# K-ONE 기술 문서 #13

# ONOS 상 LISP Provider 설계 및 구현

Document No. K-ONE #13
Version 1.0
Date 2016-05-08
Author(s) 류승호

K-ONE

ONK
OpenNetworkingKorea

■ 문서의 연혁

| 버전 | 날짜 | 작성자 | 내용 |
|---|---|---|---|
| 초안 - 0.1 | 2016. 03. 28 | 류승호 | 초안 작성 |
| 0.2 | 2016. 03. 30 | 류승호 | 프로바이더 설계 내용 수정 |
| 0.3 | 2016. 04. 02 | 류승호 | 내용 추가 (프로바이더 동작확인) |
| 0.4 | 2016. 04. 10 | 류승호 | 오탈자 수정 |
| 0.5 | 2016. 04. 18 | 류승호 | SDN 그림 추가 |
| 0.6 | 2016. 04. 22 | 류승호 | 설계 내용 추가 |
| 0.7 | 2016. 04. 25 | 류승호 | 설계 그림 수정 |
| 0.8 | 2016. 04. 30 | 류승호 | 프로바이더 동작확인 내용 수정 |
| 0.9 | 2016. 05. 02 | 류승호 | LISP 프로바이더 설명 추가 |
| 1.0 | 2016. 05. 08 | 류승호 | 오탈자 수정 |

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

소프트웨어 정의 네트워크(Software Defined Network, SDN)에서는 네트워크 인프라스트럭처를 관리 및 조작하기 위해서 전송 평면과 제어 평면을 분리시켰다. 네트워크에 있는 장비들은 데이터 전송을 담당하고 중앙 집중 컨트롤러가 네트워크를 지능을 담당하게 된다. 따라서, SDN의 핵심기술은 중앙 집중 컨트롤러와 네트워크 장비들의 동작을 제어하기 위한 프로토콜이다. 네트워크 프로토콜(일명 SBI)로는 현재 OpenFlow가 SDN의 표준으로 사용되고 있으나 SDN 컨트롤러 개발에 여러 업체와 기관이 뛰어들면서 현재 다양한 SDN 컨트롤러들이 존재하고 있다. 여러 종류의 컨트롤러 중에서 ONOS와 ODL가 대규모 네트워크에 실제 적용 가능한 컨트롤러로 고려되면서 최근에 많은 관심을 받고 있다. ODL은 다양한 네트워크 프로토콜들을 SBI로 지원하여 네트워크에 있는 기존 장비들을 포함한 대규모의 SDN을 구성하는 방법을 취하고 있다. ONOS는 SDN에서 단일 컨트롤러가 다수의 네트워크 장비들을 관리하는데 발생하는 네트워크의 성능 저하 문제를 해결하고자 한다. ONOS는 SDN의 성능을 기존 네트워크 수준으로 제공함으로써 네트워크 사업자와 시장의 요구를 만족할 수 있어 캐리어 레벨의 네트워크에 적용 가능할 것으로 기대를 받고 있다. 하지만 ONOS는 컨트롤러의 성능 향상에 초점을 맞추고 있어 ODL에 비해 SBI의 지원이 매우 미비한 실정이다. OpenFlow 장비로 현재 인터넷을 대체하기에는 높은 비용과 시간이 필요할 것으로 예상된다. SDN과 기존 인터넷이 공존하는 상황에서 SDN 컨트롤러는 네트워크에 있는 다양한 장비들과 협업을 할 필요가 존재한다. 이에 따라, ONOS의 부족한 SBI 지원을 보안하기 위한 프로젝트들이 있었으며 그 결과 NetConf, BGP, OVSDB 등이 추가되었다. 본 기술문서에서는 ONOS의 SBI를 위한 프로토콜로 LISP을 제안한다. LISP은 IP주소가 내포하고 있는 식별자와 라우팅 위치 정보를 분리하여 보다 나은 라우팅 방법과 인입트래픽 가공과 같은 기법을 적용할 수 있는 프로토콜이다. LISP는 SDN과 유사하게 제어 평면과 전송 평면을 가지고 있어 SBI에 적합한 프로토콜이다. 본 기술문서에서는 LISP을 위한 ONOS에 LISP 프로바이더 설계와 구현 방법 그리고 사용방법을 소개하도록 한다.

# Contents

## K-ONE #13. ONOS 상 LISP Provider 설계 및 구현

# 그림 목차

# K-ONE #13. ONOS 상 LISP Provider 설계 및 구현

# 1. 서론

본 서론에서는 이 기술문서를 이해하는데 필요한 배경지식으로 SDN과 ONOS에 대해서 간단하게 소개하고, ONOS에 SBI 확장이 필요한 이유를 설명하도록 한다. 그리고 마지막으로 ONOS의 SBI로 추가하고자 하는 LISP에 대해서 설명하도록 한다.

## 1.1. 소프트웨어 정의 네트워크 (Software Defined Network, SDN)

현재 인터넷은 스위치, 게이트웨이, 라우터 등의 수많은 네트워크 장비들이 계층 구조로 연결이 되어 있다. 이런 고정 토폴로지에서 네트워크 장비의 연결과 네트워크 성능향상을 위해 다양한 프로토콜들이 사용됨에 따라서 네트워크가 복잡해지게 되었다[1]. 이로 인해서 네트워크에 새로운 장비를 추가하거나 새로운 기술을 적용하는데 많은 어려움이 있다. 게다가 장비 제조사들의 긴 장비 생산 주기로 인해 새로운 기술이나 사용자의 요구에 빠르게 대응하지 하지 못하고 있다. 이런 현재 네트워크 환경에서 인터넷 사용 패턴의 변화와 시장의 요구에 빠르게 대응할 수 있는 네트워크 구조로써 SDN이 제안되었다. Open Networking Foundation (ONF)[2]에서는 SDN을 네트워크의 제어 평면을 전송 평면에서 물리적으로 분리하고 제어 평면이 다수의 장비를 관리하는 것으로 정의하고 있다. 즉, 기존 네트워크 장비들이 경로연산, 정책관리, 패킷처리 등 전송 평면과 제어 평면을 모두 수행하는 것과 다르게 SDN에서는 네트워크 장비가 전송 평면을 담당하고 중앙 집중 컨트롤러가 제어 평면을 담당한다.

