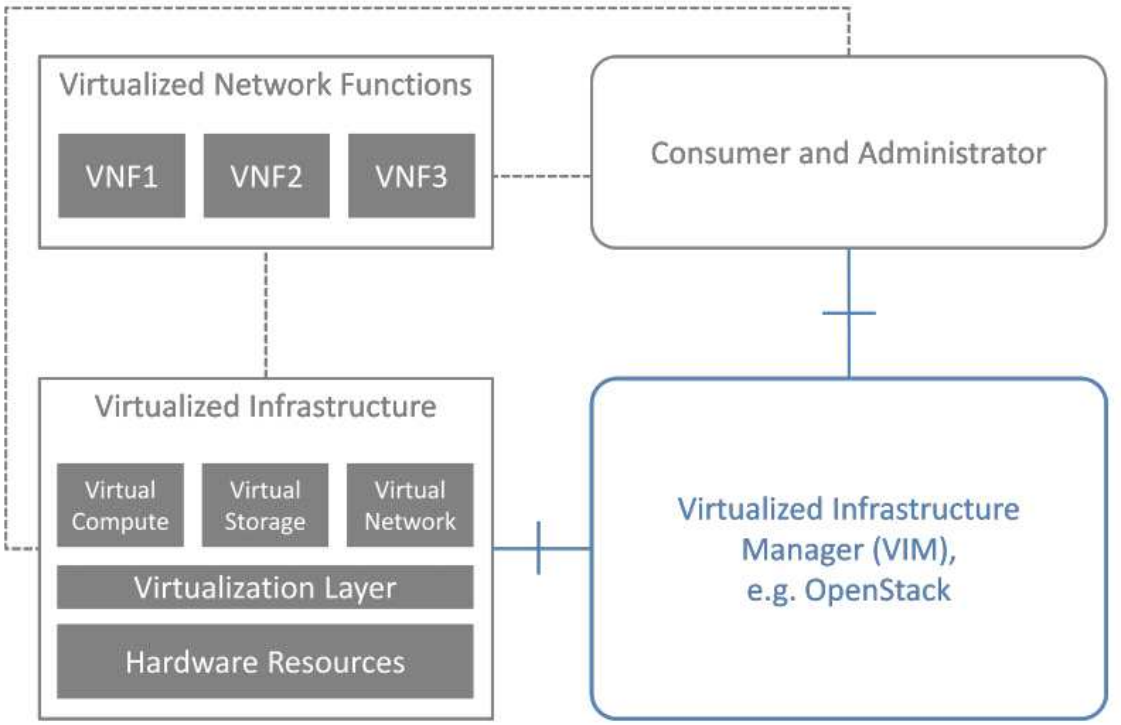

그림 3. High level architecture

실제 구현에도 참여했다. 그러나 Doctor프로젝트에서 정의한 명세를 OpenStack에 있는 프로젝트만 가지고는 부족하기 때문에 새로운 기능을 제공하는 요소(Monitor, Inspector)들을 찾고 구현할 계획을 세우고 있다. 이 요소들은 OpenStack 밖에 위치할 예정이고 Northbound 인터페이스를 통해서 오픈스택과 통신할 것이다.

5. General Features and Requirement

어플리케이션이나 네트워크 서비스의 고가용성을 제공하기 위해서는 다음과 같은 특징들이 VIM에 요구된다.

1. **Monitoring** : 물리적 · 논리적인 리소스를 모니터
2. **Detection** : 물리적인 리소스의 결함을 분석 및 확인
3. **Correlation and Congnition** : 결함과 연관된 리소스 인식
4. **Notification** : Consumer에게 결함과 관련된 정보 알림
5. **Fencing** : 오류난 리소스를 분리시키는 서비스
6. **Recovery action** : 결함을 복구하거나 관리에 필요한 일들을 실행시키는 서비스

5.1. Monitoring

VIM에서 결함관련된 일들을 처리하기 위해 물리적인 리소스와 가상화된 리소스 정보들을 모은다. Doctor 프로젝트에서는 VNF레벨의 어플리케이션의 상태정보가 아닌 인프라에 관한 상태정보를 모은다는 것에 주의해야 한다. OpenStack 프로젝트인 Ceilometer[10]가 VM에 관련된 가상자원이나 인프라에 대한 리소스 자원들을 모니터링 할 수 있으나 Doctor 프로젝트에서 정의한 요구사항을 만족하기에는 부족하고 실제 서버스를 제공하기에는 성능도 나오지 않기 때문에 써드파티 모니터링 툴인 Zabbix나 Monasca[11] 등의 사용을 논의중이다. Doctor에서 결함임을 판단하기 위해 필요한 모니터링 요소들은 Requirement 문서의 Chapter 7을 참조하면 된다.

5.2. Detection

Detection 모듈은 모니터링에서 모은 정보를 기반으로 어떤 리소스가 고장이 났는지를 찾는다. 또한 많은 정보들은 위에서 설명한 것 같이 결함을 예상하는 모듈에서도 사용될 수 있다. 이런 오류를 판단하는 기준은 관리자의 정책에 의해서 설정이 가능해야 하며 어떤 오류를 잡을 것인지, 모은 정보들을 가지고 어떻게 오류를 판단할지 등을 설정해 오류를 찾게 된다. 위에서 설명한 것 같이 이 모듈에서는 결함을 예상하는 모듈이 포함될 수 있다. 모니터링을 이용해서 Doctor프로젝트에서 요구하는 정보들을 이용해 결함이 일어날 것이라는 것을 예상해 서비스에 대한 가용성을 증가시킬 수 있다.

5.3. Correlation and Cognition

오류가 판단이 되면 VIM은 결함과 관련된 가상 리소스를 데이터베이스로부터 찾는다. 그림 1에 나와 있는 Resource Map을 보면 VM mapping과 Ownership information 데이터베이스를 볼 수 있다. 이 데이터베이스에 있는 테이블 정보를 가지고 결함이 일어난 물리 리소스와 결함과 관련된 VM을 소유한 Consumer를 찾게 된다. 그런 다음 해당 Consumer에게 이벤트가 일어 났음을 알리게 된다.

