

K-ONE 기술 문서 #7
OPNFV/OpenDaylight SFC에서
Load/Path-Aware 서비스 기능 스케줄러 개발

Document No. K-ONE #7

Version 0.2

Date 2016-04-28

Author(s) 서동은, 이재욱, 백호성

■ 문서의 연혁

버전	날짜	작성자	내용
초안 - 0.1	2016. 04. 22	서동은, 이재욱, 백호성	초안 작성
0.2	2016. 04. 28	서동은, 이재욱, 백호성	내용 추가

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

본 고는 OPNFV/OpenDaylight SFC에서 Load/Path-Aware 서비스 기능 스케줄러에 대한 기술문서이다. 본 고는 다음과 같이 구성된다. 1장은 현재 OpenDaylight 커뮤니티와 OPNFV 커뮤니티에서 개발진행중인 SFC 프로젝트의 개요 및 개발동향에 대해 분석하고, 2장에서는 현재 SFC 프로젝트에서 제공하는 기존의 서비스 기능 스케줄러들 (랜덤 서비스 기능 스케줄러, 라운드 로빈 서비스 기능 스케줄러, 로드 밸런싱 서비스 기능 스케줄러, 최단거리 서비스 기능 스케줄러)에 대해 각각 분석한다. 3장에서는 본 고에서 개발한 부하와 경로 길이를 동시에 고려하는 Load/Path-Aware 서비스 기능 스케줄러에 대한 개요와 구현에 대해 설명한다. 4장에서는 Junit 테스트를 통해 개발한 서비스 기능 스케줄러의 기능 검증을 수행하고, OpenDaylight 기반의 실험 결과를 제시한다.

Contents

K-ONE #7. OPNFV/OpenDaylight SFC에서 Load/Path-Aware 서비스 기능 스케줄러 개발

1. OpenDaylight / OPNFV SFC 개요	7
1.1. OpenDaylight SFC	8
1.1.1. OpenDaylight 개요	8
1.1.2. OpenDaylight SFC 프로젝트 개요	8
1.2. OPNFV SFC	10
1.2.1. OPNFV 개요	11
1.2.2. OPNFV SFC 프로젝트 개요	12
2. 기존 Service Function Scheduler 분석	14
2.1. 랜덤 서비스 기능 스케줄러	15
2.2. 라운드 로빈 서비스 기능 스케줄러	16
2.3. 로드 밸런싱 서비스 기능 스케줄러	19
2.4. 최단 거리 서비스 기능 스케줄러	20
3. Load/Path Aware Service Function Scheduler	22
3.1. Load/Path Aware Service Function Scheduler 개발 배경	22
3.2. Load/Path Aware Service Function Scheduler 동작과정	22
3.3. Load/Path Aware Service Function Scheduler 소스 코드	24
4. Path/Load-aware service function scheduler 테스트	31
4.1. Junit 테스트 소스 코드	31
4.1.1. Service Function(SF) 생성 소스 코드	31
4.1.2. Service Function Forwarder (SFF) 생성 소스 코드	32
4.1.3. CPU Utilization 설정 소스 코드	32
4.2. Junit 테스트	32
4.2.1. 테스트 환경	33
4.2.2. 테스트 결과	35
4.3. OpenDaylight 컨트롤러 연동 테스트	39
4.3.1. 테스트 환경	40
4.3.2. 테스트 결과	42

그림 목차

그림 1. OpenDaylight 구조	8
그림 2. OpenDaylight SFC 논리적 구조	9
그림 3. OpenDaylight SFC의 동작과정	9
그림 4. NFV 프레임워크 [6]	11
그림 5. OPNFV SFC 초기구상도	12
그림 6. 서비스 기능 경로 구성과정	14
그림 7. 서비스 기능 스케줄러 구조	15
그림 8. 랜덤 서비스 기능 스케줄러 예시	16
그림 9. 라운드 로빈 서비스 기능 스케줄러 예시 (1)	17
그림 10. 라운드 로빈 서비스 기능 스케줄러 예시 (2)	18
그림 11. 라운드 로빈 서비스 기능 스케줄러 예시 (3)	19
그림 12. 로드 밸런스 서비스 기능 스케줄러 예시	20
그림 13. 최단 거리 서비스 기능 스케줄러 예시	21
그림 14. Load/Path-Aware ServiceFunction Scheduler	23
그림 15. Load/Path-Aware 스케줄러 소스 코드 구성	25
그림 16. scheduleServiceFunctions 함수 소스 코드	25
그림 17. buildTopologyGraph 함수 소스 코드	27
그림 18. getServiceFunctionByType 함수 소스 코드 I	29
그림 19. getServiceFunctionByType 함수 소스 코드 II	29
그림 20. Junit 테스트 소스 코드 (서비스 기능 인스턴스 생성)	32
그림 21. Junit 테스트 소스 코드 (SFF 생성)	33
그림 22. Junit 테스트 소스 코드 (CPU 사용량 설정)	33
그림 23. Junit 테스트 토폴로지 환경	34
그림 24. Junit 테스트 환경 로그 (SF & SFF 생성)	34
그림 25. Junit 테스트 환경 로그 (SF & SFF 링크 생성)	35
그림 26. Junit 테스트 환경 로그 (SF CPU 사용량 설정)	35
그림 27. MNC Chain 생성 로그	36
그림 28. 거리제한 4일 때, 스케줄러 기능 (Firewall 타입의 SF 선택)	36
그림 29. 거리제한 4일 때, 스케줄러 기능 (DPI 타입의 SF 선택)	37
그림 30. 거리제한 4일 때, 스케줄러 기능 (NAT 타입의 SF 선택)	38
그림 31. 거리제한 4일 때, 스케줄러 테스트 결과	38
그림 32. 거리제한 3일 때, 스케줄러 기능 (Firewall 타입의 SF 선택)	37

그림 33. 거리제한 3일 때, 스케줄러 기능 (DPI 타입의 SF 선택)	38
그림 34. 거리제한 3일 때, 스케줄러 기능 (NAT 타입의 SF 선택)	38
그림 35. 거리제한 3일 때, 스케줄러 테스트 결과	39
그림 36. 거리제한 2일 때, 스케줄러 기능 (Firewall 타입의 SF 선택)	39
그림 37. 거리제한 2일 때, 스케줄러 기능 (DPI 타입의 SF 선택)	40
그림 38. 거리제한 2일 때, 스케줄러 기능 (NAT 타입의 SF 선택)	40
그림 39. 거리제한 2일 때, 스케줄러 테스트 결과	40
그림 40. 테스트 환경	41
그림 41. 테스트 SF 등록	42
그림 42. Karaf 로그 (SF 정보 및 CPU 사용량)	42
그림 43. 테스트 SFF 등록	42
그림 44. 체인 생성 (MNC 체인)	42
그림 45. SFP 생성	43
그림 46. Shortestpath 스케줄러 기능 검증	44
그림 47. Loadbalance 스케줄러 기능 검증	45
그림 48. 거리제한이 2홉인 경우의 Load/Path aware 스케줄러 기능 검증	45
그림 49. 거리제한이 3홉인 경우의 Load/Path-Aware 스케줄러 기능 검증	45
그림 50. 거리제한이 4홉인 경우의 Load/Path-Aware 스케줄러 기능 검증	47

표 목차

표 1. Junit 테스트 환경 (SF CPU 사용량)	34
--------------------------------	----

K-ONE #7. OPNFV/OpenDaylight SFC에서 Load/Path-Aware 서비스 기능 스케줄러 개발

1. OpenDaylight / OPNFV SFC 개요

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 주목받고 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.

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 OpenDaylight의 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 개요

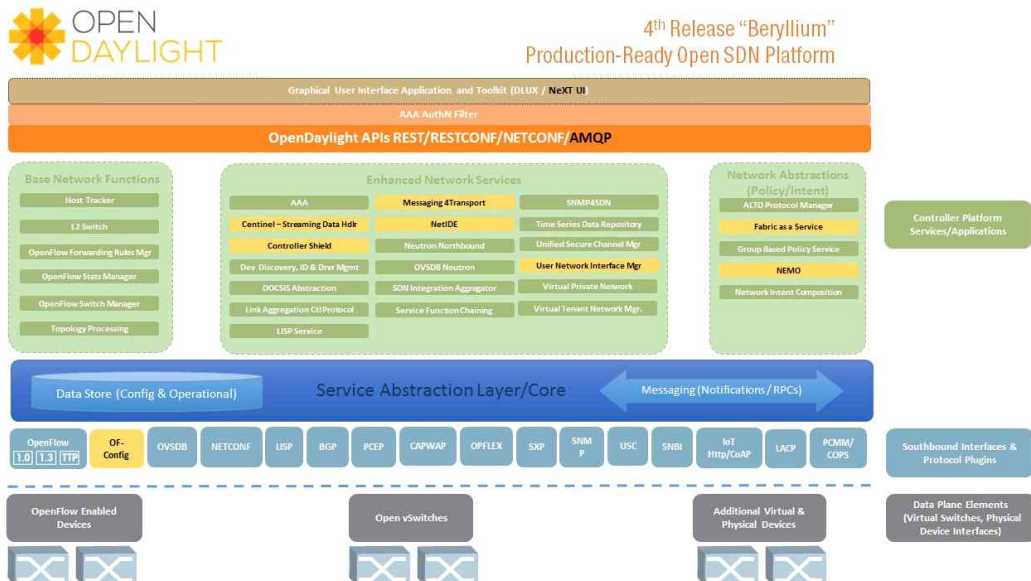


그림 1. OpenDaylight 구조

그림 1은 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 & Orchestration 계층은 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 중단점을 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 개요

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 컨

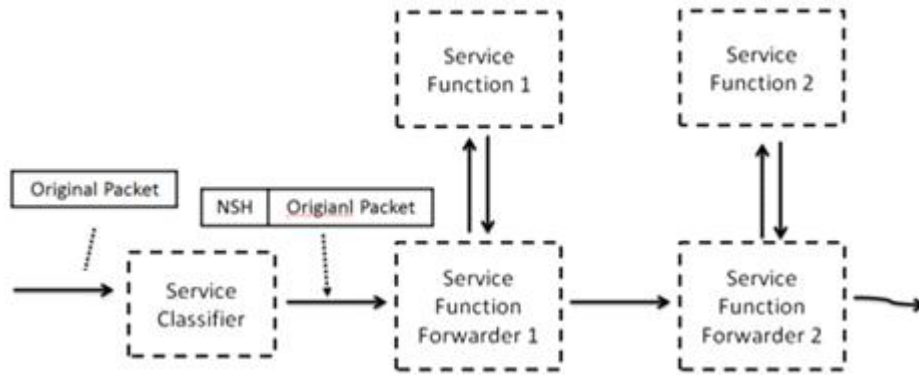


그림 2. OpenDaylight SFC 논리적 구조

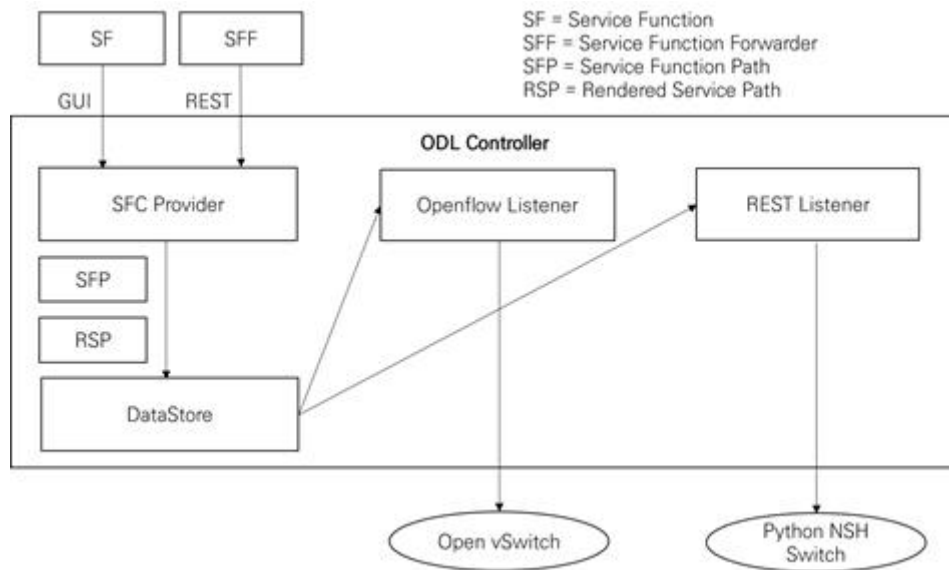


그림 3. OpenDaylight SFC의 동작과정

트롤러의 기능을 나타낸다. 먼저 Opendaylight SFC 웹 UI를 통해 SFP를 구성, 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 플

플랫폼을 개발하고 있다. 그림 4는 NFV기술을 실현하기 위한 프레임워크를 나타내는 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. Orchestrator는 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.

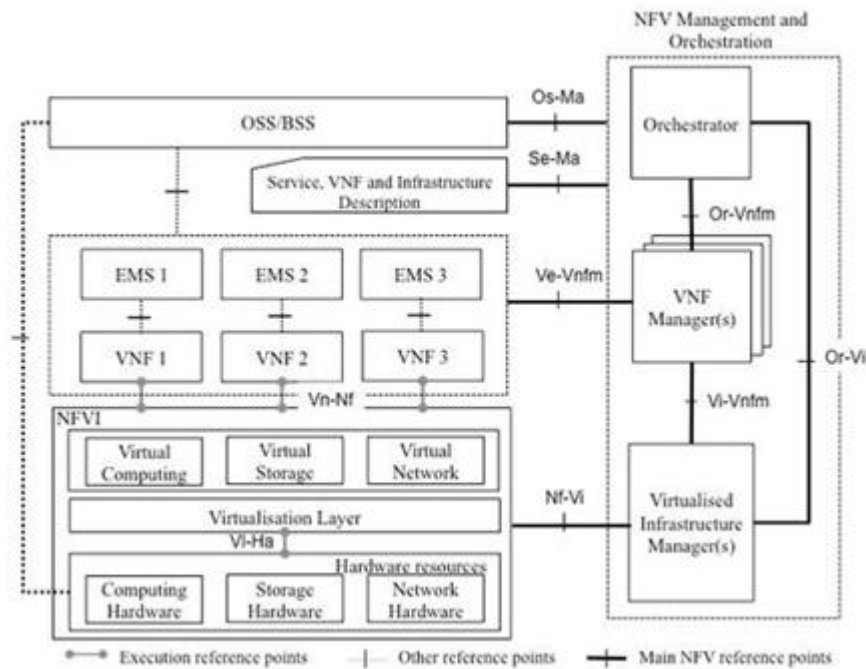


그림 4. NFV 프레임워크 [6]

1.2.1. OPNFV 개요

OPNFV는 네트워크 장비 및 기능을 가상화하기 위한 NFV의 오픈 플랫폼이다. 네트워크 기술이 빠르게 발전함에 따라 고가의 네트워크 장비에 대한 투자비용과 운

용비용 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 개요

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.

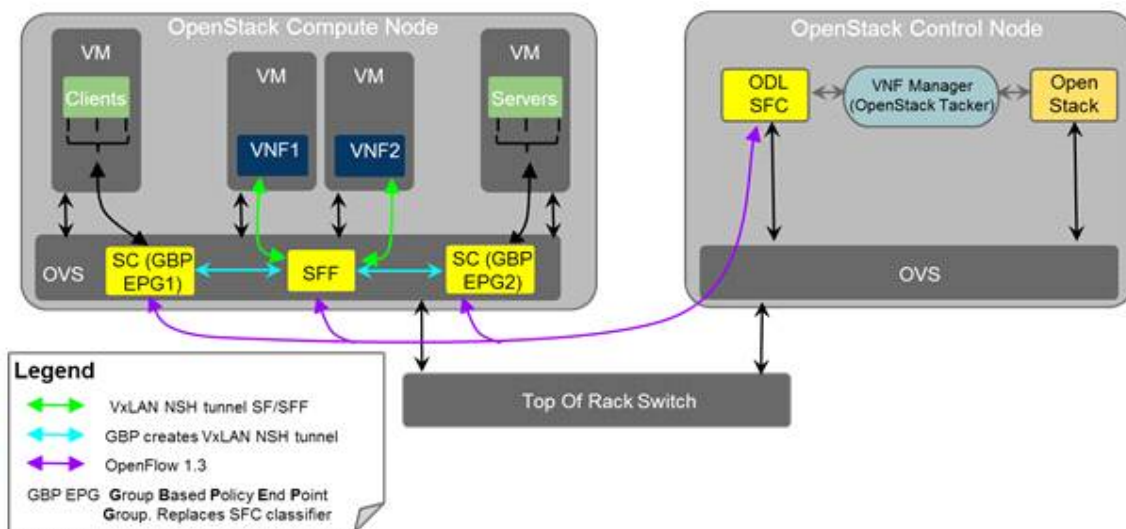


그림 5. OPNFV SFC 초기구상도

그림 5는 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, Opendaylight의 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.

1장에서 OpenDaylight / OPNFV의 SFC 프로젝트에 대해 분석하였다. 2장에서는 OpenDaylight SFC 프로젝트에서 제공되는 서비스 기능 스케줄러에 대해 분석한다.