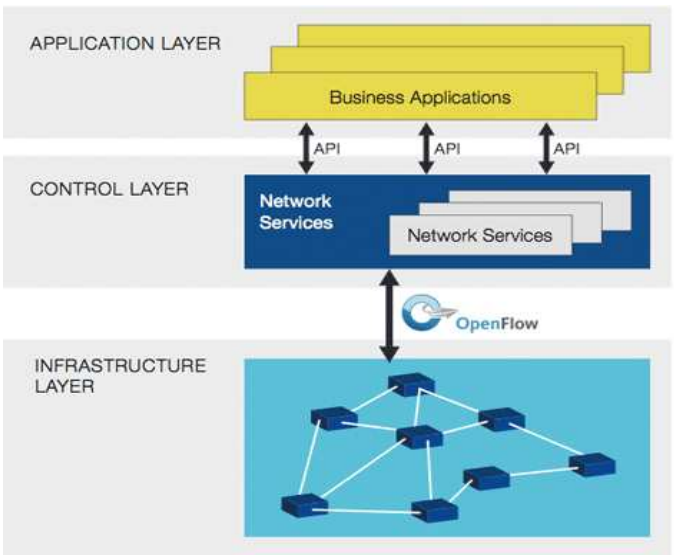


그림 1. 3-계층의 SDN 구조

그림 1은 SDN의 인프라스트럭처, 제어, 애플리케이션의 3-계층 구조를 나타내며 ONF에서 정의한 SDN의 개념을 잘 보여주고 있다. 인프라스트럭처 계층에는 물리적/논리적 네트워크 장치가 포함되며 실제 데이터 전송이 이뤄지는 전송 평면을 담당한다. 제어 계층은 이름 그대로 제어 평면을 담당하며 소프트웨어로 정의된 컨트롤러가 인프라스트럭처 계층에 존재하는 네트워크 장비들을 관리 및 제어한다. 컨트롤러가 네트워크 장비와 통신위해 사용하는 프로토콜들은 컨트롤러를 기준으로 아래쪽에 존재한다 하여 South-Bound Interface (SBI)라 부르며 일반적으로 OpenFlow[3]를 많이 사용하고 있다. 컨트롤러가 애플리케이션 개발을 위해 제공하는 API를 컨트롤러를 기준으로 위쪽이라 하여 North-Bound Interface (NBI)라고 하며 NBI를 이용하여 개발된 라우팅, 접근제어, QoS관리 등과 같은 SDN 애플리케이션들이 애플리케이션 계층에 해당된다.

SDN에서는 이러한 표준화된 인터페이스를 통해서 장비 제조사에 의존하지 않고 네트워크 장비를 제어할 수 있어 보다 유연하게 네트워크를 운용할 수 있다. 또한, SDN에서 네트워크 장비는 컨트롤러의 명령을 수행하는 단순한 구조를 가져 장비의 단가가 낮아지게 된다. 이러한 SDN의 특징으로 보다 손쉽게 네트워크를 구축 및 운용할 수 있다.

## 1.2. Open Network Operating System (ONOS)

기존 네트워크 장비에서 소프트웨어로 구현된 기능들을 SDN에서는 컨트롤러가 수행하며 컨트롤러가 다수의 네트워크 장비를 제어하기 때문에 컨트롤러에 대한 중요도가 매우 높다. 현재 많은 종류의 컨트롤러가 개발되고 있는데 최근에는 ONOS[4]와 OpenDayLight (ODL)[5]가 대규모의 네트워크에 적용 가능한 컨트롤러로써 많은 관심을 받고 있다.

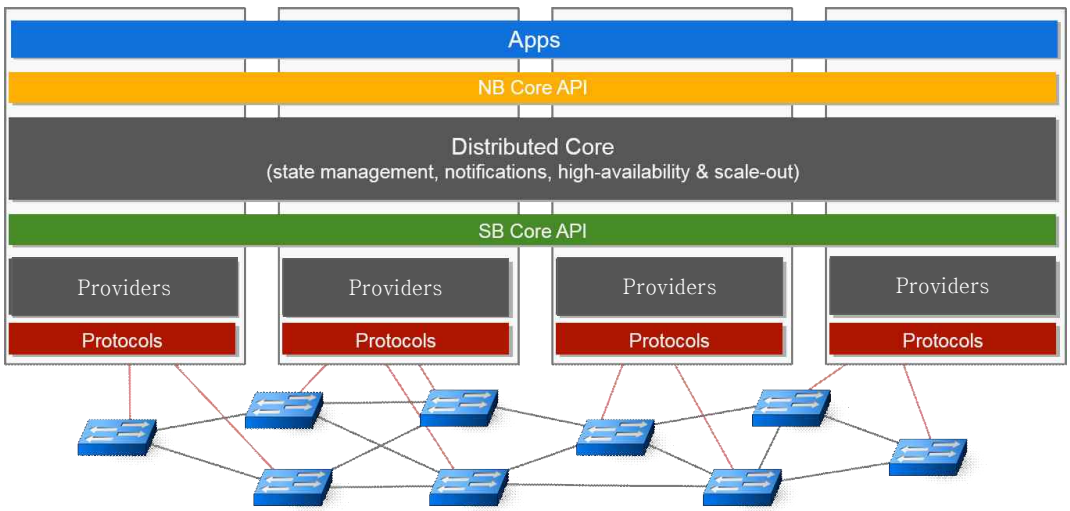


그림 2. ONOS의 분산 구조

ONOS 이전에는 다음과 같은 이유로 SDN을 캠퍼스나 작은 기업체와 같은 중규모의 네트워크에 적용이 가능한 기술로 평가되었다.

- 중앙 집중 구조에서 컨트롤러가 bottleneck이 되어 네트워크의 성능이 감소
- 다수의 컨트롤러를 사용하는 SDN에서 컨트롤러가 네트워크의 global view를 알지 못함

ONOS에서는 분산 구조를 이용하여 위의 문제를 해결하여 기존 컨트롤러보다 높은 네트워크 성능을 제공한다. 그림 2는 ONOS의 분산 구조를 나타내고 있다. ONOS는 다수의 ONOS 인스턴스(virtual machine)로 구성되며 각 인스턴스는 ONOS 컨트롤러로써 네트워크 장비를 제어한다. ONOS 인스턴스는 분산 저장소 (distributed store)을 이용하여 네트워크 정보를 공유하여 모든 ONOS 인스턴스가 global view를 볼 수 있게 도와준다. ONOS는 분산 구조를 이용하여 컨트롤러에 몰리는 제어 메시지를 분산시키면서도 네트워크의 global view를 가지고 네트워크 장치들을 제어할 수 있다.