5.4. Notification

VIM은 가상 리소스에 대한 오류가 일어났음을 결함과 관련된 VM의 소유자인 Consumer에게 Northbound 인터페이스를 통해 알리게 된다. 또한 물리적인 리소스에 오류가 났음을 administrator에게 알린다. 모든 알림은 피해를 최소화하기 위해 가능한 빠른 시간안에 결함정보와 함께 전달되어야 하며 알림을 받은 Consumer은 해당 결함에 맞는 적절한 행동을 취해야 한다. 만약 Consumer가 해당 리소스에 대한 정보를 주기적으로 가져오게 된다면 VIM에게 많은 부담을 주기 때문에 publication/subscription 메시지 모델이 더 적절하다고 판단하고 있다. VIM으로부터 전달되는 알림 메시지 또한 Consumer에서 설정한 정책을 기반으로 판단되고 실행되어지며 알림 메시지 안에는 Consumer가 원하는 결함에 대한 정보가 포함되어 있어야 한다.

5.5. Fencing

결함에 대한 복구가 진행되기 전에 반드시 오류가 일어난 호스트를 구분시켜 줘야 한다. 오류가 일어났음을 호스트가 가지고 있는 상태에 표시하게 되면 2개의 같은 서비스가 동시에 실행되어 충돌을 일어나는 것을 방지할 수 있다. 현재 오픈스택에서는 Fencing과 관련된 프로젝트가 없고 단순이 언급만 되어 있다. 그래서 Doctor 프로젝트는 Fencing 기능을 넣기 위해 pacemaker 툴을 이용하거나 OpenStack API를 불러 오류난 호스트를 구분시킬 계획에 있다. Doctor의 Functional Block에 있는 Inspector 컴퍼넌트가 오류가 생긴 리소스를 구분하여 Fencing기능을 구현할 예정이다.

5.6. Recovery Action

VIM으로부터 알림을 받은 Consumer가 결함이나 관리 정보를 기반으로 적절한 행동을 결정하여 VIM에게 행동을 하도록 명령을 내린다. Consumer로부터 명령을 받을 수 있지만 VIM은 어떤 이벤트에 대해 미리 정의된 행동에 따라 적절한 행동과 조치를 취할 수도 있다. 명령에 대한 간단한 예로는 VM을 다시 실행시키거나 VM을 이전시키는 등의 명령이 있다.

6. DETAILED ARCHITECTURE

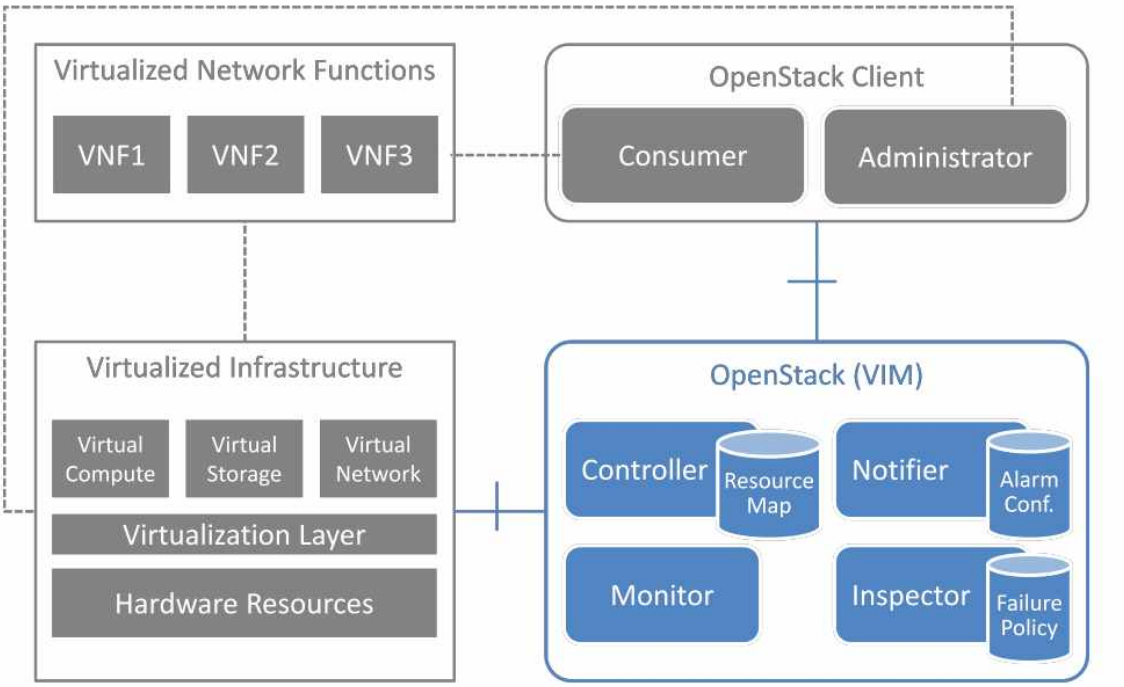

그림 4. Functional blocks

그림 4는 Doctor 프로젝트의 기능적인 블록들을 나타낸다. Doctor 프로젝트를 실제로 구현할 때 OpenStack을 VIM으로 선택하였고 VIM안에 4개의 블록들(Monitor, Inspector, Controller, Notifier)을 정의하였다. 각각 블록들의 기능은 다음과 같다.

6.1. Monitor

모니터 모듈은 가상화된 자원들을 모니터링한다. Doctor 프로젝트에서는 현재 OpenStack에서 모니터링 구현되어 있는 프로젝트인 Ceilometer에서 모니터하는 요소들과 자신들의 프로젝트에서 모니터하는 요소들과의 차이점을 분석하였다. 현재 OpenStack에서 모니터링하는 요소들이 Doctor 프로젝트에서 결함을 관리하기 위해 필요한 요소들과 비교했을 때 몇몇 요소들이 부족하고 또한 모니터하는 요소들을 처리하는 성능이 부족하다고 판단하였다. 현재 Doctor 프로젝트에서는 Zabbix같은 다른 오픈소스 툴을 사용하기 위해 분석하고 있다.

6.2. Inspector

Inspector 모듈은 모니터 모듈로부터 물리적인 리소스에 대한 정보를 받고 컨트롤러에게 해당 정보를 전달한다. 컨트롤러는 리소스 맵핑 테이블을 가지고 결함이 생긴 물리적인 리소스에게 영향을 받는 가상 리소스 정보를 다시 Inspector 모듈에게 주게 된다. Inspector 모듈은 받은 정보를 업데이트 시켜 결함에 대한 처리를 결정하

게 된다. Inspector 모듈은 관리자로부터 결함에 대한 정책 데이터베이스를 내려 받게 되고 그 정책을 기반으로 여러 이벤트들을 연관시켜 결함을 결정하게 된다. Controller 와 Inspector 모듈을 분리시킨 이유는 Controller에서 물리적인 리소스를 지속적으로 관찰하는 다양한 매커니즘을 분리시켜 단순한 작업을 하도록 설계하기 위해서이다.