2. 기존 Service Function Scheduler 분석

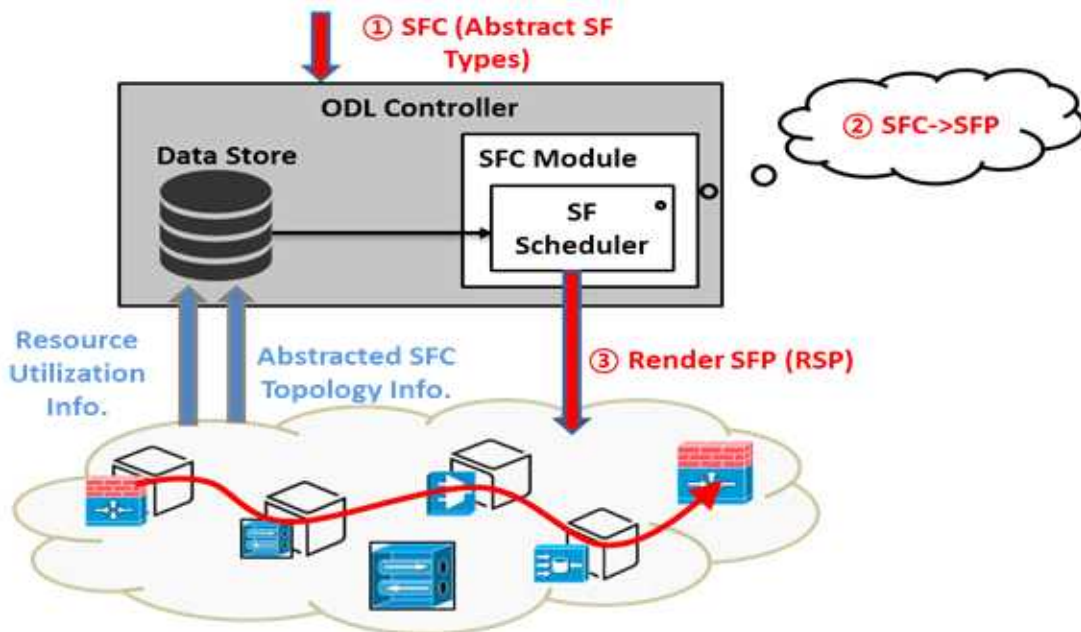


그림 6. 서비스 기능 경로 구성과정

본 장에서는 오픈소스 기반의 SDN 컨트롤러 개발 프로젝트인 OpenDaylight에서 제공하는 Service Function Chaining (SFC)기능에서 서비스 기능 스케줄러가 Service Function Path (SFP)를 결정하는 방식에 대해 알아본다. 서비스 기능 경로를 결정하는 것은 지연 시간, 서비스 기능 인스턴스의 부하 등에 영향을 미치기 때문에 중요한 문제이다. 서비스 기능 경로의 결정 과정은 그림 6과 같다. 플로우가 유입되면, 플로우의 요구사항에 맞게 서비스 기능 체인이 구성된다. 구성된 서비스 기능 체인에 따라 플로우가 실제 지나는 서비스 기능 경로가 구성된다. 이 경우, 선택할 수 있는 같은 타입의 서비스 기능이 여러 개 있을 수 있는데, 서비스 기능 체인을 구성하는 각각의 서비스 기능에 대해 어떤 서비스 기능을 선택하여 서비스 기능 경로를 구성할지 사용자의 목적에 따라 서비스 기능 스케줄러를 통해 선택된다. 이 때 데이터 스토어에 저장된 SFC 도메인 네트워크의 토폴로지와 서비스 기능 인스턴스의 자원에 대한 정보를 활용한다. 즉, 플로우가 지나는 서비스 기능 체인, ServiceFunctionChain 값이 인풋으로 들어오면, 사용자의 목적에 따라, 서비스 기능 스케줄러를 통해 선택된 서비스 기능의 리스트가 결과값으로 나오게 된다. 현재 OpenDaylight 에서 제공되는 서비스 기능 스케줄러는 4가지로, 랜덤 서비스 기능 스케줄러, 라운드 로빈 서비스 기능 스케줄러, 로드 밸런싱 서비스 기능 스케줄러 그리고 최단 거리 서비스 기능 스케줄러가 있다.

서비스 기능 스케줄러의 구조는 그림 7과 같다. 스케줄링 알고리즘을 위한 MD-SAL 데이터 스토어에 서비스 기능 스케줄링 알고리즘 타입에 대한 식별자

(identity) 가 YANG 모델로 정의되어 있다. MD-SAL 데이터 스토어에는 타입, 서비스 기능 인스턴스의 이름, 상태를 포함한 스케줄링 알고리즘에 대한 정보들이 저장되어 있다. RESTConf API는 RESTful 콜을 통해 MD-SAL 데이터 스토어에 저장되어 있는 정보들을 관리하는 API들을 제공하는 역할을 한다. SFC Renderer은 스케줄링 알고리즘 타입에 대한 정보를 받으면, 그에 따라 서비스 기능 인스턴스를 스케줄링 알고리즘에 맞게 구성한다. 다음으로, 현재 OpenDaylight에서 제공되는 4가지의 스케줄러, 랜덤 서비스 기능 스케줄러, 라운드 로빈 서비스 기능 스케줄러, 로드 밸런싱 서비스 기능 스케줄러 그리고 최단 거리 서비스 기능 스케줄러에 대해 분석한다.

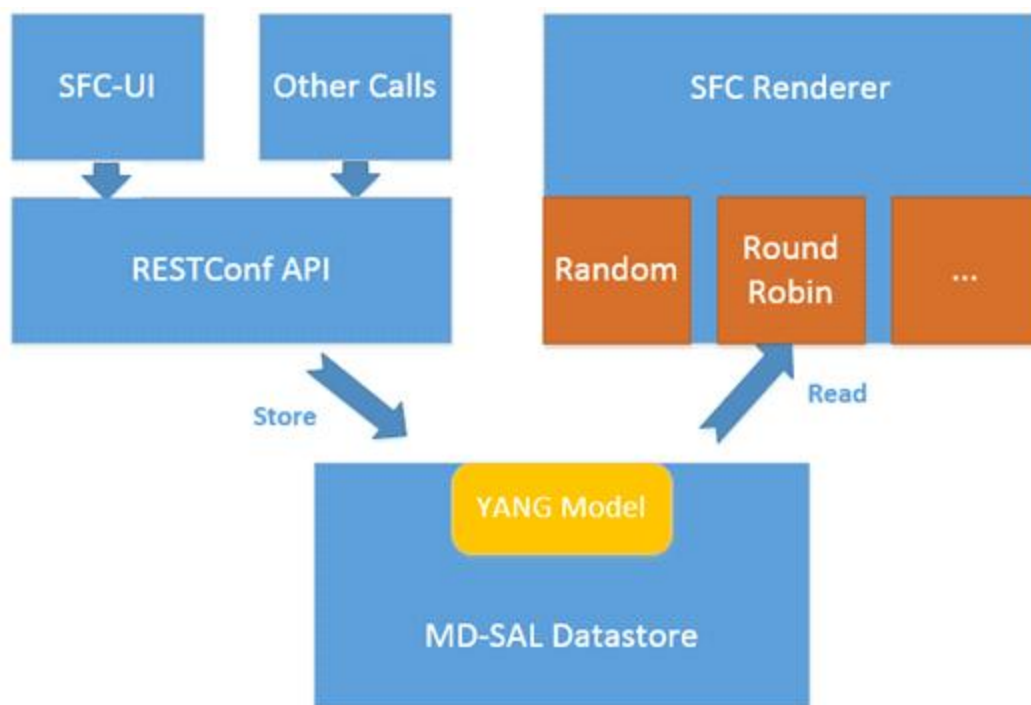


그림 7. 서비스 기능 스케줄러 구조

2.1. 랜덤 서비스 기능 스케줄러

랜덤 서비스 기능 스케줄러는 OpenDaylight에서 제공하는 4개의 서비스 기능 스케줄러 중 하나로, 임의의 서비스 기능에 대해 선택 가능한 모든 서비스 기능 인스턴스들 중 하나를 랜덤하게 선택하는 스케줄러이다. 이 스케줄러는 sfcServiceFunctionList에 받아온다. 주어진 서비스 기능 체인의 순서에 맞게 각각의 서비스 기능에 대해 sftServiceFunctionNameList에 저장된 선택할 수 있는 모든 서비스 기능 인스턴스 중 어떤 서비스 기능 인스턴스를 선택하여 구성할지 스케줄링

한다. 이 때, 랜덤 서비스 기능 스케줄러가 사용되면, 서비스 기능 체인을 구성하는 각각의 서비스 기능에 대해 같은 타입의 여러 서비스 기능 인스턴스 중 하나의 인스턴스를 랜덤하게 선택한다.

랜덤 서비스 기능 스케줄러의 예시는 그림 8과 같다. 플로우의 서비스 기능 체인이 {FW (Firewall) - DPI (Deep Packet Inspection)}으로 주어졌을 때, 각각의 서비스 기능에 대해 서비스 기능 인스턴스를 하나씩 선택한다. 즉, FW의 서비스 기능 인스턴스를 선택할 때, 선택할 수 있는 FW 타입은 FW_1, FW_2, FW_3 으로 3개의 인스턴스가 있다. 랜덤 서비스 기능 스케줄러는 랜덤하게 선택하는 스케줄러이므로, 여러 서비스 기능 인스턴스 중 하나를 랜덤하게 선택한다. 선택된 FW의 인스턴스는 FW_2이다. 다음으로 DPI에 대해서도 서비스 기능 인스턴스를 선택한다. 선택할 수 있는 DPI 타입의 인스턴스는 DPI_1, DPI_2, DPI_3 으로 FW 타입과 마찬가지로 3개의 인스턴스가 있다. 이 경우에도, 랜덤 서비스 기능 스케줄러이기 때문에 임의의 인스턴스를 랜덤하게 선택한다. 선택된 인스턴스는 DPI_3이다. 최종적으로 각각의 선택된 서비스 기능 인스턴스들을 서비스 기능 체인의 순서에 맞게 구성하여 서비스 기능 경로 구성을 한다.

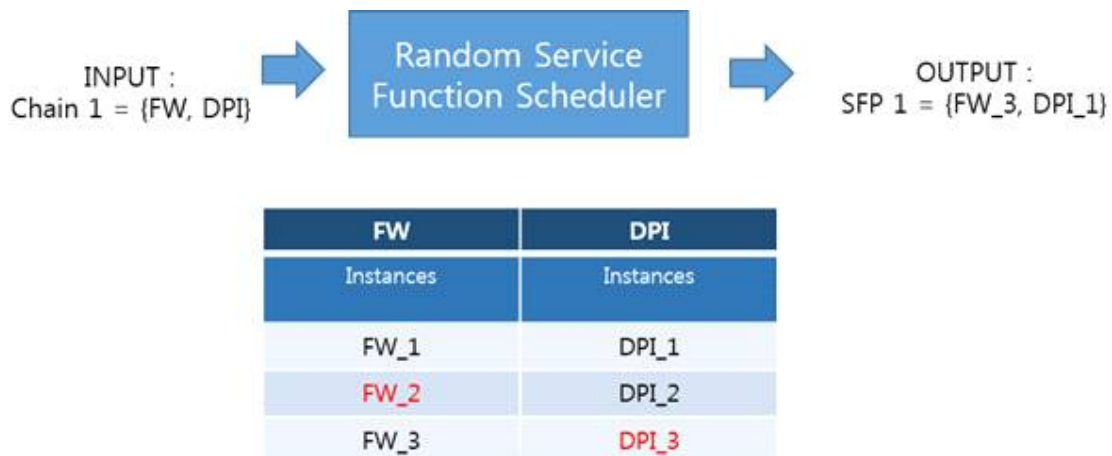


그림 8. 랜덤 서비스 기능 스케줄러 예시

2.2. 라운드 로빈 서비스 기능 스케줄러

라운드 로빈 서비스 기능 스케줄러도 랜덤 서비스 기능 스케줄러와 마찬가지로, 현재 OpenDaylight에서 제공중인 스케줄러 중 하나이다. 이 스케줄러는 랜덤 서비

스 기능 스케줄러와 마찬가지로 임의의 서비스 기능에 대해 선택 가능한 모든 서비스 기능 인스턴스들을 리스트 형태, sftServiceFunctionNameList로 유지한다. 그리고 해당 리스트 상에 포인터를 유지하여 포인터의 위치에 해당하는 서비스 기능 인스턴스를 선택한다. 포인터는 리스트의 맨 앞에 위치한 인스턴스를 시작으로 매 인스턴스가 선택 될 때마다 다음 인스턴스로 이동되어, 최종적으로 매 선택마다 다른 인스턴스가 선택되도록 한다. 즉 리스트 내에 있는 모든 서비스 기능 인스턴스들을 차례대로 하나씩 선택되고 다시 처음 선택한 서비스 기능 인스턴스로 돌아오게 되는 라운드 로빈 방식을 사용한다. 이 스케줄러를 통해 서비스 기능 인스턴스를 균등하게 선택하여 서비스 기능 인스턴스 선택을 분산되게 할 수 있다.

라운드 로빈 서비스 기능 스케줄러의 예시는 그림 9 - 11과 같다. 먼저 그림 9를 보면, 랜덤 서비스 기능 스케줄러와 마찬가지로, 플로우의 서비스 기능 체인이 {FW - DPI} 로 주어졌을 때, 서비스 기능 인스턴스를 선택하여 서비스 기능 경로를 구성한다. 선택할 수 있는 FW 인스턴스의 리스트는 FW_1, FW_2, FW_3이 있다. 처음 선택할 때는 랜덤하게 선택한다. 선택된 FW 서비스 기능 인스턴스는 FW_2이다. 다음으로 DPI의 서비스 기능 인스턴스를 선택할 때도 마찬가지로 처음 선택할 때는 랜덤하게 선택한다. 리스트로 주어진 DPI_1, DPI_2, DPI_3 중 선택된 인스턴스는 DPI_1이다. 이 과정이 끝나면, 선택된 두 인스턴스를 체인의 순서에 맞게 {FW_2, DPI_1}로 서비스 기능 경로를 구성한다.

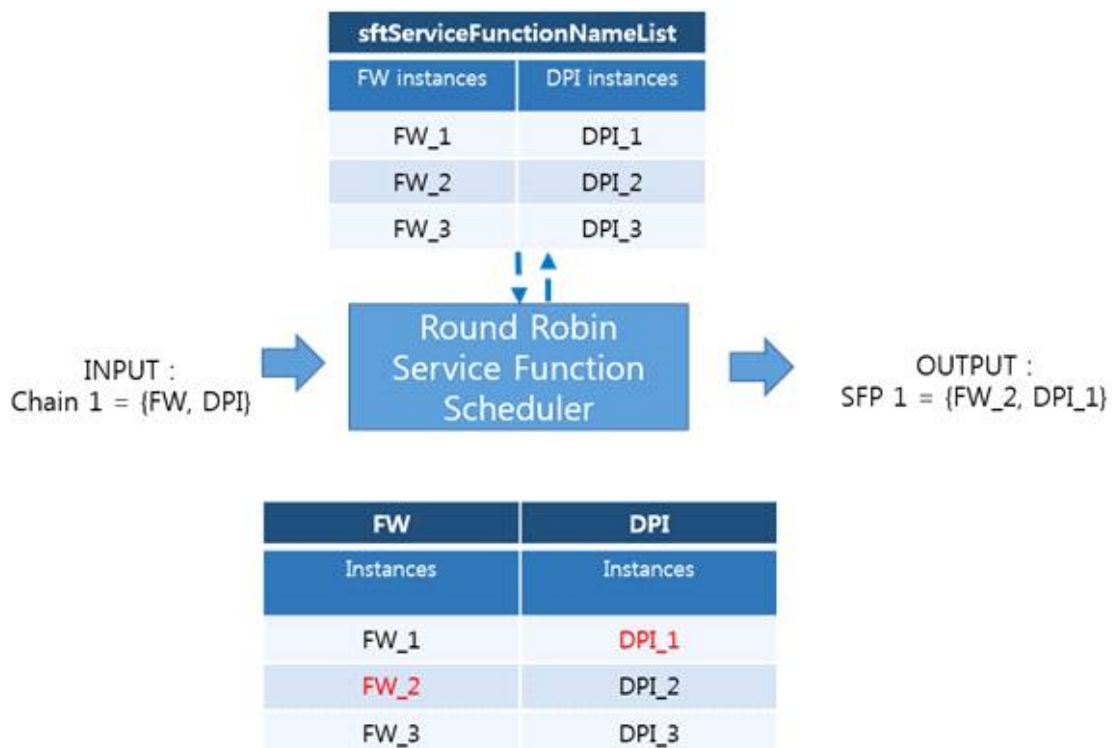


그림 9. 라운드 로빈 서비스 기능 스케줄러 예시 (1)

그림 10은 서비스 기능 경로가 구성되고 나서 다음 서비스 기능 경로를 구성하는

것을 나타낸다. 이 스케줄러는 라운드 로빈 방식으로 서비스 기능 인스턴스를 선택하는 알고리즘이기 때문에, 두번째 서비스 기능 경로를 구성할 때는 이전의 선택한 서비스 기능 인스턴스의 다음 인스턴스가 선택된다. 즉, 그림 11에서 볼 수 있듯이 FW_3, DPI_2가 선택되었다. 그림 11은 3번째로 서비스 기능 경로를 구성하는 것인데, 라운드 로빈 방식으로 선택되기 때문에, 각각의 서비스 기능 인스턴스의 다음 인스턴스가 선택되는 것을 볼 수 있다. 선택된 서비스 기능 인스턴스는 FW_1, DPI_3이다.