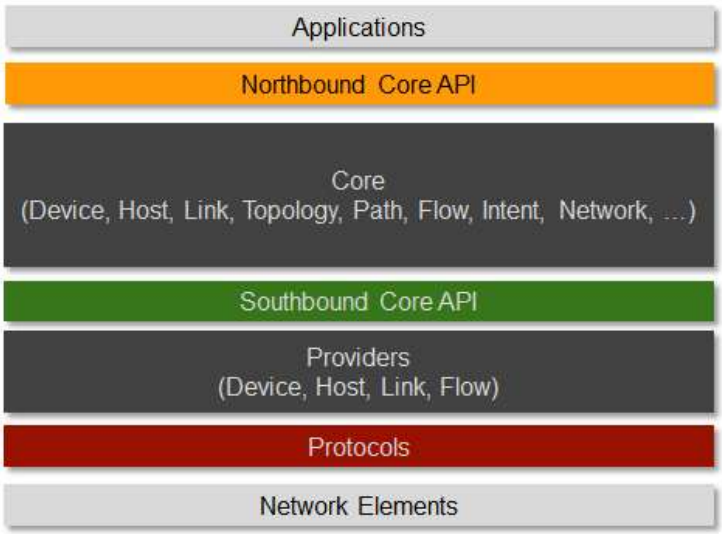


그림 3. ONOS의 계층 구조

ONOS는 계층 구조로 구현이 되어 있으며 각 계층은 1.1절에서 설명한 SDN의 계층 구조와 대응이 된다. ONOS의 핵심 기능을 담당하는 코어 영역은 SDN의 제어 영역과 매치되며 코어를 기준으로 ONOS 애플리케이션을 위한 North-Bound Core API과 네트워크 장비를 제어하기 위한 South-Bound Core API를 제공하고 있다. ONOS에서는 SBI를 프로토콜과 프로바이더 영역이 담당하고 있으며 표준화된 API를 통해서 SBI를 관리함에 따라서 플러그인 방식으로 새로운 SBI를 추가 할 수 있다.

## 1.3. ONOS SBI 확장의 필요성

일반적으로 SDN을 구성하기 위해서는 OpenFlow를 지원하는 SDN 스위치와 컨트롤러가 필요하다. 하지만 현재 인터넷을 구성하고 있는 기존 네트워크 장비들은 OpenFlow를 지원하지 않으므로, OpenFlow를 사용하는 SDN으로 인터넷을 대체하는데 문제가 있다. 따라서 기존 인터넷과 SDN이 공존하면서 SDN의 영역을 차츰차츰 넓혀감으로써 기존 인터넷을 SDN으로 교체할 수 있을 것이다. SDN의 영역을 넓히기 위해서 OpenFlow와 기존 IP 라우팅을 지원하는 스위치를 사용하거나 기존 네트워크 장비에 플러그인 혹은 펌웨어 업데이트로 OpenFlow를 지원하는 방법이 있다. 하지만 전자는 장비 교체에 비용에 많이 들고, 후자는 장비 제조사의 지원 여부가 중요하다. 두 가지 방법 모두 오랜 시간이 소요된다.

게다가 OpenFlow가 SDN을 위해서 디자인된 통신 프로토콜이긴 하지만 L2/L3 스위치에서 패킷 처리하는 것을 목표로 하고 있기 때문에 다양한 요구사항을 지원하기 위해서는 OpenFlow만으로는 한계가 있다. 이러한 문제를 해결하기 위해서 ODL은 SBI에 여러 프로토콜을 지원하고 있다. 일반적으로 SDN 컨트롤러는 OpenFlow를 유일한 SBI로 사용하는데 반해서 ODL은 OVSDB, NetConf, CAPWAP, BGP, PCEP 등 총 15종의 프로토콜을 지원함으로써 더 많은 네트워크 장비를 제어 할 수 있다[6]. 이에 반해, ONOS에서는 OpenFlow를 포함해 5 종의 프로토콜들만을 SBI로 제공하고 있다. ONOS도 네트워크의 다양한 장비들과 협업을 통해서 네트워크의 global view를 가질 필요가 있으므로 더 많은 프로토콜을 ONOS의 SBI로 확장할 필요성이 존재한다.

## 1.4. Locator/Identifier Separation Protocol (LISP)

앞 절에서 설명했듯이, SDN을 구성하기 위해서 OpenFlow만을 SBI로 제한할 필요는 없다. 이미 표준화된 프로토콜 중 SDN에 알맞은 프로토콜을 선택하여 ONOS의 SBI로 확장하는 것이 합리적이다. 본 기술문서에서는 LISP[7]을 ONOS의 SBI로 추가하고자 하고자 하며 이를 통해 네트워크에 존재하는 LISP 장비들을 ONOS로 제어하는 것을 목표로 하고 있다. LISP은 현재 TCP/IP 기반의 인터넷에서는 IP주소로 나타내는 라우팅 위치정보와 식별자를 분리하는 것을 골자로 하는 프로토콜이다. 모바일 기기의 경우 이동성에 의해 라우팅 위치가 바뀔 수 있는데, 처음 할당받은 IP주소를 가지고는 모바일 기기의 실제 위치에 맞춰서 전송 경로를 지정할 수 없는 문제가 있다. 또한, IoT의 시대가 시작됨에 따라서 미래에는 모든 장비들에게 IP 주소를 할당 해 줄 수 없을 것으로 예상되고 있다. LISP은 이러한 IP의 문제를 해결하며 라우팅의 확장성, IPv4와 IPv6 간의 주소체계 전환, 인입트래픽 가공, 단말의 모빌리티 지원 등을 제공 할 수 있다.