6.3. Controller

컨트롤러는 물리적인 리소스와 가상 리소스에 대한 맵핑 테이블을 관리하고 있다. 이 테이블은 업데이트가 가능하고 가상 리소스에 문제가 발생하게 되면 Notifier에게 결함에 관련된 모든 이벤트를 전달한다. 컨트롤러는 Inspector로부터 물리적인 리소스의 결함 알림을 받게 되면 NFVI의 용량을 재계산하여 정보를 유지하게 된다. 실제 구현에서 VIM은 여러 Controller를 가지게 되는데 일반적으로 Nova, Cinder, Neutron 등의 다른타입이 되고 각각의 Controller들은 노드들이 가지는 물리적인 리소스와 가상 리소스들의 맵핑 테이블을 데이터베이스에 저장하고 있다.

6.4. Notifier

Notifier는 Consumer로부터 받은 정책을 기반으로 Controller가 전달한 결함 이벤트들을 발생시키고 결합하는 역할을 한다. Consumer로부터 가상 리소스에 문제가 발생 시 자신에게 알림을 주라는 등록으로부터 시작한다. 등록이 끝나면 Controller부터 결함 이벤트를 받게 되고 만약 리소스가 특정 Consumer가 정의한 정책에 맞게 되면 해당 Consumer에게 알림 메시지를 보내게 된다. OpenStack의 설계 철학에 따라 Controller에서 필터링을 하여 Notifier에게 알리는 것 보다는 모든 알림 메시지를 Notifier에게 알리고 Notifier에서 필터링을 수행하는 것이 복잡성을 줄일 수 있다.

7. Sequence

7.1. Fault Management

결함 관리에 대한 흐름은 그림 5와 같다.

1. Consumer는 특정 가상 리소스에 대해서 알림을 받기 위한 등록을 요청하게 된다. 그리고 알림을 요청할 때에 필터링을 위한 정보를 함께 전달하여 Consumer가 원하는 알림만을 받게 한다.
2. 등록요청에 대한 응답메시지에 Controller가 가지고 있는 가상 리소스 정보를 함께 넣어 Consumer에게 보내게 된다.
3. 등록이 완료된 후 NFVI는 VIM이 등록한 리소스에 대한 이벤트 메시지를 받게 된다. 여기서 VIM와 NFVI의 등록절차는 먼저 수행되었다고 가정한다.

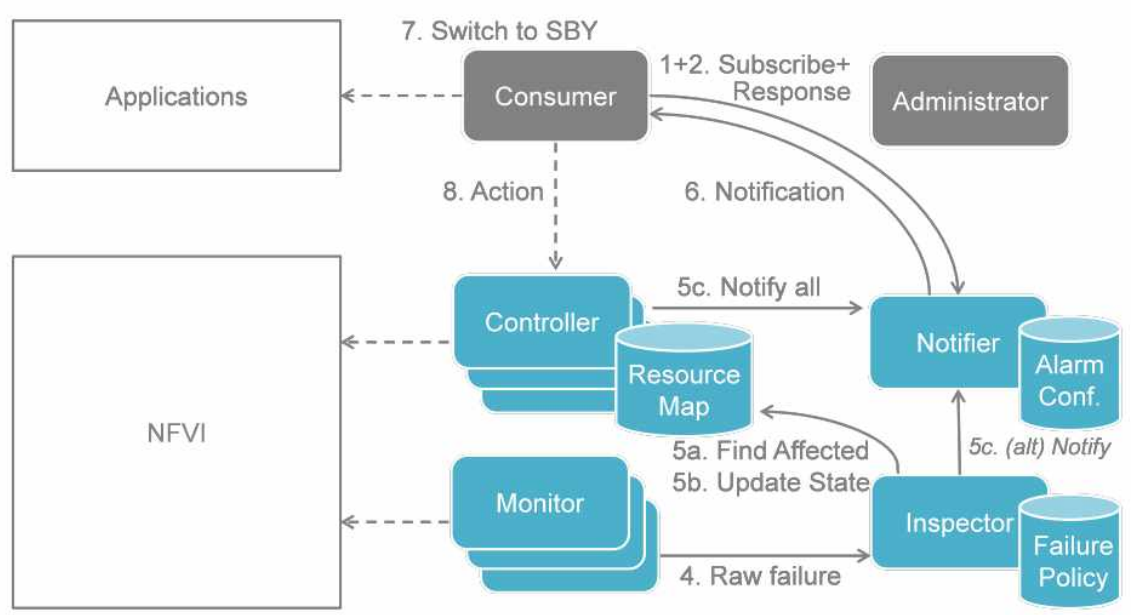

그림 5. Fault management work flow

4. Monitor는 NFVI에서의 결함을 발견하게 되고 Inspector에게 결함에 관련된 정보를 보내게 된다. 결함 메시지를 받은 Inspector는 미리설정된 정책을 사용해서 메시지를 필터하고 결합한다.
5. Inspector는 결함이 발생한 물리적인 리소스로 인해 문제가 생기는 Consumer를 찾기 위해 Controller에게 요청 메시지를 보내고 Controller는 리소스 맵핑 데이터베이스에 해당 정보를 업데이트 하게 된다. Controller는 업데이트된 정보를 통해 상태가 바뀌었음을 알게 되고 모든 알림 메시지를 Notifier에게 전달한다.
6. Notifier는 등록할 때 받은 필터정보를 통해 Controller로부터 받은 메시지를 필터링을 하여 Consumer에게 전달하게 된다.
7. Consumer는 자신이 소유하고 있는 어플리케이션의 가상 리소스가 문제임을 알게 되고 STBY 설정을 하게 된다.
8. STBY설정이 끝나게 되면 해당 리소스를 이동, 업데이트, 종료 등을 VIM에게 명령하게 된다. 명령을 받은 VIM은 해당 명령을 수행하여 결함을 해결하게 된다.

7.2. NFVI Maintenance

NFVI maintenance는 fault management의 흐름과 매우 비슷하기 때문에 Release 1에서는 서로 공통되는 부분을 공유하였다. Maintenance에 대한 흐름은 그림 6와 같다.