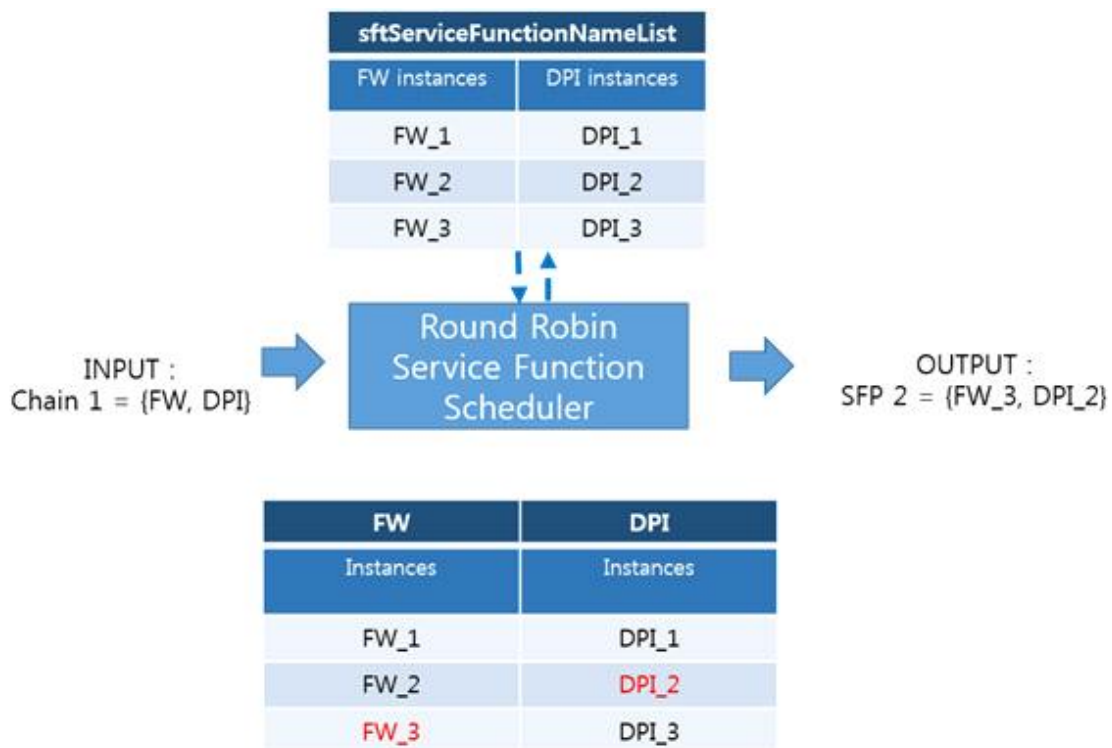


그림 10. 라운드 로빈 서비스 기능 스케줄러 예시 (2)

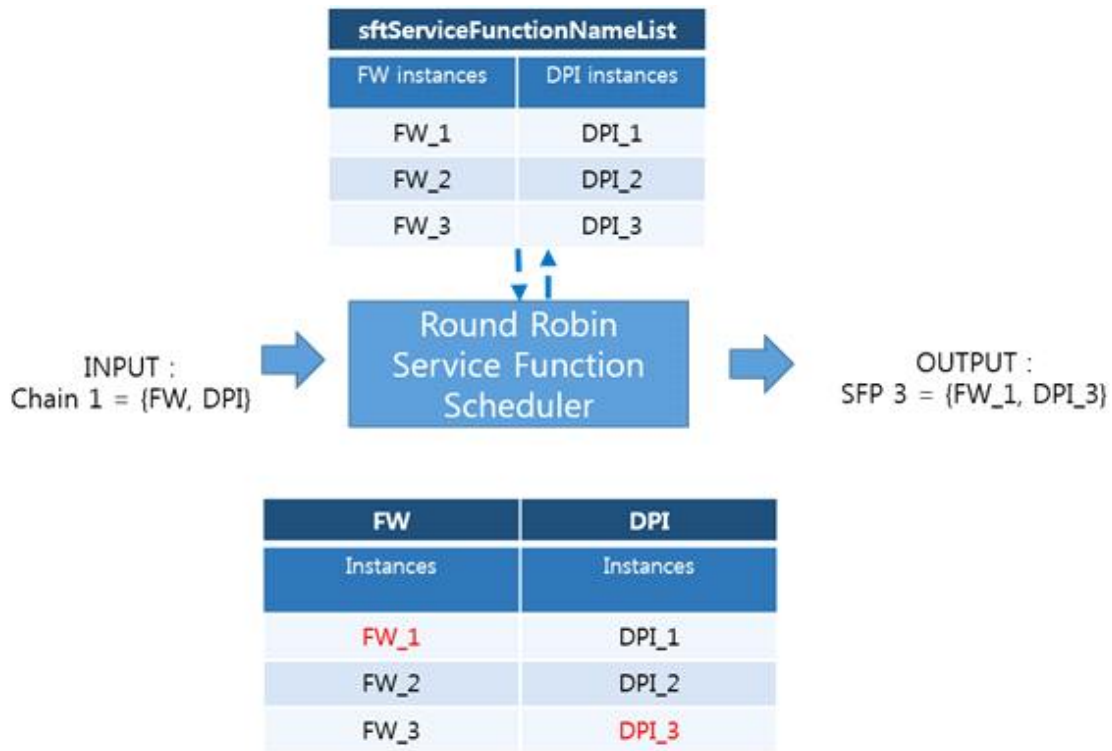


그림 11. 라운드 로빈 서비스 기능 스케줄러 예시 (3)

2.3. 로드 밸런싱 서비스 기능 스케줄러

로드 밸런싱 서비스 기능 스케줄러는 부하를 분산시킬 수 있도록 서비스 기능 인스턴스를 선택하는 스케줄러이다. 앞에서 분석한 랜덤 서비스 기능 스케줄러, 라운드 로빈 서비스 기능 스케줄러와 같이 OpenDaylight에서 제공되는 서비스 기능 스케줄러이다. 이 서비스 기능 스케줄러는 임의의 서비스 기능에 대해 선택 가능한 모든 서비스 기능 인스턴스들의 CPU 사용량을 활용한다. 즉, NETCONF 프로토콜을 통해 받아온 인스턴스들의 CPU 사용량을 데이터 스토어에 저장하고, 스케줄러는 저장된 CPU 사용량 값들을 받아온다. 그리고 선택할 수 있는 같은 타입의 리스트 중 각각의 CPU 사용량을 비교하여 제일 낮은 값을 갖는 서비스 기능 인스턴스를 선택한다. 즉, 해당 스케줄러는 부하 분산을 목적으로 가장 낮은 CPU 사용량을 갖는 서비스 기능 인스턴스를 선택한다.

로드 밸런싱 서비스 기능 스케줄러의 예시는 그림 12와 같다. 앞에 예시와 마찬가지로, 서비스 기능 체인이 {FW - DPI}로 주어진다. FW와 DPI에 대해 선택할 수 있는 각각의 인스턴스들은 리스트 형태로 주어진다. FW 타입의 선택할 수 있는 인스턴스는 FW_1, FW_2, FW_3으로 인스턴스의 CPU 사용량은 각각 100, 90, 80이다. DPI 타입의 선택할 수 있는 인스턴스는 DPI_1, DPI_2, DPI_3으로 인스턴스의 CPU 사용량은 각각 50, 60, 70이다. 이 스케줄러는 분산처리를 목적으로 하는 스케줄러

이기 때문에, 각각의 서비스 기능의 인스턴스를 선택할 때, 현재 사용중인 CPU 사용량이 가장 작은 것을 선택한다. 즉, FW_3가 CPU 사용량이 70으로 선택할 수 있는 FW 타입의 인스턴스 중 가장 작으므로 FW_3가 선택된다. 또한 DPI_1의 CPU 사용량이 50으로 선택할 수 있는 DPI 타입의 인스턴스 중 가장 작으므로 DPI_1이 선택된다. 그리고 각각의 서비스 기능 타입에 대해 인스턴스가 선택되면, 선택된 FW_3과 DPI_1로 서비스 기능 경로를 구성한다.

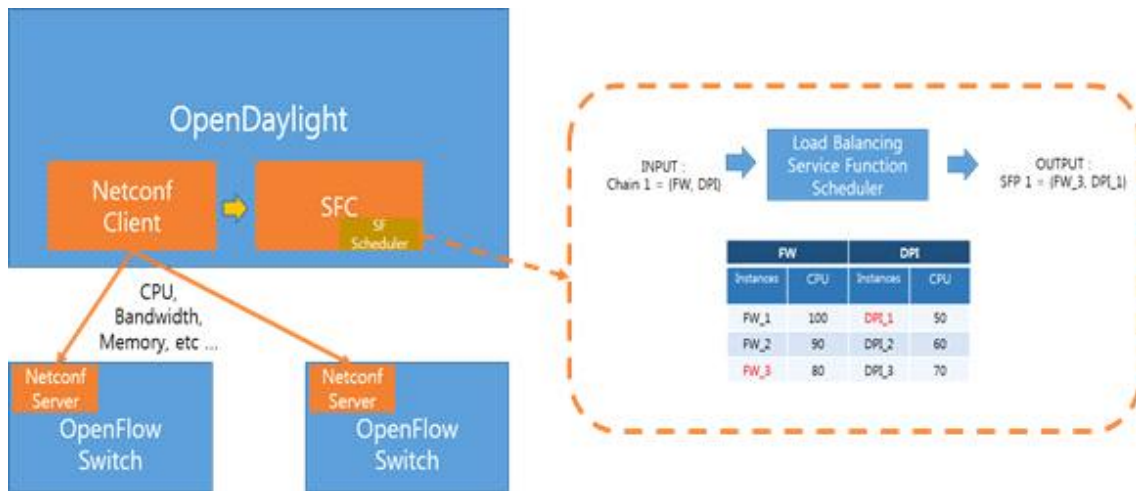


그림 12. 로드 밸런스 서비스 기능 스케줄러 예시

2.4. 최단 거리 서비스 기능 스케줄러

최단 거리 서비스 기능 스케줄러는 서비스 기능 인스턴스를 선택할 때, 서비스 기능 인스턴스간의 홉수가 가장 작은 인스턴스를 선택하는 스케줄러이다. 이 스케줄러도 앞에 스케줄러들과 마찬가지로, OpenDaylight에서 제공되는 스케줄러이다. 최단 거리 서비스 기능 스케줄러의 동작과정은 다음과 같다. 플로우의 서비스 기능 체인이 인풋으로 들어오게 되면, 서비스 기능 체인의 첫번째 서비스 기능에 대한 인스턴스는 랜덤으로 선택한다. 그리고 다음 서비스 기능에 대한 인스턴스를 선택할 시, 이전에 선택된 서비스 기능 인스턴스로부터 모든 선택 가능한 서비스 기능 인스턴스까지의 최소 경로 길이를 Breadth-First Search (BFS)를 통해 시작노드를 기준으로 토폴로지를 Tree구조로 재배열하여 계산하고 그 값들을 비교하여, 최종적으로 가장 낮은 최소 경로 길이를 갖는 인스턴스를 선택한다. 이 때, 노드간의 최단 경로는 getShortestPath function을 통해 리스트 형식으로 받게 된다. BFS는 맹목적 탐색방법의 하나로 시작되는 서비스 기능 인스턴스를 방문 후 시작 인스턴스에 인접한 모든 인스턴스들을 우선 방문하는 방법이다. 즉, 더 이상 방문하지 않은 인스

턴스가 없을 때까지 방문하지 않은 인스턴스에 대해서도 모두 검색을 적용하여 이전의 서비스 기능 인스턴스와의 홉수가 가장 작은 인스턴스를 택한다.

최단 거리 서비스 기능 스케줄러의 예시는 그림 13과 같다. SFC 도메인 네트워크의 토폴로지에 대한 정보를 받아온 정보를 활용한다. 먼저 플로우에 대한 서비스 기능 체인이 {FW - DPI}로 들어오게 되면, 체인의 첫번째 서비스 기능 FW의 인스턴스에 대해서는 랜덤하게 선택한다. 그림 8을 보면, FW_2 가 선택된 것을 알 수 있다. 그리고 체인의 두번째 서비스 기능인 DPI의 인스턴스가 선택될 때, 선택가능한 DPI 타입의 모든 인스턴스와 이전에 선택된 인스턴스 FW_2간의 홉수를 비교하여 홉수가 가장 낮은 인스턴스를 선택한다. 선택할 수 있는 DPI 타입의 인스턴스는 DPI_1, DPI_2, DPI_3 이 있다. 이전에 선택된 FW_2 와의 홉수는 각각 5, 8, 2이다. FW_2 로부터 DPI_3 까지의 홉수가 2로 제일 작은 값을 갖게 되므로, DPI_3 이 선택된 것을 알 수 있다. 앞서서와 마찬가지로, 서비스 기능 체인의 순서에 따라 선택된 두 인스턴스로 서비스 기능 경로를 구성한다.

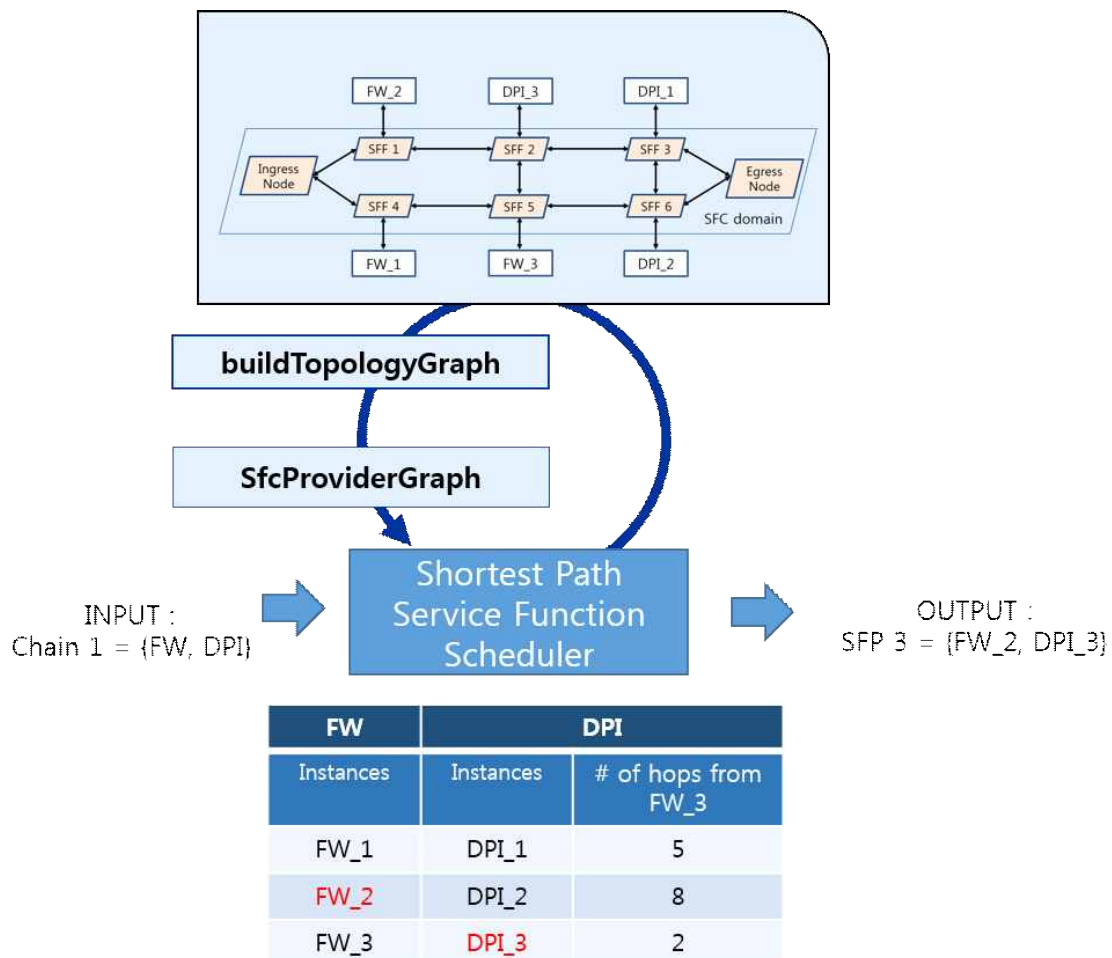


그림 13. 최단 거리 서비스 기능 스케줄러 예시

3. Load/Path-Aware Service Function Scheduler

본 장에서는 서비스 기능간의 부하와 거리를 동시에 고려하는 Load/Path-Aware 스케줄러에 대해 기술한다. 3.1에서는 새로운 스케줄러의 개발 배경, 3.2에서는 해당 스케줄러의 동작과정을 기술한다. 끝으로, 3.3에서는 해당 스케줄러를 구현하기 위한 소스 코드를 기술한다.

3.1. Load/Path-Aware Service Function Scheduler 개발 배경

2장에서 기술한 것과 같이 OpenDaylight Service Function Chain (OpenDaylight-SFC) 프로젝트에서는 4종류의 서비스 기능 스케줄러를 제공한다. 해당 스케줄러들은 각기 다른 목적을 갖고, 체인의 순서에 따른 서비스 기능 인스턴스들을 선택한다. 이미 구현된 스케줄러들의 목적이 단순하며, 이는 사용자들 (네트워크 사업자 등)의 다양한 요구사항을 충족시키지 못하는 문제점들이 존재한다. 가령, 최단거리 서비스 기능 스케줄러의 경우 체인의 순서에 따라 서비스 기능들을 선택할 때, 이전에 선택된 서비스 기능과의 거리가 가장 짧은 인스턴스를 선택한다. 따라서, 해당 스케줄러는 가장 짧은 서비스 기능 경로를 구성하게 되고, 지연시간을 최소화 할 수 있다. 하지만, 해당 스케줄러는 서비스 기능 인스턴스들의 부하를 고려하고 있지 않기 때문에, 특정 인스턴스만 지속적으로 선택하여 과부하가 걸리는 문제점이 존재한다. 이와 반대로 로드밸런스 스케줄러의 경우 체인의 순서에 따라 가장 낮은 부하를 갖는 서비스 기능 인스턴스를 선택한다. 따라서, 해당 스케줄러는 가장 낮은 부하를 갖는 SF들을 선택하기 때문에, 인스턴스 과부하 등의 문제점이 존재하지 않는다. 하지만 SF들 간의 거리를 고려하고 있지 않기 때문에, 해당 스케줄러는 긴 SFP를 구성하게 되는 문제점이 존재할 수 있다. 따라서 최단거리와 로드밸런스 스케줄러들의 문제점들을 해결하면서, 두 스케줄러의 장점을 유지할 수 있는 Load/Path-Aware 스케줄러를 개발하였다. 3.2에서 본 스케줄러의 동작과정을 설명한다.