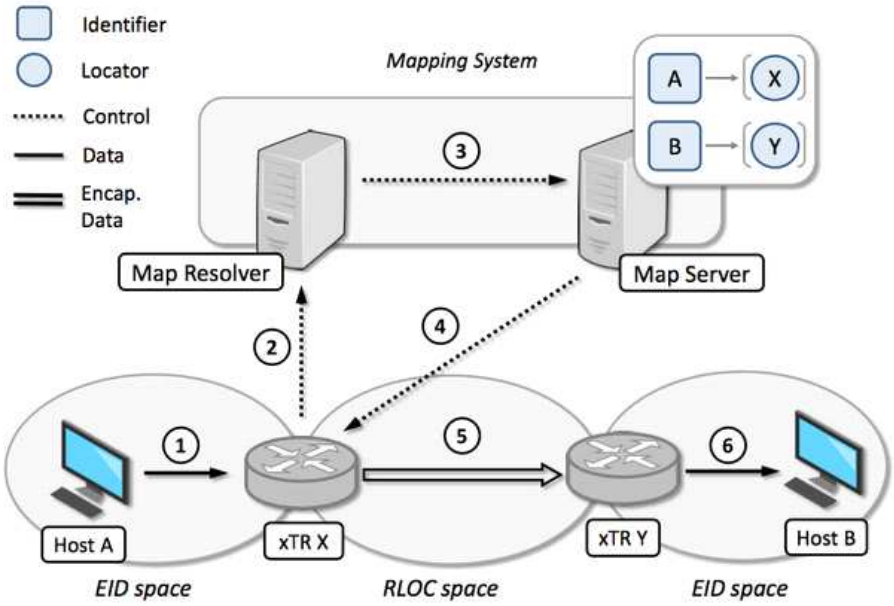


그림 4. LISP에서 패킷 전송 과정

LISP에서는 Endpoint ID(EID)와 Routing Locator(RLOC)의 두 개의 주소를 사용하여 식별자와 라우팅 정보를 나타낸다. 이러한 방법을 이용하여 LISP은 기존 IP주소 할당을 줄여서 라우팅 테이블 크기 또한 줄어들어 보다 효과적인 라우팅을 지원한다. EID는 종단 식별자를 나타내는 IP주소로 메시지의 종착지인 단말을 식별하기 위해 사용된다. 그리고 RLOC은 메시지 전송 시 사용되는 라우터의 주소를 나타낸다. 그림 3에서는 EID와 RLOC을 사용영역과 LISP의 동작 방식을 보여주고 있다. 호스트 A가 메시지를 전송할 때 xTR(LISP을 지원하는 라우터로 Ingress Tunnel Router, ITR 과 Egress Tunnel Router, ETR 역할을 모두 지원) 전까지는 EID를 이용하여 라우팅된다. 이후 두 xTR 사이에는 터널링을 통해서 메시지가 전송이 된다. 이때 xTR X가 xTR Y의 RLOC을 매핑 시스템에게 질의하는 과정을 거치게 된다. 결과적으로 xTR Y에 도달한 메시지는 호스트 B의 EID를 이용하여 호스트 B로 전송된다. LISP의 동작을 제어 평면과 전송 평면으로 구분할 수 있는다. LISP의 제어 평면은 메시지가 전송될 경로를 지정하며 xTR과 매핑 시스템간의 맵 정보 교환 (그림 3에서 2, 3, 4)이 해당된다. LISP의 전송 평면은 두 호스트 사이에서 실제로 데이터가 전송되는 것으로 그림 3의 1, 5, 6이 해당된다. LISP의 동작방식이 SDN과 매우 유사하며 ONOS에 LISP을 SBI로 추가하면 기존 LISP의 매핑 시스템의 역할을 대체하여 LISP 전송의 경로를 제어할 수 있다.

# 2. 본론

본 장은 ONOS의 SBI를 담당하는 프로바이더와 프로토콜 계층에 대해서 설명하고 LISP을 SBI로 추가하기 위해 필요한 LISP 프로바이더와 프로토콜의 구조와 구현 방법에 대해서 설명한다.

## 2.1. ONOS의 SBI

ONOS에서는 프로바이더와 프로토콜 계층이 SBI을 담당하고 있다. 프로토콜 계층은 네트워크 장비와 직접적인 통신을 하는데 필요한 기능들이 포함된다. 프로바이더 계층은 프로토콜과 코어 계층 사이의 중계역으로 프로토콜로부터 수집한 네트워크의 상태를 코어에게 추상화하여 제공하고, 코어로부터 네트워크 장비 제어 명령을 수신하면 이에 알맞은 제어 메시지를 프로토콜을 통해서 전송하는 역할을 한다.

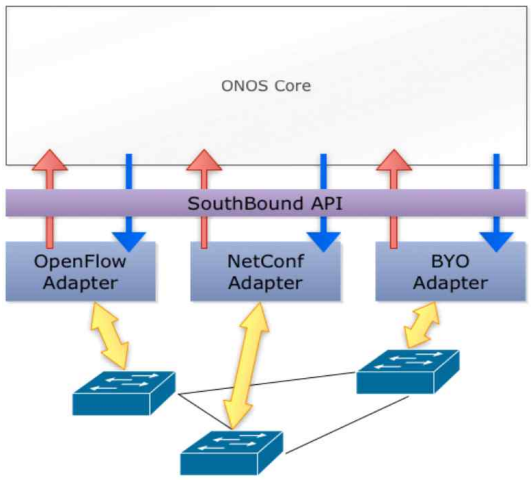


그림 5. 프로바이더의 역할 개념도

프로바이더는 ONOS의 표준화된 SB Core API를 이용함으로써 코어에서 여러 종류의 프로토콜에 대해서 알 필요 없이 네트워크 장비를 제어할 수 있다. 또한, 표준화된 API를 통해서 새로운 프로토콜을 ONOS의 SBI로 플러그인 형식으로 추가할 수 있다. 프로바이더는 ONOS에서 애플리케이션과 동일한 방법으로 취급하고 있다. 다시말해서, ONOS app을 install/uninstall 하듯이 프로바이더도 필요에 의해서 사용여부를 결정할 수 있다. 예를 들어, ONOS의 설정파일인 cell 파일이나 ONOS의 환경변수에 특정 프로바이더를 지정하면 ONOS가 시작할 때 프로바이더가 동작을 하게 할 수 있다.