1. Consumer는 Subscribe 요청과 함께 SubscribeFilter 정보를 VIM에게 전달한다. 여기서 필터정보는 Controller가 어떤 정보를 Consumer에게 전달해야 되는지를 알려준다.
2. Subscribe 요청을 받은 VIM은 요청에 해당되는 가상자원 정보와 함께 SubscribeResponse 응답 메시지를 Consumer에게 보낸다.

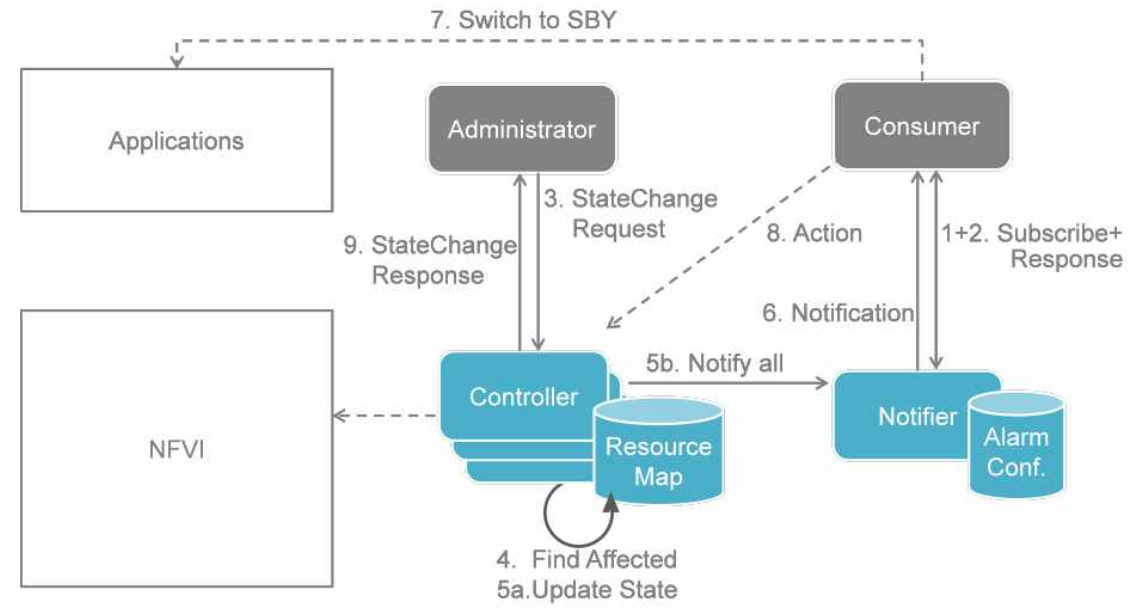

그림 6. NFVI maintenance work flow

3. Administrator가 관리를 목적(예) 하이퍼바이저 업그레이드)으로 해당하는 노드의 state를 "Maintenance"로 설정하라는 요청 메시지를 VIM에게 전달하게 된다.
4. VIM은 해당 리소스의 상태를 "Maintenance"로 바꾸게 되는데 이 의미는 더 이상 이 리소스를 사용할 수 없다라는 뜻으로 해석이 된다.
5. VIM은 Maintenance 명령에 영향을 받는 가상 리소스 자원와 해당 Consumer를 데이터베이스 맵핑 테이블에서 찾는다.
6. VIM은 Consumer에게 Maintenance 명령에 대한 사실을 알린다.
7. Consumer는 STBY 설정을 통해 해당 ACT노드를 STB로 바꾸고 STB노드를 ACT 노드로 바꾼다.
8. Consumer는 VIM에게 자신들의 서비스에 문제가 없도록 특정 명령을 수행하고 해당 물리적인 노드에 어느 자원도 사용되지 않도록 만든다.
9. VIM은 Administrator에게 Maintenance를 요청했던 노드가 비었음을 알린다.
10. Administrator는 해당 NFVI에 관리에 필요한 동작을 수행한다.

8. Implementation plan for OPNFV Release 1

8.1. Fault management

그림 7은 오픈스택 기반의 구현 계획과 Release 1에서 구현해야 하는 컴포넌트들이다. 모니터는 Zabbix툴로 Doctor 프로젝트에서 필요한 인프라 요소를 모니터링할 계획이다. 컨트롤러는 compute, network, storage를 담당하는 Nova, Neutron, Cinder 3개로 구성된다. Inspector는 OpenStack프로젝트에서 서비스를 위한 모니터

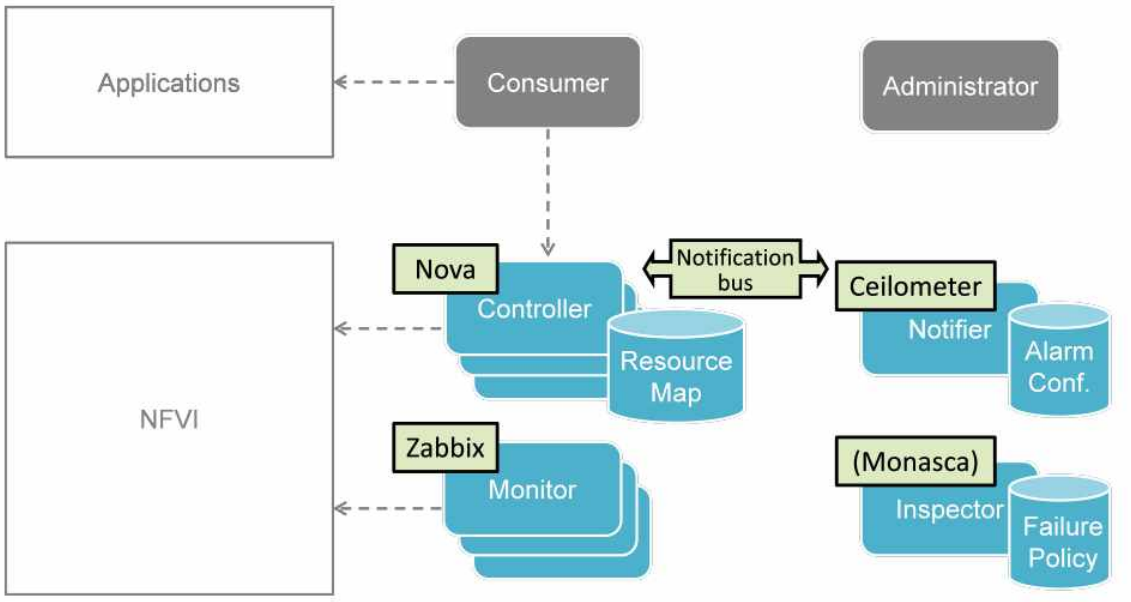

그림 7. Implementation plan in OpenStack

링 기능을 제공하는 Monasca나 물리적 리소스와 가상 리소스를 맵핑시키는 간단한 스크립트를 Nova에 구현할 수도 있다. Notifier의 경우에는 OpenStack의 Telemetry 프로젝트인 Ceilometer를 이용해 구현할 예정이다.