3.2. Load/Path-Aware Service Function Scheduler 동작과정

3.2에서는 개발한 Load/Path-Aware Service Function Scheduler의 동작과정을 기술한다. 개발한 Load/Path-Aware 스케줄러는 체인의 순서에 맞게 서비스 기능 인스턴스를 선택 시, 인스턴스간의 부하와 인스턴스들간의 거리를 동시에 고려하여 인스턴스를 선택한다. 먼저, 서비스 기능 인스턴스간의 거리를 고려하기 위해 스케

줄러는 거리제한 (L_TH) 값을 미리 정의한다. 거리제한의 의미는 다음과 같다. 이전에 선택된 서비스 기능 인스턴스에서 거리제한 내에 분포된 인스턴스들만 스케줄러가 선택 가능하다. 해당 거리제한 값은 인스턴스간의 홉 수로 정의되며, 컨트롤러 사용자가 정의 할 수 있다. 가령, 컨트롤러 사용자가 L_TH가 2홉으로 정의하게 되면, 이전에 선택된 서비스 기능 인스턴스와 동일한 SFF에 연결된 서비스 기능 인스턴스들만 선택 가능하다. 이렇게 거리제한 값에 의해 선택 가능한 인스턴스들이 선별되면, 스케줄러는 해당 인스턴스들 중 가장 낮은 CPU 사용량을 갖는 인스턴스를 선택하게 된다. 따라서, Load/Path-Aware 스케줄러는 거리제한을 통해 서비스 기능 경로의 길이를 제한하고, CPU 사용량을 통해 과부하를 최소화 하는 서비스 기능 인스턴스를 선택하게 된다.

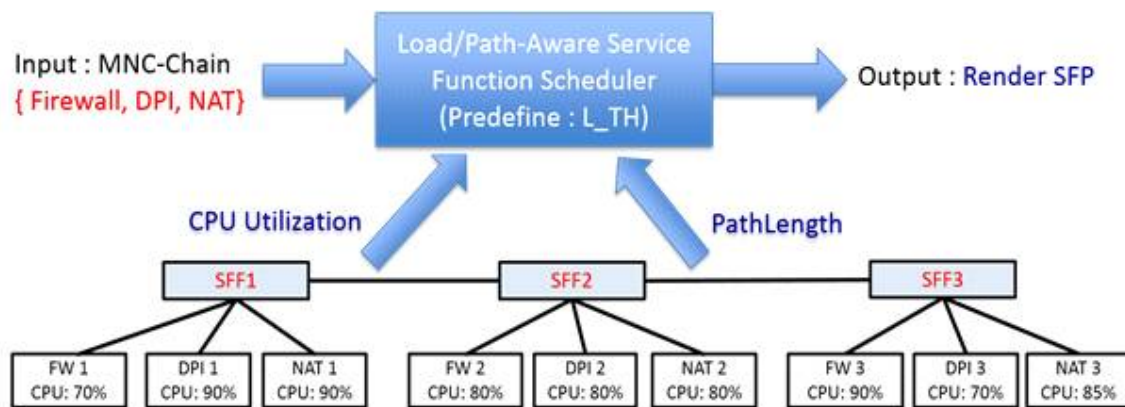


그림 14. Load/Path-Aware ServiceFunction Scheduler

그림 14는 개발한 스케줄러의 개략적인 동작과정을 나타낸다. 본 그림과 같이 Load/Path-Aware 스케줄러는 서비스 기능 인스턴스는 거리제한 값 (L_TH)을 미리 정의하고, 서비스 기능 타입의 순서가 정의된 체인을 입력값으로 받는다. 체인을 입력값으로 받은 스케줄러는 서비스 기능 인스턴스간의 거리와 인스턴스들의 부하를 계산 및 측정 한다. 계산 및 측정한 값들을 통해 본 스케줄러의 목적에 맞게 SF들을 선택하고, 최종적으로 선택된 서비스 기능 인스턴스들을 이어주는 Rendered Service Path (RSP)를 출력한다. 입력된 체인에 순서에 따라 해당 스케줄러가 서비스 기능 인스턴스를 선택하는 과정은 다음과 같다. 그림 14와 같이 체인에 정의된 서비스 기능 타입의 순서가 Firewall, DPI, 그리고 NAT이면 먼저 Firewall 타입의 SF를 선택하고, 그 다음으로 DPI, NAT 순으로 해당 타입의 인스턴스를 선택한다. 스케줄러의 서비스 기능 인스턴스 선택 과정은 첫 번째 타입에 따른 인스턴스 선택과정과 그 이후의 타입에 따른 인스턴스 선택과정으로 나뉜다. 첫 번째 타입에 따른 SF 선택과정의 동작 과정은 다음과 같다. 첫 번째 타입의 인스턴스를 선택 할 때는, 이전 타입이 존재하지 않기 때문에 거리제한의 영향을 받지 않는다. 따라서, 스케줄러는 첫 번째 타입의 모든 서비스 기능 인스턴스들 중 가장 낮은 부하를 갖는 인스턴스를 선택한다. 즉, 그림 14의 예를 들면 스케줄러는 Firewall 타입의 인스턴스 중

가장 낮은 부하를 갖는 FW1을 선택한다. 반면에, 첫 번째 타입이 아닌 타입의 서비스 기능 인스턴스의 선택의 경우, 이전 타입이 존재하고, 해당 타입에 따른 인스턴스가 선택되어 있다. 따라서, 스케줄러는 거리제한을 통해 선택 가능한 SF들을 선별하고, 선별된 SF들 중 가장 낮은 부하를 갖는 인스턴스를 선택한다. 즉, 스케줄러는 첫 번째 타입인 Firewall에 대한 인스턴스를 FW1으로 선택한 경우, 두 번째 타입인 DPI의 인스턴스를 선택 시, 컨트롤러는 FW1과 DPI 타입의 인스턴스간의 최단 거리를 모두 계산한다. FW1과 DPI1의 최단거리는 2홉, DPI2와는 3홉, DPI3과는 4홉이 된다. 계산된 거리와 미리 정의한 거리제한 값을 비교하여, 계산된 거리가 거리제한 값 보다 긴 인스턴스의 경우 제외하고, 작거나 같은 인스턴스들만 고려한다. 가령, 거리제한 값이 2인 경우 선택 가능한 인스턴스는 DPI1이고, 거리제한 값이 3인 경우 선택 가능한 인스턴스는 DPI1, DPI2가 된다. 거리제한 값이 4인 경우 모든 DPI 타입의 인스턴스를 선택 할 수 있다. 이와 같이 선택 가능한 서비스 기능 인스턴스들을 선별하였으면, 해당 인스턴스들 중 가장 낮은 부하를 갖는 인스턴스를 선택한다. 이와 같은 방법으로 마지막 서비스 기능 타입인 NAT에 대한 인스턴스 선택 과정도 선택된 DPI 타입의 인스턴스를 기준으로 동작한다.

3.3. Load/Path-Aware Service Function Scheduler 소스 코드

3.3에서는 구현된 스케줄러의 소스 코드를 설명한다. 구현된 스케줄러의 소스 코드는 그림 15 같이 3가지 함수로 구성되며, 각 함수의 입력과 출력은 다음과 같다. 먼저 `getServiceFunctionByType` 함수는 선택해야 할 서비스 기능 타입 (`serviceFunctionType` 변수), 이전에 선택된 SF의 이름 (`preSfName` 변수) 그리고 서비스 기능 인스턴스와 SFF의 연결성을 나타내는 변수 (`sfcProviderGraph` 객체)가 입력으로 들어간다. 해당 변수들을 입력 받은 함수는 이전에 선택된 인스턴스와의 거리와 선택해야 할 서비스 기능 타입 그리고 인스턴스의 부하를 고려하여 하나는 서비스 기능을 출력한다. 두번째 함수는 `buildTopologyGraph`이다. 해당 함수는 앞서 서술한 서비스 기능 인스턴스와 SFF의 연결성을 나타내는 객체인 초기화된 `sfcProviderGraph`를 입력으로 받고, 컨트롤러에 데이터베이스에 있는 네트워크 토폴로지를 해당 변수에 맞게 업데이트를 한다. 마지막으로 체인과 서비스 인덱스, `ServiceFunctionPath`를 입력으로 받아서 체인에 순서에 맞게 선택된 서비스 기능 인스턴스들을 List 변수로 출력하는 `scheduleServiceFunction` 함수가 있다. 해당 함수들간의 관계는 다음과 같다. 먼저, `scheduleServiceFunction` 함수가 체인과 서비스 인덱스, 그리고 `ServiceFunctionPath`를 입력으로 받아 동작한다. 해당 함수가 동작하게 되면, 가장 먼저 네트워크 토폴로지 추상화를 위해 `buildTopologyGraph`가 동작하여, 네트워크 추상화 객체인 `sfcProviderGraph` 객체가 업데이트가 된다. 그 후에, 입력으로 받은 체인의 서비스 기능 타입의 순서에 따라 `getServiceFunctionByType` 함수를

순차적으로 실행하여 출력 할 SfName 리스트 변수를 구한 후, 해당 리스트를 반환한다. 최종적으로 scheduleServiceFunction 함수가 반환 한 SfName 리스트 값이 스케줄러가 선택 한 SF들이 된다.

```

/**...*/
private SfName getServiceFunctionByType(ServiceFunctionType serviceFunctionType, SfName preSfName,
    SfcProviderGraph sfcProviderGraph) {...}

/**...*/
private void buildTopologyGraph(SfcProviderGraph sfcProviderGraph) {...}

/**...*/
@Override
public List<SfName> scheduleServiceFunctions(ServiceFunctionChain chain, int serviceIndex,
    ServiceFunctionPath sfp) {...}

```

그림 15. Load/Path-Aware 스케줄러 소스 코드 구성

그림 16은 scheduleServiceFunctions 함수 소스 코드의 변수 초기화 부분을 제외한 나머지 부분을 나타낸다. 먼저 해당 함수는 서비스 기능 인스턴스간의 거리를 계산하기 위해 네트워크 토폴로지들의 정보들을 sfcProviderGraph 변수로 변환한다. sfcProviderGraph 변수는 일종의 네트워크 토폴로지 추상화 변수이다. 해당 변수는 SfcProvidrGraph.java 소스 코드에서 정의되어 있으며, 서비스 기능 인스턴스와 SFF들을 한 종류의 노드들로, 서비스 기능 인스턴스와 SFF들간의 연결들을 엣지로 고려하여 네트워크 토폴로지를 트리 형태의 변수들로 저장한다. 네트워크 토폴로지 추상화 변수인 sfcProviderGrpah는 buildTopologyGrpah를 통해 업데이트 된다. 토폴로지 추상화 변수가 업데이트 된 후에, 입력으로 받은 체인의 순서대로 본 스케줄러의 목적에 맞는 인스턴스를 선택한다. sfcServiceFunctionList 변수는 입력으로 받은 체인에 정의된 타입을 순서대로 정의한 변수이고, sfsServiceFunction은 타입을 나타내는 변수이다. 즉, 반복문을 통해서 체인에 정의된 타입들의 인스턴스를 순차적으로 결정하게 된다. 반복문내에서 스케줄러의 목적에 맞게 선택하는 구문은

sfName = getServiceFunctionByType(serviceFunctionType, preSfName, sfcProviderGraph)가 된다. 해당 함수는 반복문에 의해 축출된 servieFunctionType과 우선적으로 업데이트 되었던 토폴로지 추상화 변수인 sfcProviderGraph, 끝으로 이전에 선택된 인스턴스의 이름 preSfName을 입력 받고, 선택 된 인스턴스의 이름인 sfName 변수를 반환한다. 반환된 sfName 변수는 다음 타입에 인스턴스를 선택할 시 필요한 이전 인스턴스의 이름으로 업데이트 되며, 또한 본 함수의 최종 반환 변수인 sfNameList에 순차적으로 저장된다.

그림 17은 토폴로지 추상화 변수인 sfcProviderGraph를 업데이트 하는 buildTopologyGraph 함수 소스 코드 중 변수 초기화 부분을 제외한 나머지 소스 코

```

/*
 * Build topology graph for all the nodes,
 * including every ServiceFunction and ServiceFunctionForwarder
 */
buildTopologyGraph(sfcProviderGraph);

/*
 * Select a SF instance for each ServiceFunction type in sfcServiceFunctionList.
 */
for (SfcServiceFunction sfcServiceFunction : sfcServiceFunctionList) {
    LOG.debug("ServiceFunction name: {}", sfcServiceFunction.getName());
    SfName hopSf = sfpMapping.get(index++);
    if (hopSf != null) {
        sfNameList.add(hopSf);
        continue;
    }

    ServiceFunctionType serviceFunctionType =
        SfcProviderServiceTypeAPI.readServiceFunctionType(sfcServiceFunction.getType());
    if (serviceFunctionType != null) {
        List<SftServiceFunctionName> sftServiceFunctionNameList =
            serviceFunctionType.getSftServiceFunctionName();
        if (!sftServiceFunctionNameList.isEmpty()) {
            sfName = getServiceFunctionByType(serviceFunctionType, preSfName, sfcProviderGraph);
            if (sfName != null) {
                sfNameList.add(sfName);
                preSfName = sfName;
                LOG.debug("Next Service Function: {}", sfName);
            } else {
                LOG.error("Couldn't find a reachable SF for ServiceFunctionType: {}",
                    sfcServiceFunction.getType());
                return null;
            }
        } else {
            LOG.debug("No {} Service Function instance", sfcServiceFunction.getName());
            return null;
        }
    } else {
        LOG.debug("No {} Service Function type", sfcServiceFunction.getName());
        return null;
    }
}
return sfNameList;

```

그림 16. scheduleServiceFunctions 함수 소스 코드

드 부분이다. 해당 함수는 먼저 서비스 기능 인스턴스를 sfcProviderGraph 변수에 노드로 업데이트를 한다. 컨트롤러에 등록 된 모든 SF들을 readAllServiceFunction() 함수를 통해 읽어 온 후, 리스트 변수 (serviceFunctionList)에 저장한다. 해당 리스트에 저장 된 서비스 기능 인스턴스들을 반복문을 통해 순차적으로 sfcProviderGraph 객체에 정의된 addNode()함수를 통해 해당 객체에 추가한다. 컨트롤러에 등록 된 모든 서비스 기능 인스턴스들에 대한 노드 등록이 완료되었으면, 토폴로지에 존재하는 SFF들을 노드로 등록한다. sfcProviderGraph객체에서는 서비스 기능 인스턴스와

SFF를 구분하지 않고, 하나의 노드로 저장한다. SFF들을 sfcProviderGarph 객체에 노드로 저장 할 때, SFF에 연결된 링크도 엣지로 저장한다. readAllServiceFunctionForwarders() 함수를 통해 토폴로지에 분포 된 SFF들 읽어 온 후에, 리스트 변수 (serviceFunctionForwarderList)에 저장한다. 반복문을 통해 리스트에 저장된 SFF을 순차적으로 불러오고, 해당 SFF를 sfcProviderGraph 객체에 addNode 함수를 통해 추가한다. SFF를 객체에 노드를 추가 한 후에 해당 SFF와 연결된 SF와 SFF 들의 링크를 엣지로 저장한다. SFF 변수 내에는 해당 SFF와 연결된 SF들의 변수 (ServiceFunctionDictionary) 그리고 연결 된 SFF들의 변수 (connectedSffDictionary)들이 각 각 리스트 변수로 정의되어 있기 때문에, 두 변수를 이용하여 sfcProviderGraph 객체에 해당 SFF에 대한 엣지를 추가한다. 먼저 해당 SFF와 연결 된 서비스 기능 인스턴스들의 정보인 ServiceFunctionDictionary 리스트 변수를 getServiceFunctionDictionary()를 통해 읽어 온다. 반복문을 통해 해당 리스트 변수에서 SFF와 연결된 서비스 기능 인스턴스들의 정보를 순차적으로 읽고, addEdge(sfName, sffName) 함수를 통해 객체에 엣지를 추가 시킨다. 한 SFF에 대한 서비스 기능 인스턴스들의 엣지 추가 완료 후에, 해당 SFF와 SFF들간의 엣지를 추가한다. 먼저 해당 SFF와 연결 된 SFF들의 정보를 저장한 connectedSffDictionaryList 리스트 변수를 getConnectedSffDictionary()를 통해 읽어 온 후, 반복문을 통해 연결된 SFF들의 정보를 순차적으로 읽어온다. addEdge(sffName, sffName) 함수를 통해 객체에 연결된 SFF에 대한 엣지 추가시킨다. 본 buildTopologyGraph 함수를 통해 분산적으로 저장되었던 토폴로지 정보들을 하나의 객체 인 sfcProviderGraph 객체 내에 업데이트 한다. 해당 객체를 통해 각 노드 간의 최단경로 및 홉 수를 쉽게 계산 할 수 있다.