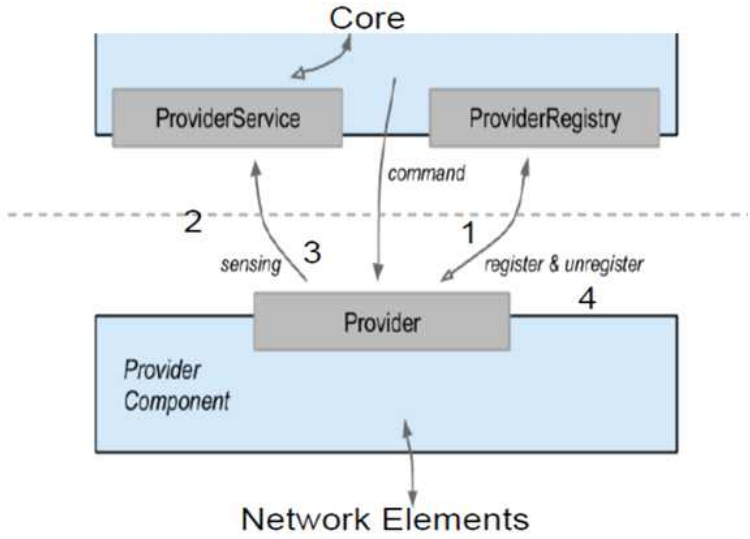


그림 6. 프로바이더의 기본 동작 패턴

프로바이더가 시작 될 때 (install 될 때), 코어의 ProviderRegistry의 register함수를 통해서 프로바이더의 시작을 알린다. 이후 코어는 ProviderService를 통해서 프로바이더로부터 이벤트 리스닝하게 된다. 코어는 이벤트를 처리하고 네트워크 장비의 동작을 변경할 필요가 있으면 프로바이더를 이용하여 네트워크 장비에 메시지를 보낸다. 그리고 프로바이더를 uninstall하면 코어에 unregister 함수를 통해서 해당 프로바이더가 더 이상 사용되지 않음을 알린다.

## 2.2. LISP 프로바이더 설계

서론에서 설명한 것과 같이 ONOS의 프로토콜 계층은 네트워크 장비와 통신하는 실제적인 역할을 하고 프로바이더 계층은 코어와 프로토콜 사이에서 중계자 역할을 한다. 이러한 구조로 인해 ONOS에서는 프로바이더 계층이 프로토콜 계층을 포함하여 SBI를 담당하는 것으로 나타낸다. 따라서 LISP을 ONOS의 SBI로 구현하기 위해서는 LISP을 위한 프로토콜과 프로바이더를 같이 구현하면 된다.

그림 7은 LISP 프로바이더와 프로토콜에 대한 구조와 메시지 처리를 위한 함수 호출을 나타내고 있다. 우선, LISP 프로바이더가 시작이 되면 LISP 장비와 매핑 정보 교환을 위한 통신 채널을 생성한다. 이후 채널로 메시지가 유입되면 LISP 프로토콜은 해당 메시지가 LISP 메시지인지 확인한 뒤, xTR에 대한 정보와 메시지를 LISP 프로바이더에게 전달한다. LISP 프로바이더는 프로토콜로부터 전달받은 메시지를 분석하여 맵 정보를 저장하고 xTR에게 응답 메시지를 전송하게 된다. 본 기술문서에서 제안하는 LISP 프로바이더와 프로토콜의 설계는 ONOS의 설계 구조에 따르지 않고 있다. ONOS에서는 네트워크 제어를 위한 지능은 코어가 담당하므로 LISP 메시지에 포함된 매핑 정보와 메시지 응답에 대한 결정은 코어에 구현이 되어야 한

다. 하지만 본 기술문서에서 LISP을 위한 프로바이더와 프로토콜을 목적으로 하고 있어 현재 코어에 대한 고려는 되어 있지 않다. 또한, LISP 프로바이더의 동작을 확인하기 위한 목적으로 LISP 프로바이더가 메시지를 처리하고 응답을 하도록 설계를 하였다. 이는 차후에 상위 계층을 고려한 ONOS에서 LISP 프로바이더에서는 변경이 될 예정이다.

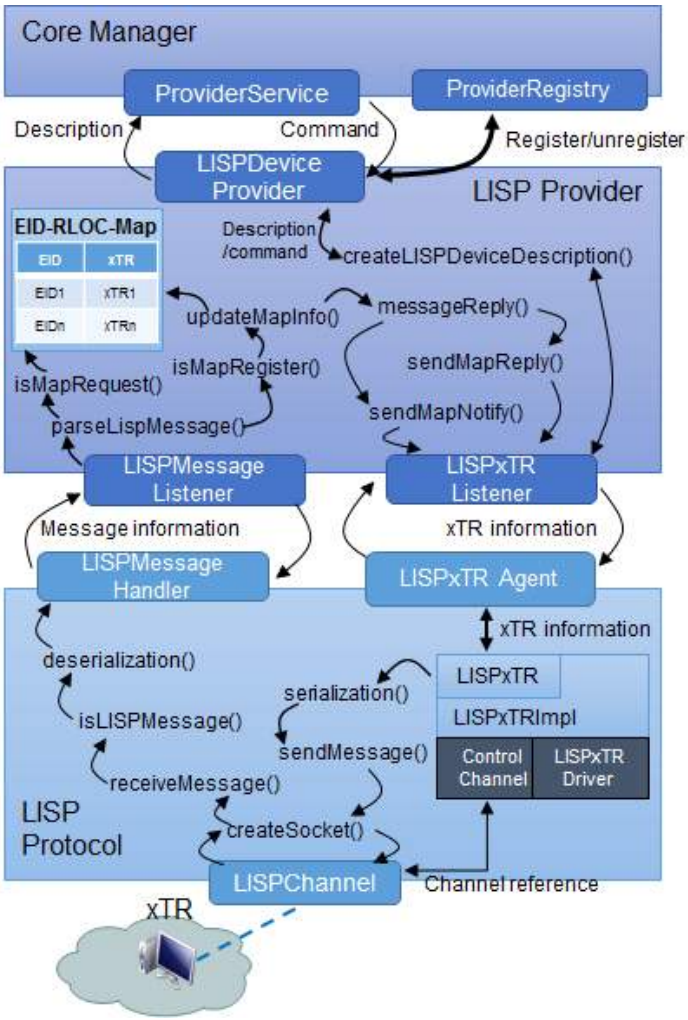


그림 7. LISP을 위한 프로토콜 및 프로바이더

## 2.3. LISP 프로바이더 구현

본 기술문서에서는 ONOS 버전 1.4를 기준으로 LISP 프로바이더와 프로토콜을 구현하였다. 구현을 위해서 ONOS의 Provider tutorial Wiki[8]과 ONOS에 구현되어 있는 다른 SBI 프로토콜 소스코드를 참고하였다.