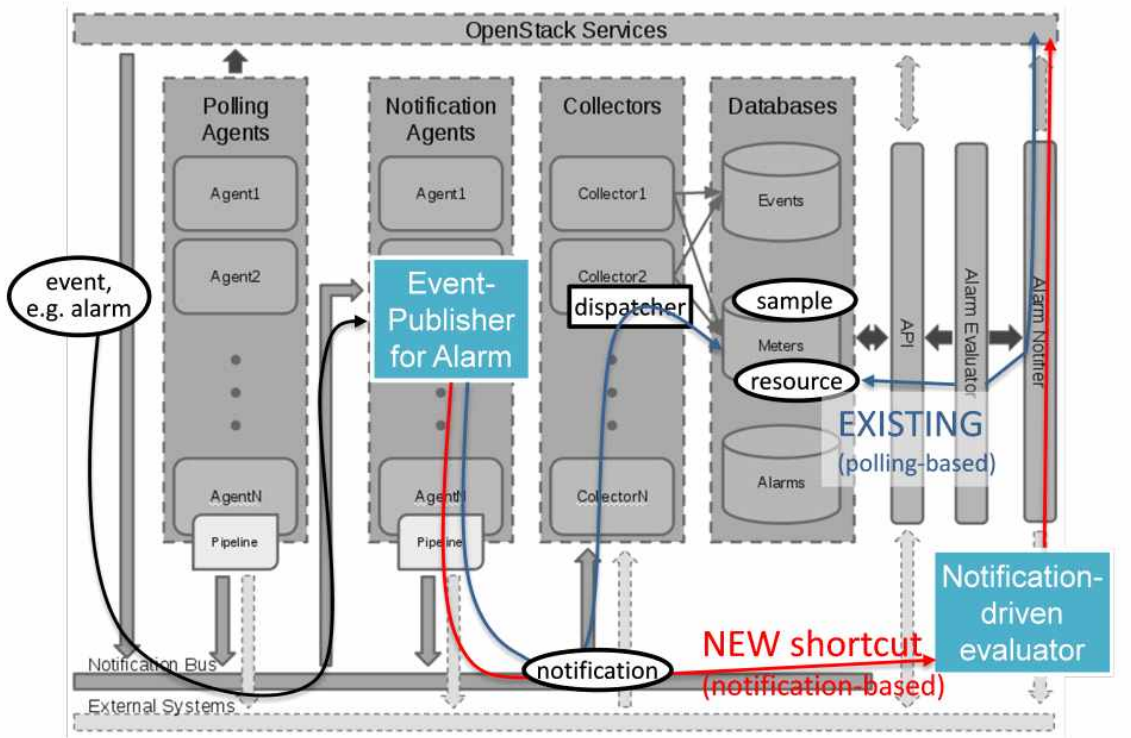

그림 8. Implementation plan in Ceilometer architecture

그림 8은 Ceilometer 프로젝트의 내부적인 동작을 설명하고 있다. 여러 OpenStack 서비스(Nova, Neutron, Cinder)로부터 event가 Notification Bus로 전달된다. Notification agent는 해당 이벤트를 잡아 Ceilometer 데이터베이스에 저장하는 Collector에 보내게 된다. 현재 Ceilometer의 내부 동작을 살펴보면 Alarm Evaluator 컴포넌트가 이벤트가 발생했는지 알기 위해 주기적으로 API를 통해 데이터베이스의 정보들을 가져온다. 데이터베이스로부터 가져온 정보와 미리 설정된 알람 설정정보를 가지고 알람임을 측정하여, 설정된 정보에 맞는 알람이 있는 경우 그 메시지를 Alarm Notifier에게 전달하게 된다. Alarm Notifier는 northbound 인터페이스를 통해 Consumer에게 알람메시지를 보낸다.

하지만 Ceilometer가 현재 가지고 있는 기능(polling-based Alarm Evaluator)만으로는 Doctor가 요구하는 1초 이내로 northbound 인터페이스를 통해 알림 메시지를 Consumer에게 보내지 못한다. 그렇기 때문에 Doctor프로젝트는 결함 알림메시지를 Consumer에게 전달하는 시간을 줄이기 위해 일부 컴포넌트를 개발하고 수정하였다. 새로운 Notification agent를 개발하여 Collector와 Polling-based Alarm Evaluator 컴포넌트를 거치지 않고 직접적으로 새로운 컴포넌트인 “Notification-driven Alarm Evaluator(NAE)” 보내어 추가적인 지연을 줄였다. NAE는 Ceilometer의 “Alarm Evaluator”와 비슷하지만 알람 메시지를 위해 데이터베이스에서 주기적으로 데이터들을 가져오는 것이 아니라 알림메시지를 통해서 NAE가 동작을 시작하게 되는 차이점을 가지고 있다. Ceilometer의 Alarm 데이터베이스에는 3개의 상태(normal, insufficient data, fired)가 있지만 Doctor프로젝트의 요구사항에 따라 새로운 상태인 “meters"가 추가되어야 한다.

8.2. NFVI Maintenance

NFVI Maintenance 같은 경우에는 Fault management처럼 모니터링을 통해서 시작되는 것이 아니라 Administrator가 Maintenance Request를 northbound 인터페이스를 통해 컨트롤러에게 전달해 Maintenance 동작을 시작하게 된다. 위 Fault management 유즈케이스와 비슷하게 컨트롤러는 notifier에게 maintenance 이벤트를 전달하게 되고 Consumer는 조치를 취한 후 다시 컨트롤러에게 특정 명령을 지시하게 된다. 컨트롤러는 물리적인 노드를 비우기 위해 특정 행동을 취하고 Administrator에게 알려 maintenance와 관련된 일들을 수행하게 된다.