그림 18과 그림 19는 하나의 타입에 따른 스케줄러의 목적에 맞는 인스턴스를 선택한 getServiceFunctionByType 함수 소스 코드를 나타낸다. 함수 소스 코드는 첫 번째 타입의 SF 선택을 위한 부분과 그 이후의 타입의 인스턴스 선택을 위한 부분으로 나뉜다. 그림 18은 체인에 정의된 타입들 중 첫 번째 타입의 인스턴스 선택을 위한 소스 코드를 나타낸다. getServiceFunctionByType은 앞서 기술한 sfcProviderGraph객체와 타입, 그리고 이전에 선택 된 인스턴스를 입력으로 받아서 동작한다. 체인에서 첫 번째 타입의 경우 이전에 선택 된 인스턴스가 없기 때문에 이전에 선택 된 SF 값을 NULL로 받는다. 이전에 선택 된 서비스 기능 인스턴스 (preSfName)가 NULL인 경우 스케줄러는 해당 타입의 인스턴스 중 가장 낮은 CPU 사용량을 갖는 인스턴스를 선택한다. 해당 동작과정을 나타내는 소스 코드 설명은 다음과 같다. 먼저 입력으로 받은 타입에 대한 서비스 기능 인스턴스들의 리스트 변수인 sftServiceFunctionList 변수에서 반복문을 통해 순차적으로 SF를 읽어 온다. 그 후에 읽어 온 인스턴스 (sfName)의 CPU 사용량을 readServiceFunctionDescriptionMonitor(sfName)을 통해 컨트롤러 데이터베이스로부터 반환 받는다. 반환 받은 CPU 사용량 (curCPUUtilization)과 이전까지 읽어 온 인


```

/* Add all the ServiceFunction nodes */
ServiceFunctions sfs = SfcProviderServiceFunctionAPI.readAllServiceFunctions();
List<ServiceFunction> serviceFunctionList = sfs.getServiceFunction();
for (ServiceFunction serviceFunction : serviceFunctionList) {
    sfName = serviceFunction.getName();
    sfcProviderGraph.addNode(sfName.getValue());
    LOG.debug("Add ServiceFunction: {}", sfName);
}

ServiceFunctionForwarders sffs = SfcProviderServiceForwarderAPI.readAllServiceFunctionForwarders();
List<ServiceFunctionForwarder> serviceFunctionForwarderList = sffs.getServiceFunctionForwarder();

/* Add edges and node for every ServiceFunctionForwarder */
for (ServiceFunctionForwarder serviceFunctionForwarder : serviceFunctionForwarderList) {
    /* Add ServiceFunctionForwarder node */
    sffName = serviceFunctionForwarder.getName();
    sfcProviderGraph.addNode(sffName.getValue());
    LOG.debug("Add ServiceFunctionForwarder: {}", sffName);

    List<ServiceFunctionDictionary> serviceFunctionDictionaryList =
        serviceFunctionForwarder.getServiceFunctionDictionary();

    /*
     * Add edge for every ServiceFunction attached
     * to serviceFunctionForwarder
     */
    for (ServiceFunctionDictionary serviceFunctionDictionary : serviceFunctionDictionaryList) {
        sfName = serviceFunctionDictionary.getName();
        sfcProviderGraph.addEdge(sfName.getValue(), sffName.getValue());
        LOG.debug("Add SF-to-SFF edge: {} => {}", sfName, sffName);
    }

    List<ConnectedSffDictionary> connectedSffDictionaryList =
        serviceFunctionForwarder.getConnectedSffDictionary();

    /*
     * Add edge for every ServiceFunctionForwarder connected
     * to serviceFunctionForwarder
     */
    for (ConnectedSffDictionary connectedSffDictionary : connectedSffDictionaryList) {
        toSffName = connectedSffDictionary.getName();
        sfcProviderGraph.addEdge(sffName.getValue(), toSffName.getValue());
        LOG.debug("Add SFF-to-SFF edge: {} => {}", sffName, toSffName);
    }
}

```

그림 17. buildTopologyGraph 함수 소스 코드

스턴스들의 CPU 사용량 (preCPUUtilization)중 가장 낮은 CPU 사용량과 비교하여,

반환 받은 CPU 사용량이 더 작으면 해당 CPU 사용량을 가장 작은 CPU 사용량으로 업데이트 한다. 최종적으로 입력으로 받은 타입을 갖는 인스턴스 중 가장 낮은 CPU 사용량을 갖는 인스턴스를 반환한다.

그림 19는 첫 번째로 정의 된 타입이 아닌 타입들의 인스턴스를 선택하는 과정을 구현한 소스 코드이다. 우선 입력 받은 타입에 대한 모든 서비스 기능 인스턴스가 저장된 리스트 변수 (sftServiceFunctionNameList)에서 반복문을 통해 순차적으로 인스턴스의 이름 (curSfName)을 얻는다. 입력으로 받은 sfcProviderGraph 객체에 정의된 getShortestPath() 함수를 통해, 추출된 SF와 입력으로 받은 이전의 인스턴스들간의 최단 경로를 구한다. 이때 반환되는 최단 경로는 두 서비스 기능 인스턴스간의 최단 경로에 존재하는 인스턴스들로 구성된 리스트 변수이다. 따라서, 두 인스턴스간의 최단경로의 홉 수는 리스트 크기에서 일을 뺀 값이 된다. 최단경로 홉 수를 구한 후에 해당 인스턴스의 CPU 사용량을 readServiceFunctionDescriptionMonitor()를 통해 컨트롤러의 데이터베이스로부터 읽어온다. 이전까지 얻은 CPU 사용량 중 가장 작은 값과 읽어온 CPU 사용량을 비교하고, 또한 이전에 선택된 인스턴스와의 최단경로 홉 수가 미리 정해진 거리제한 (L_TH)값과 비교하여 거리제한에 만족하면서 최소의 CPU 사용량을 갖는 인스턴스를 반환한다.

지금까지 새로 구현 된 Load/Path aware 스케줄러의 소스 코드를 기술하였다. 다음 4장에서는 구현 된 스케줄러의 기능을 검증하기 위해 실행 된 테스트와 결과에 대해 기술한다.

```

if (preSfName == null) {
    SfName sfName = null;
    SfName sftServiceFunctionName = null;
    java.lang.Long preCPUUtilization = java.lang.Long.MAX_VALUE;

    for (SftServiceFunctionName curSftServiceFunctionName : sftServiceFunctionNameList) {
        sfName = new SfName(curSftServiceFunctionName.getName());

        /* Check next one if ourSftServiceFunctionName doesn't exist */
        ServiceFunction serviceFunction = SfcProviderServiceFunctionAPI.readServiceFunction(sfName);
        if (serviceFunction == null) {
            LOG.error("ServiceFunction {} doesn't exist", sfName);
            continue;
        }

        /* Read ServiceFunctionMonitor information */
        SfcSfDescMon sfcSfDescMon = SfcProviderServiceFunctionAPI.readServiceFunctionDescriptionMonitor(sfName);
        if (sfcSfDescMon == null) {
            // TODO As part of typedef refactor not message with SFTs
            sftServiceFunctionName = sfName;
            LOG.error("Read monitor information from Data Store failed! serviceFunction: {}", sfName);
            // Use sfName if no sfcSfDescMon is available
            break;
        }

        java.lang.Long curCPUUtilization =
            sfcSfDescMon.getMonitoringInfo().getResourceUtilization().getCPUUtilization();

        if (preCPUUtilization > curCPUUtilization) {
            preCPUUtilization = curCPUUtilization;
            sftServiceFunctionName = sfName;
        }
    }

    if (sftServiceFunctionName == null) {
        LOG.error("Failed to get one available ServiceFunction for {}", serviceFunctionType.getType());
    }

    SfcProviderTopologyNode firstHopNode;
    sfcProviderTopologyNodeName = sftServiceFunctionName;
    /*
     * XXX noticed that SfcProviderGraph sometimes refers to SFFs as well so leaving
     * that alone for now until a general discussion
     * about Schedulers can be had.
     */
    firstHopNode = sfcProviderGraph.getNode(sfcProviderTopologyNodeName.getValue());
    LOG.debug("The first ServiceFunction name: {}", sfcProviderTopologyNodeName);
    return sfcProviderTopologyNodeName; // The first hop
}

```

그림 18. getServiceFunctionByType 함수 소스 코드 1


```

/* Select the next SF instance whose CPU Utilization is minimum in a refined candidate set (i.e., sftServiceFunctionNameList) */
for (SftServiceFunctionName sftServiceFunctionName : sftServiceFunctionNameList) {
    SfName curSfName = new SfName(sftServiceFunctionName.getName());
    SfcProviderTopologyNode curSfcProviderTopologyNode = sfcProviderGraph.getNode(curSfName.getValue());
    if (curSfcProviderTopologyNode == null) {
        // curSfName doesn't exist in sfcProviderGraph, so skip it
        continue;
    }

    /* Calculate Length from the preSfName to curSfName */
    List<SfcProviderTopologyNode> sfcProviderTopologyNodeList =
        sfcProviderGraph.getShortestPath(preSfName.getValue(), curSfName.getValue());
    SIZE = sfcProviderTopologyNodeList.size();
    length = SIZE-1;
    LOG.debug("Shortest path length between {} and {} : {}", preSfName, curSfName, length);

    if (length <= 1) {
        LOG.debug("No path from {} to {}", preSfName, curSfName);
        continue;
    }
    ServiceFunction serviceFunction = SfcProviderServiceFunctionAPI.readServiceFunction(curSfName);
    if (serviceFunction == null) {
        LOG.error("ServiceFunction {} doesn't exist", curSfName);
        continue;
    }

    /* Read ServiceFunctionMonitor information of curSfName */
    SfcSfDescMon sfcSfDescMon = SfcProviderServiceFunctionAPI.readServiceFunctionDescriptionMonitor(curSfName);

    java.lang.Long curCPUUtilization =
        sfcSfDescMon.getMonitoringInfo().getResourceUtilization().getCPUUtilization();

    if (preCPUUtilization > curCPUUtilization && length <= L_TH) {
        preCPUUtilization = curCPUUtilization;
        sfcProviderTopologyNodeName = curSfName;
    }
}

return sfcProviderTopologyNodeName;

```

그림 19. getServiceFunctionByType 함수 소스 코드 II

4. Path/Load-aware service function scheduler 테스트

본 장에서는 3장에서 기술한 Path/Load-aware service function scheduler의 기능 검증을 위한 테스트와 테스트 결과를 기술한다. 기능 검증을 위한 테스트는 Java 소스 코드 테스트인 Junit 테스트와 컨트롤러를 통한 테스트를 수행하였으며, 해당 테스트들의 결과를 통해 개발한 스케줄러의 기능을 검증 하였다. 4.1에서는 Junit 테스트에 필요한 핵심 소스 코드들을 설명하고, 4.2에서는 Junit 테스트 결과를 기술한다. 또한, 4.3에서는 2장에서 기술한 스케줄러 중 2개의 스케줄러 (최단거리 서비스 기능 스케줄러와 로드밸런스 서비스 기능 스케줄러)와 개발한 스케줄러의 기능을 컨트롤러를 통해 비교한 결과를 기술한다.

4.1. Junit 테스트 소스 코드

4.1에서는 JAVA 코드 테스트인 Junit 테스트 결과를 통해 3장에서 기술한 Load/Path-Aware service function scheduler의 기능을 확인한다.

Junit 테스트 코드는 테스트 클래스의 인스턴스 및 객체를 초기화 하는 코드와 테스트를 실행 하는 코드로 구성된다. 인스턴스 및 객체를 초기화 하는 코드를 통해 테스트 할 환경이 구현되며, 테스트를 실행하는 코드를 통해 구현된 테스트환경에서 테스트를 진행한다. 인스턴스 및 객체를 초기화는 @Before 어노테이션이 선언된 메서드를 통해 구현되며, 본 테스트를 위해 작성된 메서드는 다음과 같이 구성된다.

4.1.1. Service Function(SF) 생성 소스 코드

서비스 기능 인스턴스를 생성하기 위해서는 인스턴스의 이름 (SF_NAME), 타입 (SF_TYPES), Management IP address (ipMgmtAddr) 와 DataPlane에 대한 정보 (dataPlaneLocatorList) 등이 요구된다. 그림 20는 미리 선언된 서비스 기능 인스턴스를 생성하기 위한 변수들을 통해 인스턴스를 생성하는 소스 코드이다. 생성된 서비스 기능 인스턴스들은 sfList 변수에 해당 정보들이 저장 된다.

4.1.2. Service Function Forwarder (SFF) 생성 소스 코드

SFF를 생성하기 위해서는 SFF의 이름 (SFF_NAME), DataPlane에 대한 정보 (locatorList), 연결되어 있는 서비스 기능 인스턴스들의 정보 (sfDictionaryList)와 연결되어 있는 SFF들의 정보 (sffDictionaryList) 등이 요구된다. 그림 21는 미리 선언된 SFF를 생성하기 위한 변수들을 통해 SFF를 생성하는 소스 코드이다. 생성된 SFF들은 putServiceFuctionForwarder() 함수를 통해 데이터베이스에 저장 된다.

```
ServiceFunctionKey serviceFunctionKey = new ServiceFunctionKey(new SfName(SF_NAMES.get(i)));
sfBuilder.setName(new SfName(SF_NAMES.get(i)))
    .setKey(serviceFunctionKey)
    .setType(SF_TYPES.get(i))
    .setIpMgmtAddress(ipMgmtAddr)
    .setSfDataPlaneLocator(dataPlaneLocatorList);
sfList.add(sfBuilder.build());
```

그림 20 Junit 테스트 소스 코드 (서비스 기능 인스턴스 생성)

```
ServiceFunctionForwarderBuilder sffBuilder = new ServiceFunctionForwarderBuilder();
sffBuilder.setName(new SfName(SFF_NAMES.get(i)))
    .setKey(new ServiceFunctionForwarderKey(new SfName(SFF_NAMES.get(i))))
    .setSffDataPlaneLocator(locatorList)
    .setServiceFunctionDictionary(sfDictionaryList)
    .setConnectedSffDictionary(sffDictionaryList)
    .setServiceNode(null);
ServiceFunctionForwarder sff = sffBuilder.build();
SfcProviderServiceForwarderAPI.putServiceFunctionForwarder(sff);
```

그림 21. Junit 테스트 소스 코드 (SFF 생성)

4.1.3. CPU Utilization 설정 소스 코드

OpenDaylight-SFC에서는 서비스 기능 인스턴스에 대한 CPU Utilization을 Netconf 프로토콜을 통해 인스턴스로부터 받아오며, 해당 정보는 OpenDaylight 데이터베이스에 저장된다. 본 테스트 소스 코드 상에서는 CPU Utilization을 Netconf 서버에서 받아오는 부분은 구현하지 않고, 받은 CPU Utilization을 데이터베이스에 putServiceFunctionState(sfstate)함수를 통해 저장한다. 그림 22는 서비스 기능 인스턴스의 CPU Utilization을 데이터베이스 저장하는 소스 코드이다.

```
ServiceFunctionStateKey serviceFunctionStateKey = new ServiceFunctionStateKey(sfName);
ResourceUtilization resrcUtil = new ResourceUtilizationBuilder().setCPUUtilization((long) cpuUtil).build();
MonitoringInfo monInfo = new MonitoringInfoBuilder().setResourceUtilization(resrcUtil).build();
SfcSfDescMon sfDescMon = new SfcSfDescMonBuilder().setMonitoringInfo(monInfo).build();
ServiceFunctionState sfStatel = new ServiceFunctionStateBuilder().setSfcSfDescMon(sfDescMon).build();
ServiceFunctionState serviceFunctionState = new ServiceFunctionStateBuilder()
    .setKey(serviceFunctionStateKey).addAugmentation(ServiceFunctionState.class, sfStatel).build();
SfcProviderServiceFunctionAPI.putServiceFunctionState(serviceFunctionState);
```

그림 22. Junit 테스트 소스 코드 (CPU 사용량 설정)

4.2. Junit 테스트

4.2에서는 작성된 테스트 소스 코드를 통해 Load/Path-Aware service function scheduler가 Path제한 (L_TH)의 변화에 따라 올바르게 동작하는지 검증한다. Path 제한 (L_TH)의 변화는 2,3,4이다. 테스트 토폴로지는 그림 23과 같다. 해당 토폴로지에서는 3개의 SFF (SFF1, SFF2, SFF3)가 SFF1은 SFF2와 SFF2은 SFF1과 SFF3와 SFF3는 SFF2와 연결되어있다. 또한, 각 SFF들은 서비스 기능 타입이 다른 (Firewall, DPI, NAT) 3개의 서비스 기능이 연결되어 있으며, SF들은 CPU Utilization을 갖고 있다. Junit 테스트를 통해 생성된 토폴로지 와 스케줄러 기능은 로그 파일을 통해 확인한다.