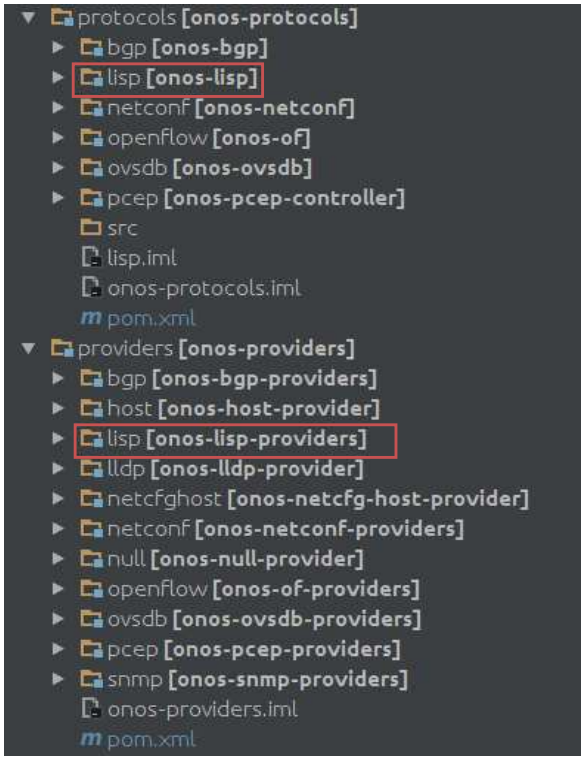


그림 8. LISP 프로바이더와 프로토콜 경로

ONOS의 소스코드를 다운받으면 protocols와 providers라는 디렉토리가 존재하는데 그림 8과 같이 해당 디렉토리에 LISP을 위한 디렉토리를 생성한다. 이때 각 디렉토리에 있는 pom파일에 작성하여 LISP 프로토콜과 프로바이더가 사용됨을 명시해야 한다. ONOS Wiki를 보면 프로바이더를 위한 pom 파일 작성과 함께 프로바이적 작성을 위한 가이드가 존재한다[8]. LISP 프로바이더를 예를들면 providers 디렉토리의 pom 파일에 lisp 프로바이더가 추가되었음을 명시한다. lisp 디렉토리에 구현하고자 하는 프로바이더들을 pom파일로 명시한다. 마지막으로 구현하고자 하는 LISP 프로바이더의 디렉토리에서 pom파일 작성하여 준다. 유사한 방법으로 프로토콜을 위한 pom 파일을 작성한다.

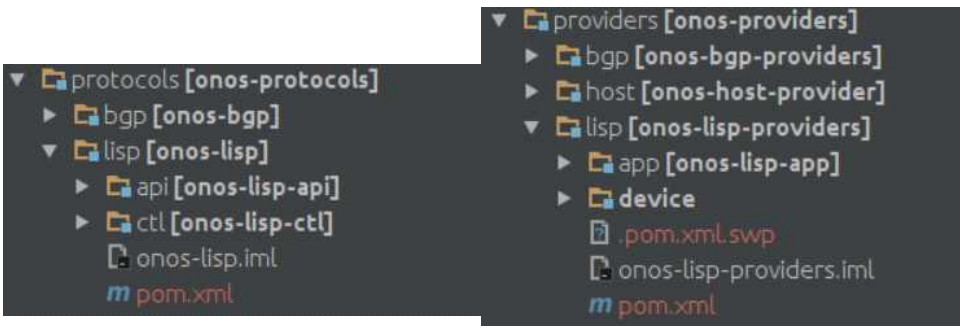


그림 9. LISP 프로바이더와 프로토콜의 소스트리

LISP 프로토콜을 LISP 프로바이더가 조작할 수 있는 API와 네트워크 장비와 통신을 위한 채널 및 메시지 송수신을 기능이 구현되어 있다. 프로바이더에서는 LISP 프로토콜로부터 받은 정보를 통해서 LISP 장비에 대한 추상화를 제공한다.

```
70      public void run() {
71          try {
72              ConnectionlessBootstrap udpBootstrap =
73                      new ConnectionlessBootstrap(new NioDatagramChannelFactory());
74              cg = new DefaultChannelGroup();
75              udpBootstrap.setPipelineFactory(new LispUDPPipelineFactory(cg));
76              udpBootstrap.bind(new InetSocketAddress(lispPort));
77
78              log.info("Listening port " + lispPort);
79          } catch (Exception e) {
80              throw new RuntimeException(e);
81          }
82      }
```

그림 10. LISP 장비와 통신을 위한 채널 생성

LISP 프로바이더를 시작하면 LISP 프로토콜은 LISP 장비와 통신을 위해서 포트 4342로 UDP 소켓을 생성하고 LISP 메시지를 기다린다. ONOS가 LISP 메시지를 수신하면 LISP 메시지의 타입을 확인하고 이에 따른 응답을 하게 된다.

```
55          byte tmp = buf.readByte();
56          int type = (int) (tmp >> 4);
57
58          if (type == 1) {
59              log.info("Map request");
60              ByteBuffer reply = handleMapRequest(bb);
61              e.getChannel().write(reply.array(), e.getRemoteAddress());
62          } else if (type == 3) {
63              log.info("Map register");
64              ByteBuffer reply = handleMapRegister(bb);
65              e.getChannel().write(reply.array(), e.getRemoteAddress());
66
67              ByteBuffer notify = handleMapRegister(bb);
68              e.getChannel().write(reply.array(), e.getRemoteAddress());
69          } else if (type == 7) {
70              log.info("Map 7");
71              ByteBuffer reply = handleInfoMsg(bb);
72              e.getChannel().write(reply.array(), e.getRemoteAddress());
73          } else if (type == 8) {
74              log.info("Map 8");
75              ByteBuffer reply = handleECM(bb);
76              e.getChannel().write(reply.array(), e.getRemoteAddress());
77          }
```