9. Demo scenario

9.1. Doctor Demo Overview

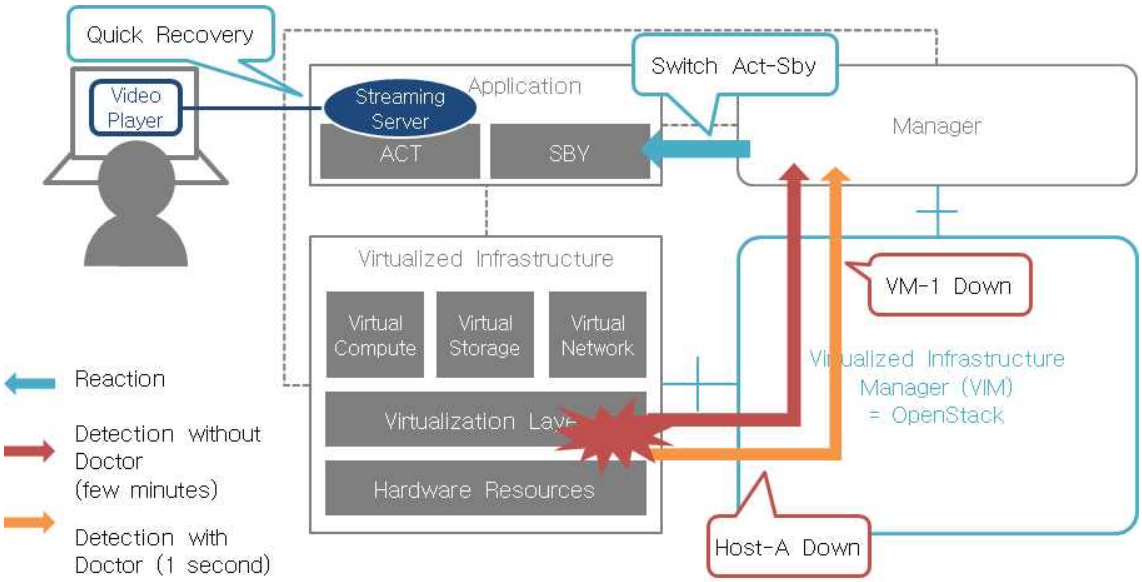

그림 9. Doctor demo overview of Release B

그림 9은 OPNFV Release B 배포에서의 Doctor 프로젝트 데모시나리오이다. 위에서 설명한 Fault management 시나리오에 맞춰 인프라를 모니터링 하고 컴퓨터 노드에 결함이 발생하게 되면 consumer에게 알려 정책에 따른 특정 행동을 VIM에게 전달하게 된다. 지난 7월에 발표한 OPNFV Hackfest의 데모시나리오에서는 모니터링툴을 Zabbix를 사용하였지만 실제 Release B에서는 Zabbix가 아닌 VM이 살아있는지에 대한 ping 테스트를 실행하는 단순한 스크립트를 작성하였다. 위에서 설명한 Inspector 경우에도 Monasca를 사용하는 것이 아니라 역시 단순한 기능을 스크립트로 작성되었다. Monitor.py로부터 VM이 ping이 전달되지 않음을 나타내는 메시지가 오게되면 VM을 실행하고 있는 해당 host ID를 가져와 Doctor 프로젝트에서 개발한 “Mark nova-compute down API"를 사용하여 host의 상태를 ’up’에서 ’down’으로 바꾼다. Ceilometer는 호스트 컴퓨터의 상태가 변화된 것을 감지하여 event를 만들어 Doctor 프로젝트가 개발한 ”Notification-driver evaluator"에게 알리게 된다. Evaluator는 알람을 만들 때 정의했던 notifier의 설정값에 맞춰 Consumer에게 알람을 전달한다. Consumer도 Hackfest에서처럼 오픈스택에서 제공하는 load-balancer를 이용하는 것이 아니라 단순히 알람이 도착했음을 로그파일에 기록하게 된다. 현재 이 문서를 작성하고 있는 시점에서 Doctor 프로젝트의 패키지를 담고있는 배포 프로젝트는 Apex밖에 없고 나머지 배포 프로젝트인 Fuel, Compass, JOID는 아직 지원하고 있지 않아 OPNFV JIRA에 Open Issues에 작성되어 있고 추후에 모든 배포툴에 지원할 것이라고 언급하고 있다.

9.2. Doctor Demo script and configuration file

Doctor 프로젝트를 테스트하기 위해서는 Liberty 이상의 오픈스택을 설치해야하고 Nova, Ceilometer, Aodh 서비스들은 지원을 해야한다.

- Openstack Compute (Nova)
- OpenStack Telemetry (Ceilometer)
- OpenStack Alarming (Aodh)
- Doctor Inspector (simple script for test)
- Doctor Monitor (simple script for test)

해당 환경을 구성하면 git 명령어를 통해 Doctor 프로젝트에서 작성한 스크립트 파일들을 가져온다.

```
git clone https://gerrit.opnfv.org/gerrit/doctor -b stable/brahmaputra
cd doctor/tests
CONSUMER_PORT=12346
python consumer.py "$CONSUMER_PORT" > consumer.log 2>&1 &
```

위 명령어를 통해 가져온 파일 구성을 보면 다음과 같다.

```
root@controller:~/doctor# ls
docs  INFO  LICENSE  tests
```

위 파일에서 docs 디렉토리에서는 Doctor프로젝트의 문서파일들이 있고 tests 디렉토리에는 데모에 필요한 스크립트 파일(monitor.py, inspector.py, consumer.py 등)들로 구성되어 있다.

```
root@controller:~/doctor/tests# ls
clean.py      inspector.log  monitor.py          run.sh
consumer.log  inspector.py   nova_force_down.py  test.sh
consumer.py   monitor.log    nova_force_down.pyc
```

Doctor 데모 시나리오를 테스트하기 전에 먼저 OpenStack에서 설정해야 하는 요소가 있다. 가장 처음에 해야 하는 것은 알람을 등록하는 것이다. 알람을 생성하기 위한 속성을 살펴보면 --alarm-action은 해당 알람이 발생했을 경우 어떤 동작을 취해야 할지를 결정하고 --repeat-actions은 해당 알람을 주기적으로 발생할지를 선택한다. 그리고 이벤트 타입을 나타내는 --even-type과 -q 속성에서 정의한 요소로 이벤트 메시지를 필터링한다. 이 command로 알람을 등록시키면 터미널창에 알람에 관한 설정정보들을 볼 수 있다. 알람을 등록하지 않으면 Consumer에게 특정 이벤트에 대한 정보를 줄 수 없기 때문에 가장 먼저 선행되어야 한다.

```
ceilometer alarm-event-create --name "$ALARM_NAME" \
    --alarm-action "http://localhost:$CONSUMER_PORT/failure" \
    --description "VM failure" \
    --enabled True \
    --repeat-actions False \
    --severity "moderate" \
    --event-type compute.instance.update \
    -q "traits.state=string::error; traits.instance_id=string::$vm_id"
```

알람을 등록하게 되면 밑에 그림 10과 같이 알람이 등록되어 설정정보에 맞는 이벤트가 발생하게 되면 alarm_actions필드에 정의되어 있는 URL로 몇가지 정보와 함께 알람메시지를 보내게 된다.