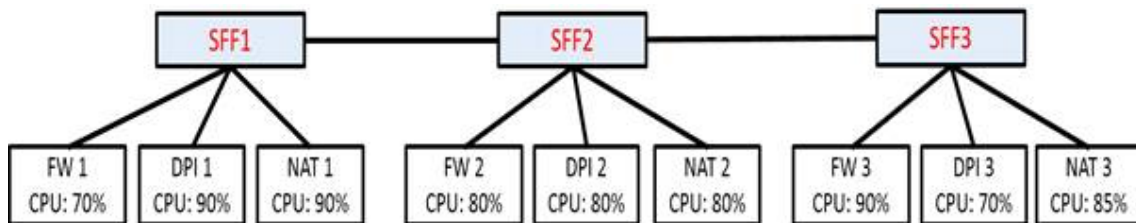


그림 23. Junit 테스트 토폴로지 환경

4.2.1. 테스트 환경

그림 24 로그를 통해 Firewall 타입을 갖는 인스턴스 3개 (FW_1, FW_2, FW_3), DPI 타입을 갖는 인스턴스 3개 (DPI_1, DPI_2, DPI_3), 그리고 NAT 타입을 갖는 SF (NAT_1, NAT_2, NAT_3)이 생성 된 것을 확인 할 수 있다. 각 서비스 기능 인스턴스들은 앞 절에서 기술한바와 같이 인스턴스의 management ip와 dataplane locator 등의 정보들을 통해 생성 된다. 서비스 기능 인스턴스와 같이 SFF 3개 (SFF1, SFF2, SFF3)가 생성된 것을 확인 할 수 있다. 해당 SFF들은 아직 인스턴스와 SFF간의 연결은 이루어지지 않은 상태이다.

```

SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create firewall SF: FW_1
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create napt44 SF: NAT_1
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create dpi SF: DPI_1
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create firewall SF: FW_2
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create napt44 SF: NAT_2
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create dpi SF: DPI_2
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create firewall SF: FW_3
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create napt44 SF: NAT_3
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create dpi SF: DPI_3
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create SFF SFF1
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create SFF SFF2
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create SFF SFF3
  
```

그림 24. Junit 테스트 환경 로그 (SF & SFF 생성)

그림 25 로그를 통해 SFF간의 연결과 SFF와 서비스 기능 인스턴스간의 연결을 확

인 할 수 있다. 먼저 SFF1에 대한 edge를 먼저 설정한다. SFF1과 SFF2와의 edge를 생성 후에 SFF1과 3개의 서비스 기능 인스턴스 (FW_1, NAT_1, DPI_1)들과의 edge를 각각 생성한다. SFF2의 경우 SFF1과 SFF3에 연결되어 있기 때문에 각 SFF들의 edge를 생성하고, SFF1과 마찬가지로 3개의 서비스 기능 인스턴스 (FW_2, NAT_2, DPI_2)들과의 edge를 각각 생성한다. 끝으로 SFF3은 SFF2와의 edge와 3개의 서비스 기능 인스턴스 (FW_3, NAT_3, DPI_3)들과의 edge를 생성한다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create SFF1-to-SFF or SFF1-to-SF edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF1-to-SFF2 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF1-to-FW_1 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF1-to-NAT_1 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF1-to-DPI_1 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create SFF2-to-SFF or SFF2-to-SF edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF2-to-SFF1 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF2-to-SFF3 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF2-to-FW_2 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF2-to-NAT_2 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF2-to-DPI_2 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create SFF3-to-SFF or SFF3-to-SF edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF3-to-SFF2 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF3-to-FW_3 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF3-to-NAT_3 edge
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] - Create SFF3-to-DPI_3 edge
```

그림 25. Junit 테스트 환경 로그 (SF & SFF 링크 생성)

그림 26 로그를 통해 각 인스턴스들의 CPU Utilization을 확인 할 수 있다. 앞 절에서 서술한 바와 같이 각 서비스 기능 인스턴스의 CPU Utilization은 putServiceFunctionState(sfstate)를 통해 OpenDaylight 데이터베이스에 저장한다. 저장된 CPU Utilization은 스케줄러가 특정 서비스 기능 인스턴스의 CPU Utilization이 필요할 때, 데이터베이스로부터 읽어오게 된다. 본 로그를 통해 표 1과 같이 CPU Utilization을 설정한 것을 확인 할 수 있다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF FW_1 CPU : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF FW_2 CPU : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF FW_3 CPU : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF DPI_1 CPU : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF DPI_2 CPU : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF DPI_3 CPU : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF NAT_1 CPU : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF NAT_2 CPU : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Set SF NAT_3 CPU : 85
```

그림 26. Junit 테스트 환경 로그 (SF CPU 사용량 설정)

표 1. Junit 테스트 환경 (SF CPU 사용량)

Type	FW	DPI	NAT
SF Instance / CPU Utilization	FW_1 / 70%	DPI_1 / 90%	NAT_1 / 90%
	FW_2 / 80%	DPI_2 / 80%	NAT_2 / 80%
	FW_3 / 90%	DPI_3 / 70%	NAT_3 / 85%

스케줄러는 체인의 서비스 기능 타입 순서에 맞게 인스턴스를 선택한다. 본 테스트

트에서는 MNC라는 이름을 갖는 MNC-Chain을 설정하였으며, 본 Chain은 Firewall, DPI, NAT 순으로 구성되어 있다. 그림 27 로그는 MNC-Chain의 구성을 나타낸다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - [Test Topology] Create Service Function Chain MNC-Chain : firewall => dpi => napt
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - sfcServiceFunction.name = firewall
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - sfcServiceFunction.name = dpi
SfcServiceFunctionLoadPath-AwareSchedulerAPITest - sfcServiceFunction.name = napt
```

그림 27. MNC Chain 생성 로그

4.2.2. 테스트 결과

먼저 거리제한이 4 홉 일 때 스케줄러의 기능을 확인한다. 거리제한 (L_TH)이 4이면 스케줄러는 이전에 선택한 SF를 기준으로 4홉 거리 까지의 서비스 기능 인스턴스들의 CPU Utilization을 고려하여 인스턴스를 선택한다. MNC-Chain은 Firewall, DPI, NAT 순이므로 스케줄러는 해당 체인의 서비스 기능 타입순으로 인스턴스를 선택한다. 그림 28은 MNC-Chain의 첫 번째 SF 타입인 Firewall의 인스턴스를 FW_1으로 선택한 것에 대한 로그이다. Firewall은 MNC-Chain의 첫 번째 서비스 기능 타입이기 때문에 거리의 제한을 받지 않고, Firewall의 타입을 갖는 인스턴스 중 CPU Utilization이 가장 낮은 인스턴스인 FW_1 (CPU Utilization : 70)을 선택하는 것을 확인 할 수 있다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] ServiceFunctionType name: firewall
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction FW_1 : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction FW_3 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction FW_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] The Seleted SF name: FW_1
```

그림 28. 거리제한 4일 때, 스케줄러 기능 (Firewall 타입의 SF 선택)

그림 29은 두 번째 서비스 기능 타입인 DPI의 인스턴스를 선택하는 과정을 나타내는 로그이다. 이전에 선택된 인스턴스가 FW_1이기 때문에, 스케줄러는 FW_1 기준으로 4홉 이내의 DPI 타입의 인스턴스들 중 CPU Utilization이 가장 낮은 인스턴스를 선택하게 된다. 해당 테스트 토폴로지에서는 모든 서비스 기능 인스턴스간의 거리가 모두 4홉 이내로 분포되어 있기 때문에 거리제한 (L_TH)이 4인 경우 모든 인스턴스들을 고려한다. 따라서, DPI_1, DPI_2, DPI_3와 FW_1간의 거리는 2홉, 3홉, 4홉이기 때문에 모든 DPI 타입의 인스턴스들은 선택가능하며, 그 중 가장 낮은 CPU Utilization을 갖는 DPI 타입의 인스턴스인 DPI_3 (CPU Utilization : 70)을 선택하게 된다. 그림 30은 마지막 서비스 기능 타입인 NAT 타입의 인스턴스를 선택하는 과정을 나타내는 로그이다. 이전에 선택된 인스턴스가 DPI_3이고, DPI_3를 기준으로 NAT_1, NAT_2, NAT_3 까지의 거리는 4홉, 3홉, 2홉이기 때문에 모든 NAT 타입의 인스턴스들은 선택 가능하다. 따라서, 스케줄러는 가장 낮은 CPU

Utilization을 갖는 NAT타입의 인스턴스인 NAT_2 (CPU Utilization : 80)를 선택한다. 그림 31은 거리제한 (L_TH)이 4일 때, 선택된 인스턴스를 나타낸다. 그림에서 초록색 박스는 이전에 선택된 서비스 기능 인스턴스를 기준으로 거리제한 (L_TH)안에 분포된 인스턴스를 의미한다. 그림과 같이 스케줄러는 거리제한 (L_TH)이 4일 때, FW_1, DPI_3, NAT_2 순으로 서비스 기능 인스턴스를 선택한다. MNC-Chain에 대한 RSP는 그림 31의 로그와 같다.

다음으로는 거리제한이 3 홉인 경우에 Junit 테스트 결과를 확인한다. 거리제한 (L_TH)이 3이면 스케줄러는 이전에 선택한 SF를 기준으로 3 홉까지의 인스턴스들의 CPU Utilization를 고려하여 인스턴스를 선택한다. MNC-Chain은 Firewall, DPI, NAT 순이므로 스케줄러는 해당 체인의 서비스 기능 타입의 순으로 인스턴스를 선택한다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] ServiceFunctionType name: firewall
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction FW_1 : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction FW_3 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction FW_2 : 80
```

그림 32. 거리제한 3일 때, 스케줄러 기능 (Firewall 타입의 SF 선택)

그림 32는 MNC-Chain의 첫 번째 서비스 기능 타입인 Firewall의 인스턴스를 FW_1으로 선택한 것에 대한 로그이다. Firewall은 MNC-Chain의 첫 번째 서비스 기능 타입이기 때문에 거리의 제한을 받지 않고, Firewall의 타입을 갖는 SF중 CPU Utilization이 가장 낮은 인스턴스인 FW_1 (CPU Utilization : 70)을 선택하는 것을 확인 할 수 있다. 그림 33은 두 번째 서비스 기능 타입인 DPI의 인스턴스를 선택하는 과정을 나타내는 로그이다. 이전에 선택된 서비스 기능 인스턴스가 FW_1이기 때문에, 스케줄러는 FW_1 기준으로 3홉 이내의 DPI 타입의 인스턴스들 중 CPU Utilization이 가장 낮은 인스턴스를 선택하게 된다. 따라서, FW_1을 기준으로 DPI_1, DPI_2, DPI_3들과의 거리는 2홉, 3홉, 4홉이기 때문에, DPI_3은 거리제한을 벗어나게 된다. 따라서, 비록 DPI_3 (CPU Utilization : 70)이 DPI 타입의 인스턴스 중 가장 낮은 CPU Utilization을 갖지만 거리제한으로 인해, 스케줄러는 선택 가능한 DPI 타입 인스턴스들 (DPI_1, DPI_2)중 낮은 CPU Utilization을 갖는 DPI_2 (CPU Utilization : 80)을 선택한다. 그림 34는 마지막 서비스 기능 타입인 NAT 타입의 인스턴스를 선택하는 과정을 나타내는 로그이다. 이전에 선택된 서비스 기능 인스턴스가 DPI_2이고, DPI_2를 기준으로 NAT_1, NAT_2, NAT_3까지의 거리는 3홉, 2홉, 3홉이기 때문에 모든 NAT 타입의 인스턴스들은 선택 가능하다. 따라서, 스케줄러는 가장 낮은 CPU Utilization을 갖는 NAT 타입의 인스턴스인 NAT_2 (CPU Utilization : 80)를 선택한다. 그림 35는 거리제한 (L_TH)이 3일 때, 선택된 인스턴스를 나타낸다. 그림에서 초록색 박스는 이전에 선택된 인스턴스를 기준으로 거리제한 (L_TH)안에 분포된 인스턴스를 의미한다. 그림과 같이 스케줄러는 거리제한 (L_TH)이 3일 때, FW_1, DPI_2, NAT_2 순으로 서비스 기능 인스턴스를 선택한다. MNC-Chain에 대한 RSP는 그림 35 로그와 같다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] ServiceFunctionType name: dpi
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction DPI_1 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Length of shortestpath between FW_1 and DPI_1 : 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Refining candidate set: Inclue a candidate SF intance FW_1 whose distance to DPI_1 is less than or equal to 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction DPI_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Length of shortestpath between FW_1 and DPI_2 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Refining candidate set: Inclue a candidate SF intance FW_1 whose distance to DPI_2 is less than or equal to 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction DPI_3 : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Length of shortestpath between FW_1 and DPI_3 : 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Refining candidate set: Inclue a candidate SF intance FW_1 whose distance to DPI_3 is less than or equal to 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] The Seleted SF name: DPI_3
```

그림 29. 거리제한 4일 때, 스케줄러 기능 (DPI 타입의 SF 선택)

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] ServiceFunctionType name: napt
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction NAT_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Length of shortestpath between DPI_3 and NAT_2 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Refining candidate set: Inclue a candidate SF intance DPI_3 whose distance to NAT_2 is less than or equal to 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction NAT_3 : 85
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Length of shortestpath between DPI_3 and NAT_3 : 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Refining candidate set: Inclue a candidate SF intance DPI_3 whose distance to NAT_3 is less than or equal to 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Read CPU of ServiceFunction NAT_1 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Length of shortestpath between DPI_3 and NAT_1 : 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] Refining candidate set: Inclue a candidate SF intance DPI_3 whose distance to NAT_1 is less than or equal to 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=4] The Seleted SF name: NAT_2
```

그림 30. 거리제한 4일 때, 스케줄러 기능 (NAT 타입의 SF 선택)

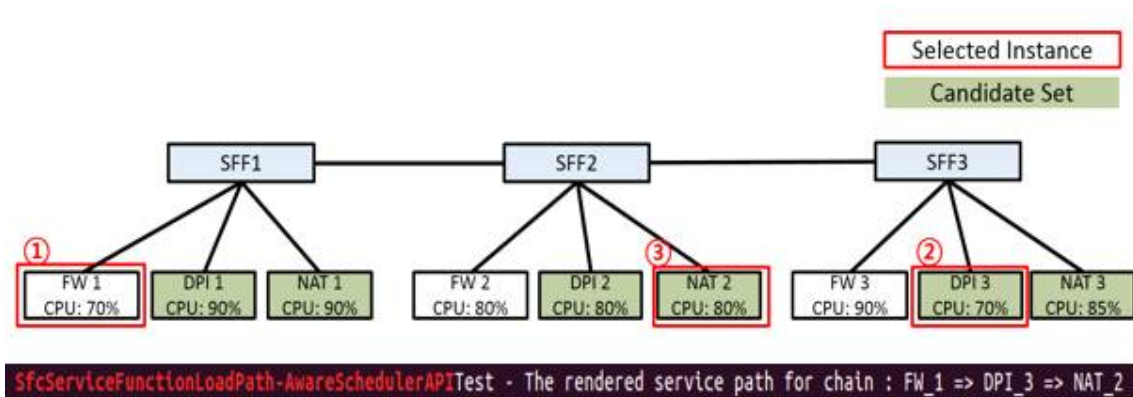


그림 31. 거리제한 4일 때, 스케줄러 테스트 결과

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] ServiceFunctionType name: dpi
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction DPI_1 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Length of shortestpath between FW_1 and DPI_1 : 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Refining candidate set: Inclue a candidate SF intance FW_1 whose distance to DPI_1 is less than or equal to 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction DPI_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Length of shortestpath between FW_1 and DPI_2 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Refining candidate set: Inclue a candidate SF intance FW_1 whose distance to DPI_2 is less than or equal to 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction DPI_3 : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Length of shortestpath between FW_1 and DPI_3 : 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Refining candidate set: Exclue a candidate SF intance FW_1 whose distance to DPI_3 is larger than 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] The Seleted SF name: DPI_2
```

그림 33. 거리제한 3일 때, 스케줄러 기능 (DPI 타입의 SF 선택)


```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] ServiceFunctionType name: napt
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction NAT_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Length of shortestpath between DPI_2 and NAT_2 : 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Refining candidate set: Include a candidate SF Instance DPI_2 whose distance to NAT_2 is less than or equal to 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction NAT_3 : 85
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Length of shortestpath between DPI_2 and NAT_3 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Refining candidate set: Include a candidate SF Instance DPI_2 whose distance to NAT_3 is less than or equal to 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction NAT_1 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Length of shortestpath between DPI_2 and NAT_1 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Refining candidate set: Include a candidate SF Instance DPI_2 whose distance to NAT_1 is less than or equal to 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] The Selected SF name: NAT_2
```

그림 34. 거리제한 3일 때, 스케줄러 기능 (NAT 타입의 SF 선택)