그림 11. 수신한 LISP 메시지 처리

본 기술문서에서는 LISP에서 매핑 정보를 교환을 위한 4개의 기본 메시지 타입인 Map-register, Map-notify, Map-request 그리고 Map-reply의 처리에 대해서 다루고 있다. 그림 11은 각 메시지를 타입에 따라서 처리하기 위한 코드이다. LISP의 동작 방식을 보면 Map-notify은 Map-register의 응답이고 Map-reply은Map-request의 응

답으로 사용된다. 따라서, LISP의 매핑 시스템으로는 인입되는 메시지는 Map-register와 Map-request이며 수신한 메시지 타입에 따라서 응답 메시지가 결정된다. 우선, Map-register와 Map-notify 메시지에 대해서 보면 다음과 같다.

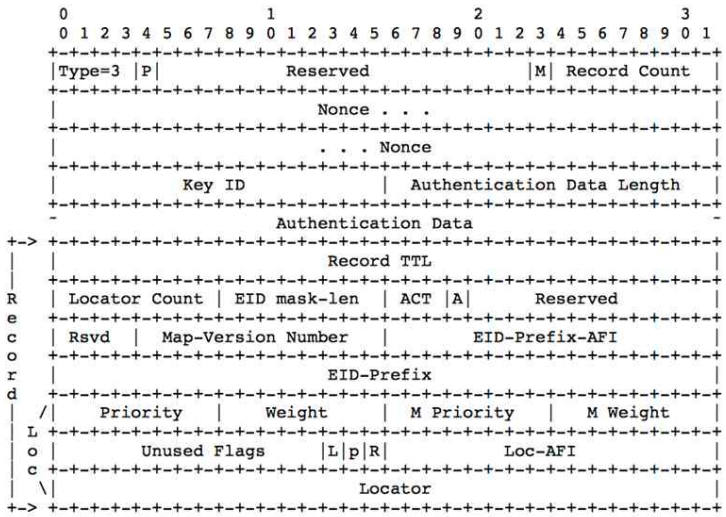


그림 12. Map-register 메시지 포맷

Map-register는 EID에 메시지를 전송할 때 사용가능한 라우터 (xTR)의 정보를 매핑 시스템에 등록하기 위해서 사용된다. EID-RLOC 매핑 정보를 저장단위로 하는 레코드를 전송한다. LISP에서는 8 바이트의 Nonce를 이용하여 메시지의 유효성을 확인하는데 메시지 전송 시 사용한 Nonce 값이 들어있는 응답 메시지만 유효한 메시지로 판단하고 수신하게 된다. 매핑 시스템은 Map-register를 수신하면 메시지에 포함되어 있는 레코드를 EID-RLOC 맵 테이블에 저장한 뒤, Map-notify를 되돌려 보낸다.

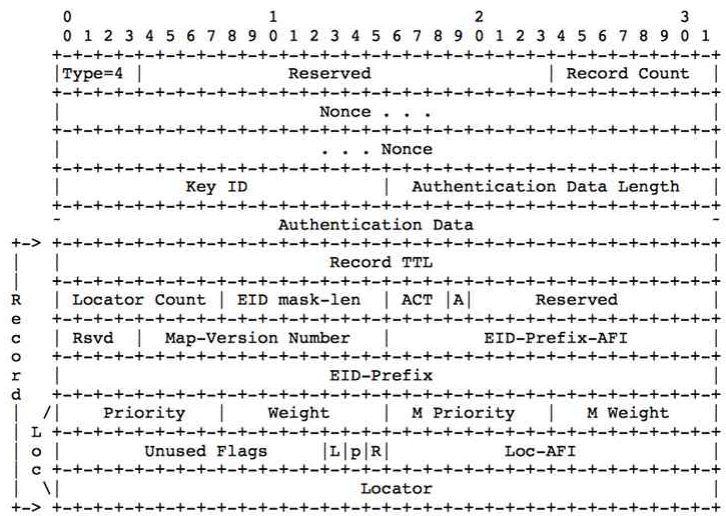


그림 13. Map-notify 메시지 포맷

Map-notify는 Map-register와 거의 똑같으며 수신한 Map-register의 내용을 복사하여 손쉽게 Map-notify를 생성할 수 있다. 생성한 메시지를 Map-register를 보낸 LISP 장비에게 되돌려 보내서 매핑 시스템에 매핑 정보가 정상적으로 등록되었음을 알린다.

```
29    public LispMapRegister deserialize(ByteBuffer registerBuffer) {
30        try {
31            LispMapRegister msg = new LispMapRegister();
32            registerBuffer.position(0);
33            msg.setProxyMapReply(ByteUtil.extractBit(registerBuffer.get(), proxy));
34            registerBuffer.position(registerBuffer.position() + res);
35            msg.setWantMapNotify(ByteUtil.extractBit(registerBuffer.get(), wantMapReply));
36
37            byte recordCount = (byte) ByteUtil.getUnsignedByte(registerBuffer);
38            msg.setNonce(registerBuffer.getLong());
39            msg.setKeyId(registerBuffer.getShort());
40            short authenticationLength = registerBuffer.getShort();
41            byte[] authenticationData = new byte[authenticationLength];
42            registerBuffer.get(authenticationData);
43            msg.setAuthenticationData(authenticationData);
44
45            for (int i = 0; i < recordCount; i++) {
46                msg.getEidToLocatorRecords()
47                        .add(EidToLocatorRecordSerializer
48                                .getInstance().deserialize(registerBuffer));
49            }
50            return msg;
51        } catch (RuntimeException e) {
52            log.info("Invalid map register");
53        }
54        return null;
55    }
```

그림 14. Map-register 메시지 처리 코드

그림 14는 LISP 프로바이더에서 Map-register 메시지를 처리하는 코드로 메시지를 분석한 뒤 Map-notify 메시지를 생성하여 돌려보내게 된다. Map-request은 LISP 장비에서 메시지 전송에 필요한 라우팅 위치 정보를 요청하기 위해서 사용하며 메시지 포맷은 그림 15와 같다.