Property	Value
alarm_actions	["http://10.0.0.100:12346/failure"]
alarm_id	7ade1426-b2b4-44f6-a2c6-52513819a649
description	VM failure
enabled	True
event_type	compute.instance.update
insufficient_data_actions	[]
name	DOCTOR_ALARM
ok_actions	[]
project_id	faef71fcc1e44257bbbc5cb1c3303d86
query	[{"field": "traits.state", "type": "string", "value": "error", "op": "eq"}, {"field": "traits.instance_id", "type": "string", "value": "7801f5ed-ec87-47c9-8df6-4fd116e6beeb", "op": "eq"}]
repeat_actions	True
severity	moderate
state	insufficient data
type	event
user_id	15b1c331018345598a1e71cfe669d701

그림 10. Ceilometer alarm-event-create

알람을 생성하게 되면 오프스택에서 다른 서비스로부터 받은 이벤트를 처리하는 설정정보를 생성해야 한다. 오픈스택 설치가이드에도 나와있지 않아 configuration 가이드에 가서 정보를 얻어와야 한다. 먼저 /etc/ceilometer/ 디렉토리로 이동해 이벤트를 파이프라이닝 하는 설정정보인 event_pipeline.yaml 파일을 만든다. 해당 파일에서 주의할 점은 모든 이벤트를 전달하는 publishers를 디폴트값인 notifier://가 아닌 Doctor프로젝트가 만든 "Notification-driver evaluator"로 이벤트 정보를 보내기 위해서는 다음 그림 11과 같이 정의해야 한다.

```
---
sources:
    - name: event_source
      events:
          - "*"
      sinks:
          - event_sink
sinks:
    - name: event_sink
      transformers:
      triggers:
      publishers:
          - notifier://?topic=alarm.all
```

그림 11. Ceilometer event-pipeline configuration

그리고 다른 오픈스택 서비스로부터 이벤트를 받으면 Ceilometer가 event를 어떻게 처리해야 할지 정의하는 파일이 필요하다. 이 파일 또한 /etc/ceilometer에 위치하게 된다. 밑에 있는 그림 12가 이벤트를 정의한 event_definitions.yaml파일이다. 이 파일은 이벤트 타입과 오퍼레이터들이 흥미있어 하는 요소들로 구성되어 있다. 오프스택의 configuration 가이드에 있는 파일들은 우선 모든 요소들을 집어넣었고 필요에 따라 수정이 가능하다. (그림 12은 내용이 많아 특정 traits까지 자른 그림임)

```
---
- event_type: compute.instance.*
  traits: &instance_traits
    tenant_id:
      fields: payload.tenant_id
    user_id:
      fields: payload.user_id
    instance_id:
      fields: payload.instance_id
    host:
      fields: publisher_id.`split(., 1, 1)`
    service:
      fields: publisher_id.`split(., 0, -1)`
    memory_mb:
      type: int
      fields: payload.memory_mb
    disk_gb:
      type: int
      fields: payload.disk_gb
```

그림 12. Ceilometer event-definitions configuration

9.2.1. Doctor Monitor

Monitor.py를 실행할 때 필요한 3개의 인자는 다음과 같다.

- 모니터링이 필요한 컴퓨터 호스트 이름
- 모니터링이 필요한 컴퓨터 호스트 IP 주소
- 모니터링 결과를 알릴 Inspector URL

```
def start_loop(self):
    print "start ping to host %(h)s (ip=$(i)s)" % {'h': self.hostname,
                                                  'i': self.ip_addr}
    sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
                         socket.IPPROTO_ICMP)
    sock.settimeout(self.timeout)
    while True:
        try:
            sock.sendto(ICMP_ECHO_MESSAGE, (self.ip_addr, 0))
            data = sock.recv(4096)
        except socket.timeout:
            print "doctor monitor detected at %s" % time.time()
            self.report_error()
            print "ping timeout, quit monitoring..."
            return
        time.sleep(self.interval)
```

그림 13. Ping action of monitor.py

3개의 인자를 받은 Monitor 프로세스는 3개의 인자 중 하나인 컴퓨터 호스트 IP를 이용하여 위 그림 13과 같이 소켓을 만들고 해당 호스트로 ping 테스트를 보내게 된다. ping을 보내는 동안 메시지를 받는 해당 호스트 컴퓨터의 네트워크 인터페이스의 연결을 해제하거나 VM인 경우 메시지를 받는 네트워크 인터페이스를 "ifdown ethx" 명령어로 다운시키면 인자 중 하나인 Inspector의 URL로 에러 메시지를 보내게 된다.

```
def report_error(self):compute.host.down
    payload = {"type": self.event_type, "hostname": self.hostname}
    data = json.dumps(payload)
    headers = {'content-type': 'application/json'}
    requests.post(self.inspector, data=data, headers=headers)
```

그림 14. Error report of monitor.py

Inspector에게 알리는 함수는 위 그림 14에 나와 있는 report_error이다. Inspector에게 컴퓨터 호스트이름을 알리는 이유는 Inspector가 해당 컴퓨터 호스트의 상태를 up에서 down으로 바꾸는 Nova의 API를 부르기 때문이다. 인자로 넘어온 URL로 미리정의된 이벤트 타입인 "comput.host.down"과 호스트 이름을 json 데이터로 덤프시켜 전달한다.