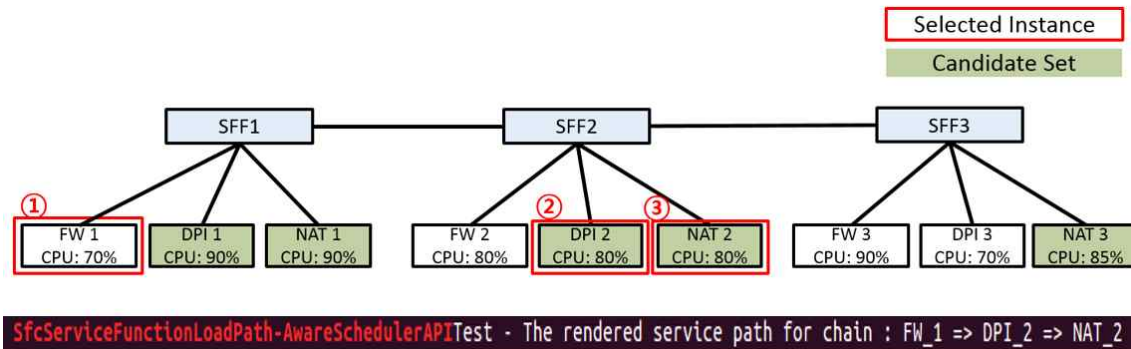


그림 35. 거리제한 3일 때, 스케줄러 테스트 결과

마지막으로, 거리제한이 2 hops인 경우의 Junit 테스트 결과를 확인한다. 거리제한 (L_TH)이 2이면 스케줄러는 이전에 선택한 서비스 기능 인스턴스를 기준으로 2 hops 거리 까지의 인스턴스들의 CPU Utilization를 고려하여 서비스 기능 인스턴스를 선택한다. MNC-Chain은 Firewall, DPI, NAT 순이므로 스케줄러는 해당 체인의 서비스 기능 타입의 순으로 인스턴스를 선택한다. 그림 36은 MNC-Chain의 첫 번째 서비스 기능 타입인 Firewall의 인스턴스를 FW_1으로 선택한 것에 대한 로그이다. Firewall은 MNC-Chain의 첫 번째 서비스 기능 타입이기 때문에 거리의 제한을 받지 않고, Firewall의 타입을 갖는 인스턴스 중 CPU Utilization이 가장 낮은 인스턴스인 FW_1 (CPU Utilization : 70)을 선택하는 것을 확인 할 수 있다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] ServiceFunctionType name: firewall
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction FW_1 : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction FW_3 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=3] Read CPU of ServiceFunction FW_2 : 80
```

그림 36. 거리제한 2일 때, 스케줄러 기능 (Firewall 타입의 SF 선택)

그림 37은 두 번째 서비스 기능 타입인 DPI의 인스턴스를 선택하는 과정을 나타내는 로그이다. 이전에 선택된 인스턴스가 FW_1이기 때문에, 스케줄러는 FW_1 기준으로 2 hops 이내의 DPI 타입의 인스턴스들 중 CPU Utilization이 가장 낮은 인스턴스를 선택하게 된다. 따라서, FW_1과 동일한 SFF인 SFF1 에 연결된 DPI 타입의 인스턴스인 DPI_1만 선택 가능하다. FW_1을 기준으로 DPI_1, DPI_2, DPI_3까지의 거리는 2 hops, 3 hops, 4 hops로 정의된다. 비록 DPI_1 (CPU Utilization : 90)이 DPI 타입의 인스턴스들 중에 가장 높은 CPU Utilization을 갖지만, 거리제한으로 인해 스케줄러

는 DPI_1을 선택하게 된다. 그림 38은 마지막 서비스 기능 타입인 NAT타입의 인스턴스를 선택하는 과정을 나타내는 로그이다. 이전에 선택된 인스턴스가 DPI_1이고, DPI_1를 기준으로 NAT_1, NAT_2, NAT_3까지의 거리는 2홉, 3홉, 4홉이기 때문에 NAT_1을 제외한 NAT 타입의 인스턴스들 (NAT_2, NAT_3)은 선택이 불가능하다. 따라서, 선택 가능한 NAT 타입의 인스턴스가 NAT_1으로 하나이기 때문에, 스케줄러는 NAT_1을 선택한다. 그림 39는 거리제한 (L_TH)이 2일 때, 선택된 인스턴스를 나타낸다. 그림에서 초록색 박스는 이전에 선택된 서비스 기능 인스턴스를 기준으로 거리제한 (L_TH)안에 분포된 인스턴스를 의미한다. 그림과 같이 스케줄러는 거리제한 (L_TH)이 2일 때, FW_1, DPI_1, NAT_1 순으로 서비스 기능 인스턴스를 선택한다. MNC-Chain에 대한 RSP는 그림 39 로그와 같다.

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] ServiceFunctionType name: dpi
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Read CPU of ServiceFunction DPI_1 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Length of shortestpath between FW_1 and DPI_1 : 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Refining candidate set: Include a candidate SF instance FW_1 whose distance to DPI_1 is less than or equal to 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Read CPU of ServiceFunction DPI_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Length of shortestpath between FW_1 and DPI_2 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Refining candidate set: Exclude a candidate SF instance FW_1 whose distance to DPI_2 is larger than 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Read CPU of ServiceFunction DPI_3 : 70
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Length of shortestpath between FW_1 and DPI_3 : 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Refining candidate set: Exclude a candidate SF instance FW_1 whose distance to DPI_3 is larger than 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] The Seleted SF name: DPI_1
```

그림 37. 거리제한 2일 때, 스케줄러 기능 (DPI 타입의 SF 선택)

```
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] ServiceFunctionType name: napt
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Read CPU of ServiceFunction NAT_2 : 80
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Length of shortestpath between DPI_1 and NAT_2 : 3
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Refining candidate set: Exclude a candidate SF instance DPI_1 whose distance to NAT_2 is larger than 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Read CPU of ServiceFunction NAT_3 : 85
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Length of shortestpath between DPI_1 and NAT_3 : 4
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Refining candidate set: Exclude a candidate SF instance DPI_1 whose distance to NAT_3 is larger than 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Read CPU of ServiceFunction NAT_1 : 90
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Length of shortestpath between DPI_1 and NAT_1 : 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] Refining candidate set: Include a candidate SF instance DPI_1 whose distance to NAT_1 is less than or equal to 2
SfcServiceFunctionLoadPath-AwareSchedulerAPI - [L_TH=2] The Seleted SF name: NAT_1
```

그림 38. 거리제한 2일 때, 스케줄러 기능 (NAT 타입의 SF 선택)

4.2에서는 거리제한을 변경하면서 구현된 스케줄러의 기능을 소스 코드 레벨의 테스트를 통해 확인 하였다. 4.3에서는 구현된 스케줄러가 포함된 컨트롤러 구동하여 컨트롤러 상에서의 구현된 스케줄러의 기능을 테스트 한다. 또한 구현된 스케줄러 뿐 만 아니라, 2장에서 기술한 최단거리 서비스 기능 스케줄러와 로드밸런스 서비스 기능 스케줄러도 동일한 환경에서 테스트하여 구현된 스케줄러와 기존의 스케줄러의 기능을 비교 검증한다.

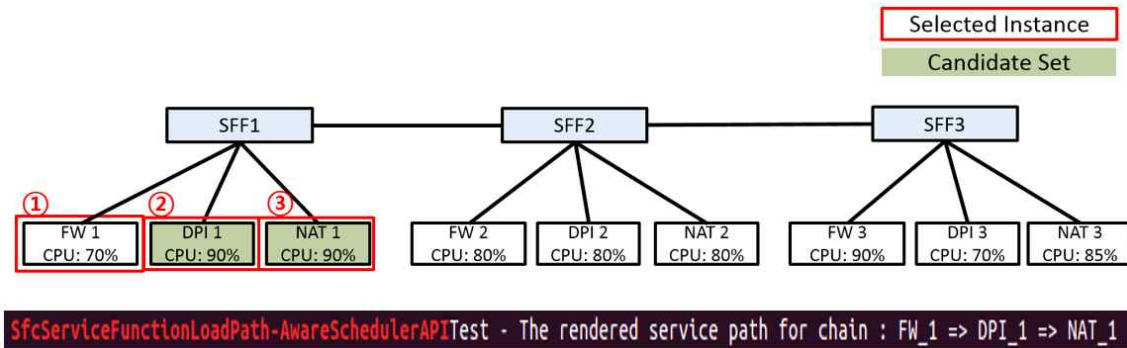


그림 39. 거리제한 2일 때, 스케줄러 테스트 결과

4.3. OpenDaylight 컨트롤러 연동 테스트

4.3에서는 구현된 스케줄러를 OpenDaylight 컨트롤러 상에서 수행된 테스트에 대한 결과와 동일한 환경에서 기존의 스케줄러들의 테스트 결과를 기술한다. 먼저 테스트 환경에 대해 기술하고, 해당 환경에서 최단거리, 로드밸런스, Load/Path-Aware 스케줄러의 테스트 결과를 비교한다.

4.3.1. 테스트 환경

스케줄러들의 검증을 위한 테스트 환경은 그림40과 같다. 테스트 토폴로지에 구성된 서비스 기능 인스턴스와 SFF들은 OpenDaylight-SFC 컨트롤러 UI를 통해 확인할 수 있다.

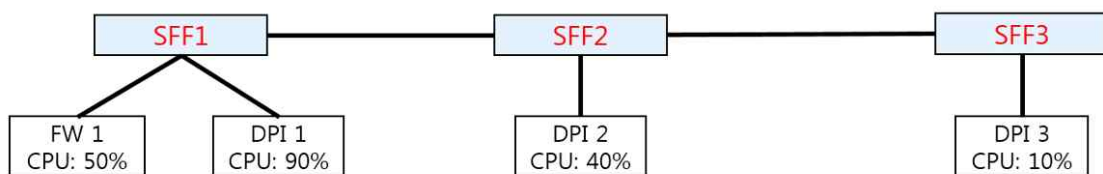


그림 40. 테스트 환경

그림 41은 OpenDaylight 컨트롤러에 등록된 인스턴스들을 나타낸다. 그림[토폴로지]처럼 3개의 DPI 타입의 인스턴스들은 DPI1, DPI2, DPI3로, Firewall 타입의 인스턴스는 FW1으로 등록 된 것을 확인 할 수 있다.

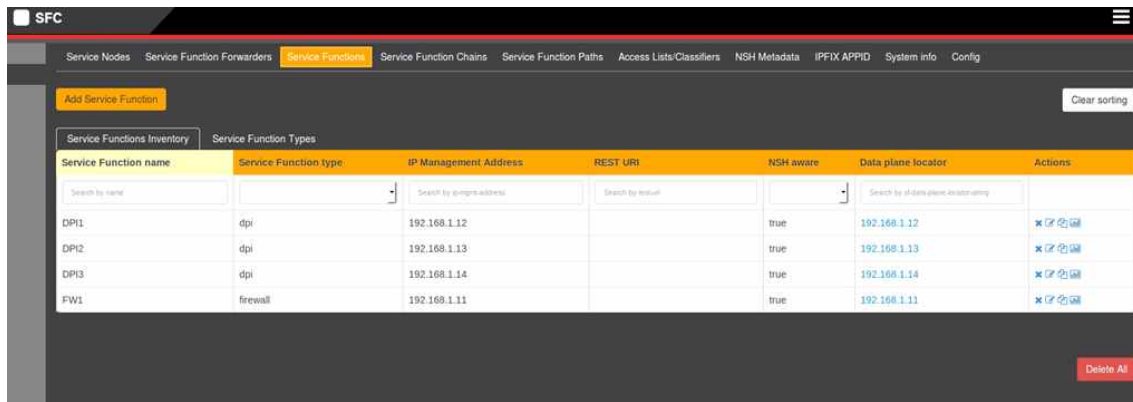


그림 41. 테스트 SF 등록

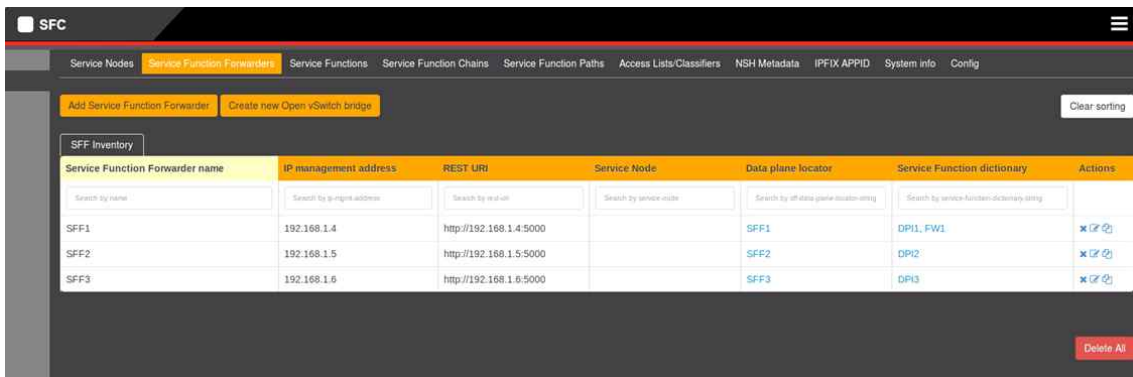
그림 42는 서비스 기능 인스턴스의 Description 정보 (서비스 기능 타입, 데이터 플랜 IP, Port)와 인스턴스의 모니터링 정보 (CPU 사용량, Memory 사용량)을 해당 SF에서부터 컨트롤러가 받아오는 Karaf 로그이다. 그림 42을 통해 FW1의 CPU 사용량은 50, DPI1의 CPU 사용량은 90, DPI2의 CPU 사용량은 40, 그리고 DPI3의 CPU 사용량은 10인 것을 확인 할 수 있다. 해당 정보는 컨트롤러의 데이터베이스에 저장된다.

INFO	MonitoringInfo of SF FW1: CPU utilization: 50, Memory utilization: 17	INFO	MonitoringInfo of SF DPI3: CPU utilization: 10, Memory utilization: 17
INFO	getSFMonitoringInfo() succeeded.	INFO	getSFMonitoringInfo() succeeded.
INFO	Successfully created SF from Netconf node FW1	INFO	Successfully created SF from Netconf node DPI3
INFO	DescriptionInfo of SF FW1: type: firewall, data-plane-ip: 192.168.1.11, data-plane-port: 6634	INFO	DescriptionInfo of SF DPI3: type: dpi, data-plane-ip: 192.168.1.14, data-plane-port: 6634
INFO	getSFDescription() successfully.	INFO	getSFDescription() successfully.
INFO	NETCONF Listener created event: FW1	INFO	NETCONF Listener created event: DPI3
FW 1 SF정보 및 CPU 사용량		DPI 1 SF정보 및 CPU 사용량	
INFO	MonitoringInfo of SF DPI2: CPU utilization: 40, Memory utilization: 17	INFO	MonitoringInfo of SF DPI1: CPU utilization: 90, Memory utilization: 17
INFO	getSFMonitoringInfo() succeeded.	INFO	getSFMonitoringInfo() succeeded.
INFO	Successfully created SF from Netconf node DPI2	INFO	Successfully created SF from Netconf node DPI1
INFO	DescriptionInfo of SF DPI2: type: dpi, data-plane-ip: 192.168.1.13, data-plane-port: 6634	INFO	DescriptionInfo of SF DPI1: type: dpi, data-plane-ip: 192.168.1.12, data-plane-port: 6634
INFO	getSFDescription() successfully.	INFO	getSFDescription() successfully.
INFO	NETCONF Listener created event: DPI2	INFO	NETCONF Listener created event: DPI1
DPI 2 SF정보 및 CPU 사용량		DPI 3 SF정보 및 CPU 사용량	

그림 42. Karaf 로그 (SF 정보 및 CPU 사용량)

그림 43는 컨트롤러에 등록된 3개의 SFF들과 각 SFF에 연결 된 서비스 기능 인스턴스를 나타낸다. 해당 UI를 통해 SFF1에는 FW1과 DPI1이 연결되어 있으며, SFF2에는 DPI2가, 그리고 SFF3에는 DPI 3가 연결 된 것을 확인 할 수 있다.

그림 44는 생성된 Service Function Chain을 나타낸다. 본 테스트에서는 MNC의 이름 갖은 체인을 생성하였으며, 해당 체인의 순서는 Firewall에서 DPI로 정의된다. 즉, 스케줄러는 MNC 체인에 대한 SF를 선택 할 때, 해당 스케줄러의 목적에 맞는 Firewall 타입의 인스턴스를 먼저 선택하고, 그 이후, DPI 타입의 인스턴스를 선택



The screenshot shows the SFC UI with the 'Service Function Forwarders' tab selected. Below the navigation bar, there are buttons for 'Add Service Function Forwarder' and 'Create new Open vSwitch bridge'. The main area displays the 'SFF Inventory' table with the following data:

Service Function Forwarder name	IP management address	REST URI	Service Node	Data plane locator	Service Function dictionary	Actions
SFF1	192.168.1.4	http://192.168.1.4:5000		SFF1	DPI1, FW1	[x] [i] [e]
SFF2	192.168.1.5	http://192.168.1.5:5000		SFF2	DPI2	[x] [i] [e]
SFF3	192.168.1.6	http://192.168.1.6:5000		SFF3	DPI3	[x] [i] [e]

At the bottom right, there is a 'Delete All' button.

그림 43. 테스트 SFF 등록

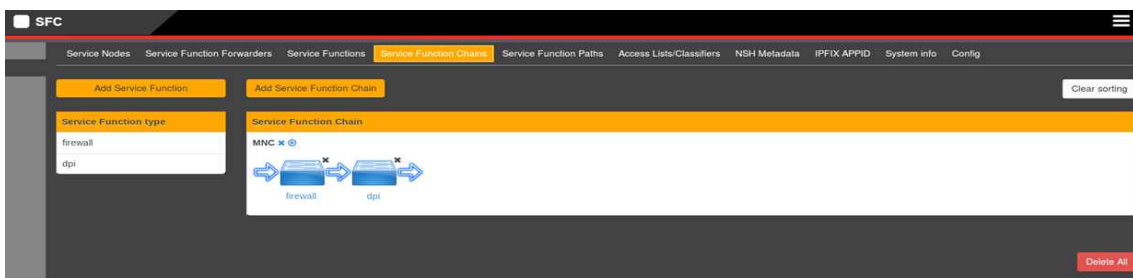


그림 44. 체인 생성 (MNC 체인)

한다.

지금까지 OpenDaylight-SFC UI를 통해 테스트 환경을 검증 하였다. 4.3.2에서는 생성된 테스트 환경에서 스케줄러들의 기능을 확인한다.

4.3.2. 테스트 결과

그림 40과 같은 환경에서 Firewall, DPI 순으로 정의된 MNC 체인에 대한 스케줄러의 기능을 확인하다. 먼저 기존의 스케줄러인 최단거리 (MNC-SP)와 로드밸런스 (MNC-LB)스케줄러가 테스트 환경에서 제대로 동작하는지 확인한다. 끝으로는 홑수 제한이 2, 3, 4인 경우 구현 된 스케줄러 (MNC-LP-LTH2, 3, 4)의 기능을 확인하고, 앞서 확인 한 최단거리 서비스 기능 스케줄러와 로드밸런스 서비스 기능 스케줄러의 기능과 비교한다.

최단거리 서비스 기능 스케줄러의 목적은 선택된 서비스 기능 인스턴스들이 체인의 순서를 만족하면서, 인스턴스들간의 홑 수를 최소화 하는 것이다. 그림 46은 최단거리 서비스 기능 스케줄러가 MNC 체인에 따라 선택한 서비스 기능 인스턴스들을 나타낸다. 그림46을 통해 최단거리 서비스 기능 스케줄러가 FW1과 DPI1을 지속적으로 선택하는 것을 확인 할 수 있다. 해당 결과의 이유는 다음과 같다. 우선

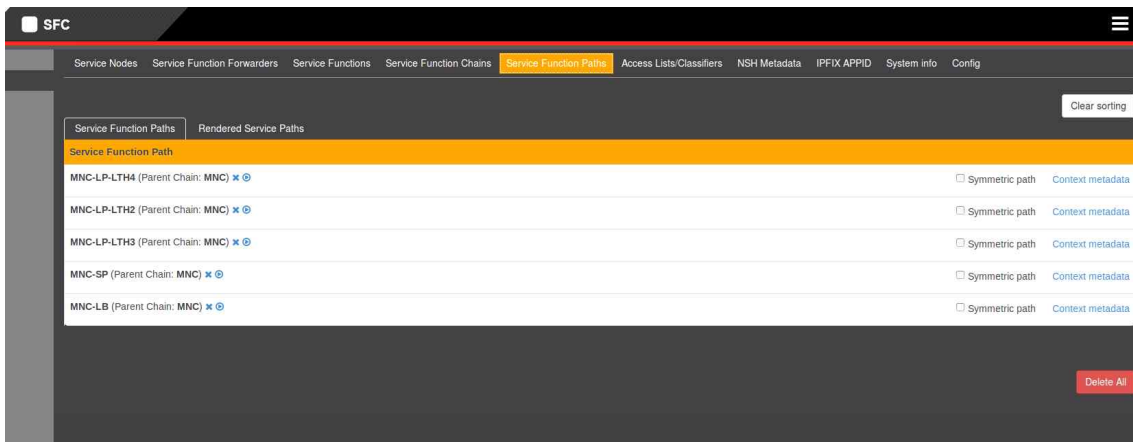


그림 45. SFP 생성

MNC 체인 순서에 의해 Firewall 타입의 인스턴스를 선택하게 된다. Firewall 타입의 인스턴스가 테스트 환경에서는 FW1만 존재하기 때문에, FW1을 선택한다. 그 다음 DPI 타입의 인스턴스를 선택한다. DPI 타입의 인스턴스는 3개가 존재하고, FW1과 DPI1, DPI2, DPI3들과의 홉 수는 2홉, 3홉, 4홉이 된다. 따라서, FW1과 홉 수가 가장 작은 DPI1을 지속적으로 선택하게 된다.

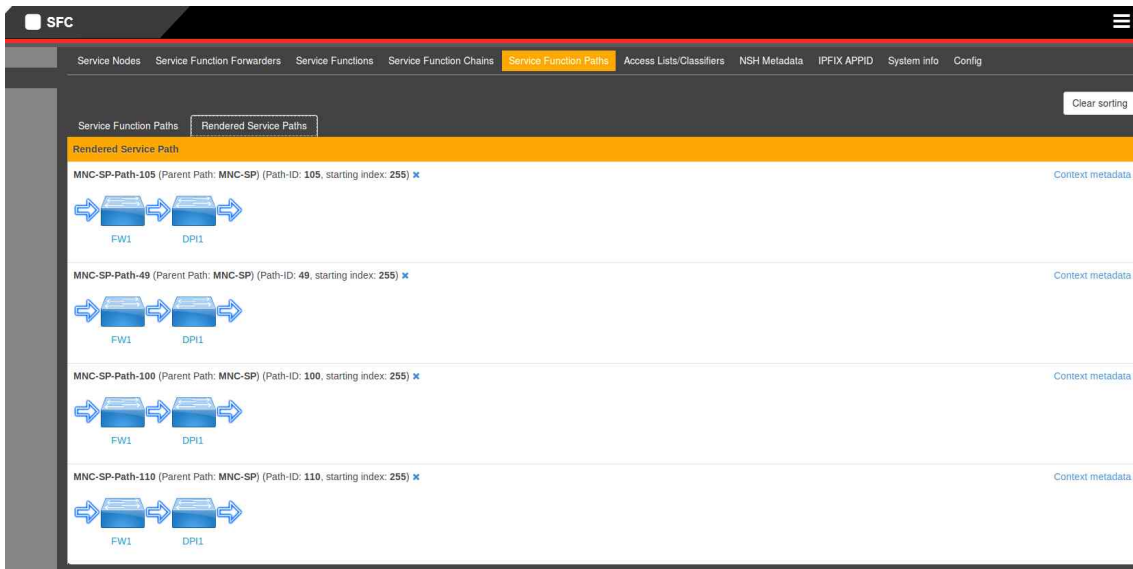


그림 46. Shortestpath 스케줄러 기능 검증

로드밸런스 스케줄러의 목적은 체인의 순서대로 서비스 기능 인스턴스를 선택할 때, 최소의 부하를 갖는 인스턴스를 선택 하는 것이다. 그림 47은 스케줄러가 MNC 체인 순서에 따라 선택한 서비스 기능 인스턴스들을 나타낸다. 그림47을 통해 스케줄러가 FW1과 DPI3을 지속적으로 선택하는 것을 확인 할 수 있다. 해당 결과의 이유는 다음과 같다. 우선 MNC 체인 순서에 의해 Firewall 타입의 인스턴스를 선택하게 된다. Firewall 타입의 인스턴스가 테스트 환경에서는 FW1만 존재하기 때문

에, FW1을 선택한다. 그 다음 DPI 타입인 인스턴스들 중 가장 낮은 CPU 사용량을 갖는 DPI3을 선택한다. 해당 결과를 통해 스케줄러가 CPU 사용량이 가장 낮은 인스턴스를 지속적으로 선택하는 것을 확인 할 수 있다.

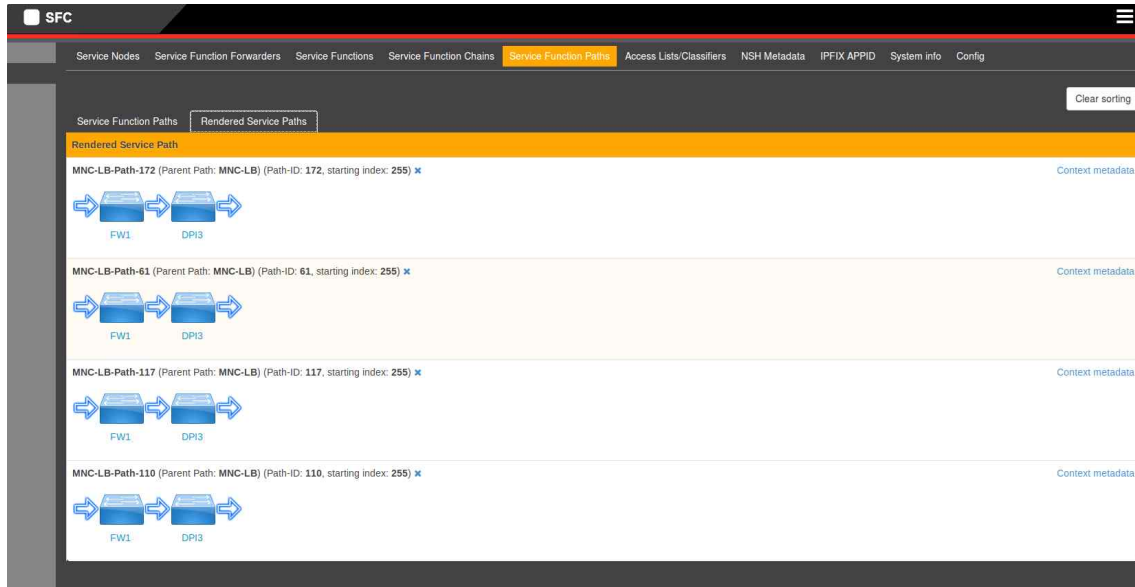


그림 47. Loadbalance 스케줄러 기능 검증

개발 한 Load/Path-Aware 스케줄의 목적은 체인의 순서대로 서비스 기능 인스턴스를 선택 할 때, 앞서 선택한 서비스 기능 인스턴스 기준으로 거리제한(L_TH)내의 인스턴스 중 가장 낮은 부하를 갖는 인스턴스를 선택한 것이다. 본 테스트는 거리 제한이 2,3 그리고 4인 경우의 구현된 스케줄러의 기능을 확인한다.

그림 48은 거리제한 (L_TH)이 2홉인 경우의 스케줄러에 의해 선택된 서비스 기능 인스턴스들을 나타낸다. 해당 결과를 통해 거리제한이 2홉인 경우 FW1과 DPI1을 지속적으로 선택하는 것을 확인 할 수 있다. Firewall 타입의 인스턴스가 FW1 하나이기 때문에 선택되고, FW1 기준으로 2홉내의 DPI 타입의 인스턴스는 DPI1이기 때문에 스케줄러는 두번째 인스턴스로 DPI1을 선택한다. 본 결과는 앞서 확인 최단 거리 서비스 기능 스케줄러의 결과와 동일 한 것을 확인 할 수 있다. 하지만, 본 환경에서는 서비스 기능 타입이 두 종류로 분포되며, 두 종류의 인스턴스들간의 최소 거리가 2홉이기 때문에 동일한 결과가 나온 것이지, 항상 거리제한이 짧다고 두 종류의 스케줄러의 결과가 항상 같지는 않다.

그림 49는 거리제한이 3홉인 경우의 스케줄러의 결과를 나타낸다. 스케줄러는 지속적으로 FW1과 DPI2을 선택하게 된다. 거리제한이 2홉인 경우와 다르게 3홉의 경우 FW1을 기준으로 거리제한 내의 DPI 타입의 인스턴스는 DPI1, DPI2가 된다. 선택 가능한 DPI 타입의 인스턴스들 중 DPI2의 CPU 사용량이 가장 낮기 때문에 스케줄러는 DPI2를 두 번째 인스턴스로 선택한다.

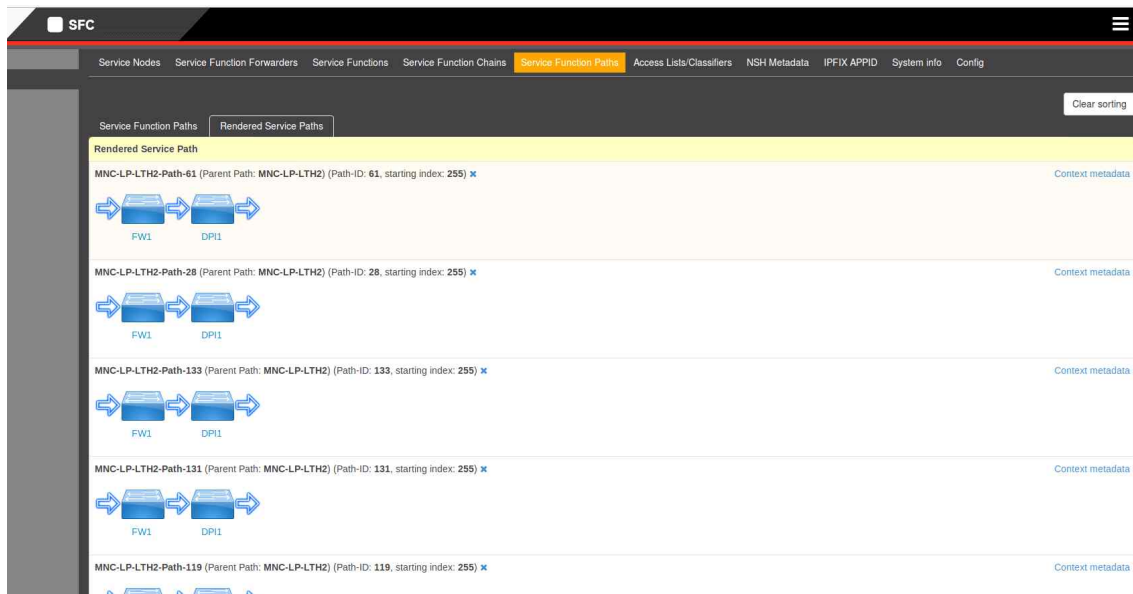


그림 48. 거리제한이 2홉인 경우의 Load/Path aware 스케줄러 기능 검증

본 테스트 환경에서는 모든 서비스 기능 인스턴스간의 거리가 4 홉 이내로 분포하기 때문에, 거리제한이 4홉인 스케줄러는 모든 인스턴스들을 거리제한 없이 선택할 수 있다. 그림 50은 거리제한이 4홉인 경우의 스케줄러의 결과를 나타낸다. 본 결과는 앞서 설명한 로드밸런스 서비스 기능 스케줄러와 동일한 결과인 것을 확인할 수 있다. 거리제한이 모든 서비스 기능 인스턴스간의 거리보다 길기 때문에, 개발한 스케줄러는 로드밸런스 서비스 기능 스케줄러와 같이 동작한다. 즉, 거리제한 모든 인스턴스들 간의 거리들 보다 길면 스케줄러는 로드밸런스 서비스 기능 스케줄러와 동일하게 동작한다.

본 장에서는 개발한 스케줄러의 소스 코드 검증 및 기능 검증을 위해 JAVA 코드 테스트인 Junit 테스트와 컨트롤러를 통한 테스트를 진행하였으며, 해당 테스트들을 통해 개발한 스케줄러의 기능을 검증 하였다.

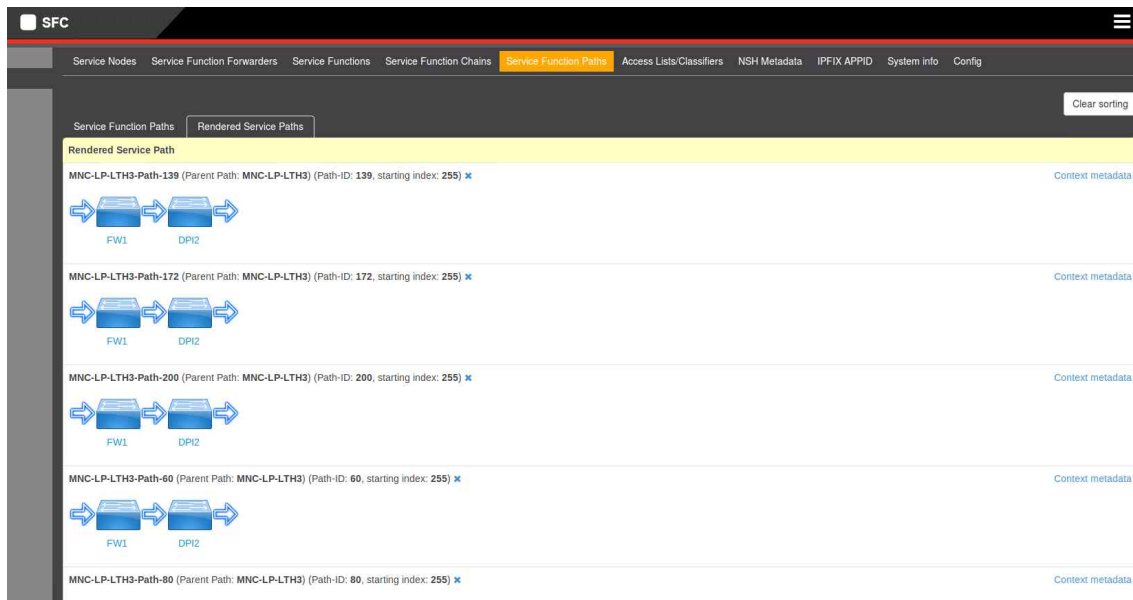


그림 49. 거리제한이 3홉인 경우의 Load/Path-Aware 스케줄러 기능 검증

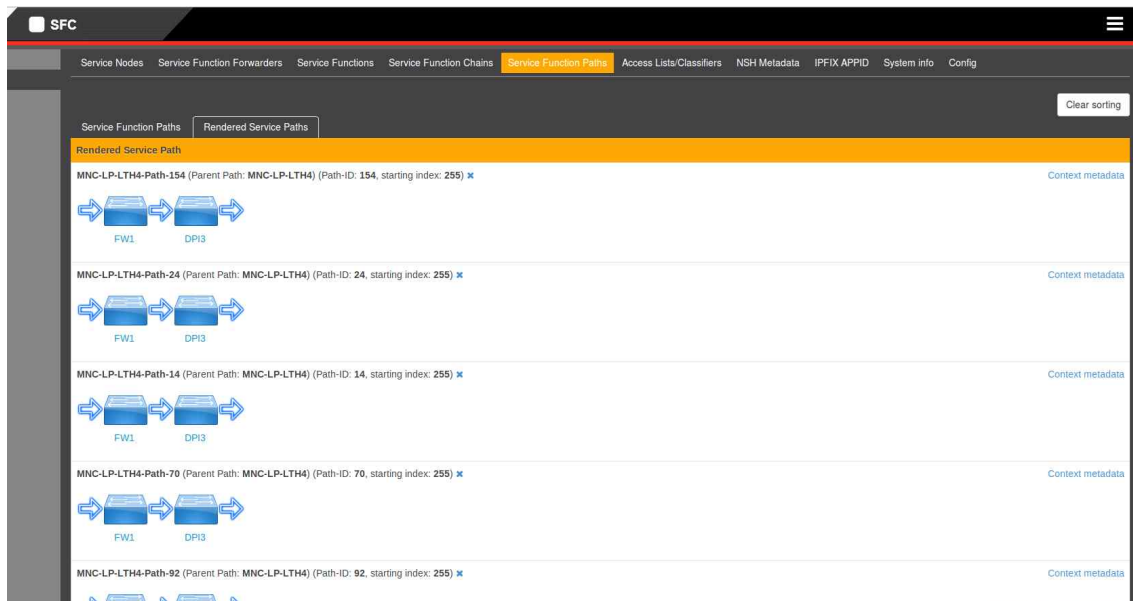


그림 50. 거리제한이 4홉인 경우의 Load/Path-Aware 스케줄러 기능 검증

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] ETSI ISG NFV, “Network Functions Virtualisation; Use Cases,” GS NFV-002 v1.1.1, October 2013.

K-ONE 기술 문서

- K-ONE 컨소시엄의 확인과 허가 없이 이 문서를 무단 수정하여 배포하는 것을 금지합니다.
- 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: 2016/04