9.2.2. Doctor Inspector

Inspector.py에는 Nova에게 host의 state를 "down"으로 변경하는 API를 사용하기 때문에 Keystone에게 인증을 받고 토큰을 받아야 한다. 인증한 토큰 정보를 가지고 Monitor로부터 받은 메시지를 해석하여 Nova의 Force-down API를 호출하게 된다. 여기서 주의해야 할 점은 오픈스택 환경변수를 주의깊게 설정해야 한다. Doctor 프로젝트는 아직 진행중인 프로젝트이므로 Keystone의 버전 2로 요청을 하지만 Liberty나 Mitaka의 경우에는 버전 3을 사용하기 때문에 코드를 수정해야 하는 경우도 있다. 그렇기 때문에 Doctor 프로젝트의 소스를 stable버전이 아닌 버그를 고치고 있는 master로 받는 것도 하나의 방법이다. 하지만 master 브랜치 또한 완벽하지 않고 Doctor 코드를 테스트할 때에 오픈스택 환경에 맞는 코드인지를 신중히 주의해서 봐야 한다.

```
def __init__(self):
    self.nova = novaclient.Client(self.nova_api_version,
                                  os.environ['OS_USERNAME'],
                                  os.environ['OS_PASSWORD'],
                                  os.environ['OS_TENANT_NAME'],
                                  os.environ['OS_AUTH_URL'],
                                  connection_pool=True)
    # check nova is available
    self.nova.servers.list(detailed=False)

def disable_compute_host(self, hostname):
    opts = {'all_tenants': True, 'host': hostname}
    for server in self.nova.servers.list(detailed=False, search_opts=opts):
        self.nova.servers.reset_state(server, 'error')
    nova_force_down.force_down(hostname)
```

그림 15. Keystone authentication and Force-down API of inspector.py

위 그림 15를 보면 처음에 Inspoector.py를 실행하게 되면 Novaclient를 가지고 Keystone에게 인증을 요청하게 된다. 인증을 받은 이후에는 Nova의 API를 호출할 수 있는데 Doctor 시나리오에서는 Monitor로부터 받은 json데이터에 있는 컴퓨터호스트를 향해 Force-down API를 호출한다.

```
app = Flask(__name__)
inspector = DoctorInspectorSample()

@app.route('/events', methods=['POST'])
def event_posted():
    app.logger.debug('event posted')
    app.logger.debug('inspector = %s' % inspector)
    app.logger.debug('received data = %s' % request.data)
    d = json.loads(request.data)
    hostname = d['hostname']
    event_type = d['type']
    if event_type == 'compute.host.down':
        inspector.disable_compute_host(hostname)
    return "OK"
```

그림 16. Route URL of inspector.py for host-down

Doctor 프로젝트에서 Inspector는 파이썬 웹 프레임워크인 Flask를 사용하여 간단한 웹 어플리케이션을 개발하였다. URL을 해석하여 "/events"가 있을 경우 event_posted 함수를 부르게 되고 Monitor가 준 event_type의 데이터가 compute.host.down이라면 inspector는 Nova API를 불러 컴퓨터 호스트가 오류가 났음을 표시하게 된다.

9.2.3. Doctor Consumer

```
app = Flask(__name__)

@app.route('/failure', methods=['POST'])
def event_posted():
    app.logger.debug('doctor consumer notified at %s' % time.time())
    app.logger.debug('received data = %s' % request.data)
    d = json.loads(request.data)
    return "OK"
```

그림 17. Comsuper.py for host-down

Doctor 프로젝트는 NFV에서 NFVM/NFVO를 담당하는 Consumer는 아주 심플하게 구현을 하였다. Ceilometer에 등록한 알람설정 정보 중 "--alarm-action"에 해당하는 URL이 Consumer의 IP와 포트로 구성된 URL로 작성되면 해당 event에 대해 consumer에게 알람을 보내게 된다. 알람을 받은 Consumer는 알람에 있는 정보를 로그에 출력한다.

10. Conclusion

현재 Doctor프로젝트는 NFV환경에서의 결함을 관리하기 위한 필요한 요구사항들을 정의하고 있고 요구사항을 만족시키기 위해 인프라 환경인 OpenStack에 Blueprint를 제안하고 있다. Doctor 프로젝트는 인프라에서 결함을 찾고 Consumer에게 알리는 시간을 1초이내로 정의하고 있기 때문에, 현재 모니터링 기능을 제공하고 있는 Ceilometer 프로젝트에 몇가지 기능을 추가하고 개발하였다. 또한 Nova 프로젝트에서도 현재 호스트에 대한 상태정보를 즉시 업데이트하지 않고 있기 때문에 인프라 환경의 결함을 빨리 처리할 수 있도록 Nova의 API를 추가하였다. 본 기술문서에서는 Doctor 프로젝트의 요구사항과 구조에 대해서 자세히 소개하였고 Doctor 프로젝트의 데모를 테스트하기 위한 설정과 주의사항도 설명하였다.

Doctor 프로젝트는 2016년 4월에 출시된 Release B에 이 2가지 기능을 가지고 간단한 스크립트를 만들어 결함에 대한 빠른 처리를 데모로 테스트하도록 패키지에 추가하였다. 하지만 기술문서를 적고 있는 현재 OPNFV를 배포하는 Apex 프로젝트에서만 Doctor 프로젝트를 테스트할 수 있고 다른 배포 프로젝트인 Fuel, JOID, compass에서는 지원하지 않고 있다. Doctor 프로젝트를 테스트하기 위한 스크립트 또한 OpenStack 환경설정에 따라 버그가 발생할 수 있기 때문에 주의가 필요하다. 앞으로는 Doctor 프로젝트의 데모 시나리오의 발생되는 버그를 찾고 직접 코드를 작성할 예정이다.

References

[1] ETSI ISG NFV, "Network Functions Virtualization –Introductory White Paper v2.0", Oct. 2013.
[2] M. A. Sharkh et al., "Resource Allocation in a Network-Based Cloud Computing Environment: Design Challenges," IEEE Commun. Mag., vol. 51, no. 11, Nov. 2013, pp. 46–52.
[3] ETSI, Network Functions Virtualisation (NFV): Architectural Framework, Technical Report ETSI GS NFV 002 v1.1.1, Oct. 2013.
[4] OPNFV, "Open platform for nfv": https://www.opnfv.org.
[5] OPNFV, "Fault management": https://wiki.opnfv.org/doctor
[6] OpenStack, "Configuration Reference": http://docs.openstack.org/mitaka/config-reference/
[7] Y. Zaki et al., LTE mobile network virtualization, Mobile Networks and Applications Journal, vol. 16, pp. 424-432, August 2011.
[8] Zabbix, "Enterprise-class Monitoring Solution": http://www.zabbix.com/
[9] OPNFV "Data Collection for Failure Prediction": https://wiki.opnfv.org/display/prediction
[10] OpenStack, "Telemetry Project": https://wiki.openstack.org/wiki/Ceilometer
[11] OpenStack "Monasca Project" : https://wiki.openstack.org/wiki/Monasca