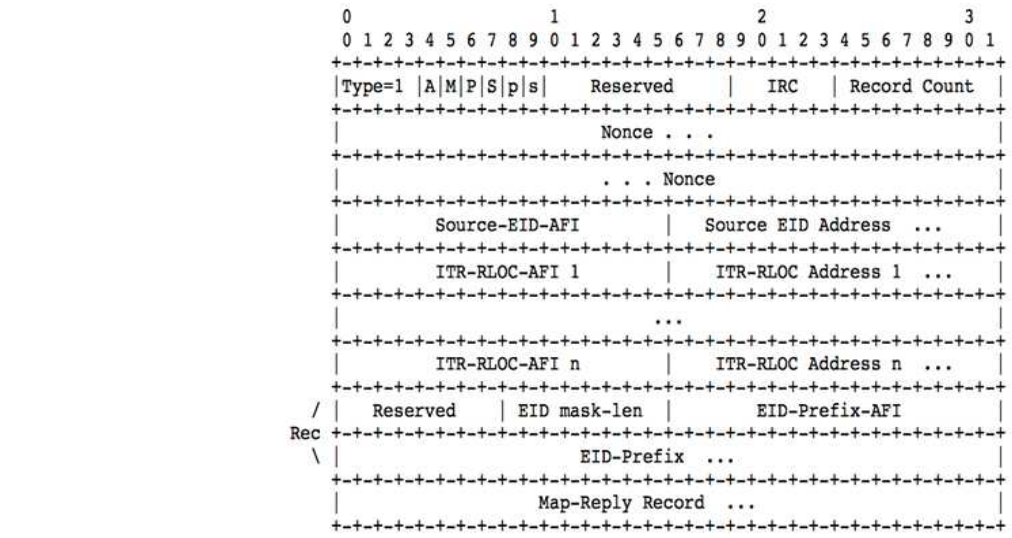


그림 15. Map-request 메시지 포맷

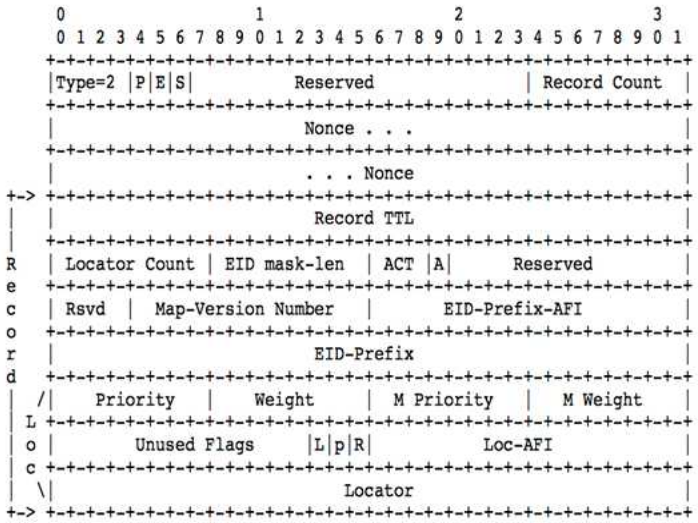


그림 16. Map-reply 메시지 포맷

Map-request에 대한 응답으로 Map-reply를 사용하는데 EID-RLOC 맵 정보중에서 Map-request를 통해서 질의한 EID에 도달할 수 있는 RLOC의 정보를 실어서 보내게 된다. 그림 17과 18은 LISP 프로바이더에서 Map-request 메시지를 수신하면 메시지의 내용을 분석하고 이에 따른 Map-reply 메시지를 생성하여 보내는 코드를 나타낸다.

```
91    private ByteBuffer handleMapRequest(ByteBuffer buffer) {
92        LispMapRequest msg = LispMapRequestSerializer.getInstance().deserialize(buffer);
93        ByteBuffer reply = LispMapReplyGenerator.getInstance().generateMapReply(msg);
94        return reply;
95    }
```

그림 17. Map-request 메시지 분석 및 Map-reply 메시지 생성 함수 호출

```
21    public ByteBuffer generateMapReply(LispMapRequest msg) {
22        ByteBuffer replyBuffer = ByteBuffer.allocate(40);
23
24        replyBuffer.position(0);
25        replyBuffer.put((byte) ((byte) (2 << 4) | 0x08));
26        replyBuffer.putShort((short) 0);
27        replyBuffer.put((byte) 1);
28
29        replyBuffer.putLong(msg.getNonce());
30
31        replyBuffer.putInt(60);
32
33        replyBuffer.put((byte) 1);
34        replyBuffer.put((byte) 16);
35        replyBuffer.putShort((short) 0);
36
37        replyBuffer.putShort((short) 0);
38        replyBuffer.putShort((short) 1);
39
40        try {
41            replyBuffer.put(Inet4Address.getByName("192.168.0.1").getAddress());
42        } catch (UnknownHostException e) {
43            // TODO Auto-generated catch block
44            e.printStackTrace();
45        }
```

```
46
47          replyBuffer.put((byte) 1);
48          replyBuffer.put((byte) 100);
49          replyBuffer.put((byte) 1);
50          replyBuffer.put((byte) 100);
51
52          replyBuffer.putShort((short) 0);
53          replyBuffer.putShort((short) 1);
54
55          try {
56              replyBuffer.put(Inet4Address.getByName("13.0.0.101").getAddress());
57          } catch (UnknownHostException e) {
58              // TODO Auto-generated catch block
59              e.printStackTrace();
60          }
61
62          return replyBuffer;
63      }
```

그림 19. Map-reply 생성 코드

구현한 LISP 프로바이더와 LISP 프로토콜을 빌드하고 나면 ONOS에서 사용할 수 있게 된다. ONOS의 CLI에서 "feature:install onos-lisp-app" 명령으로 LISP 프로바이더를 실행할 수 있다. LISP 프로바이더가 동작 중인지 확인하기 위해서 "list" 명령을 이용하면되며 그림 20은 LISP 프로바이더가 ONOS에서 제대로 동작 중임을 보여준다.

```
179 | Active |   80 | 1.4.1.SNAPSHOT    | onos-host-provider
180 | Active |   80 | 1.4.1.SNAPSHOT    | onos-lldp-provider
181 | Active |   80 | 1.4.1.SNAPSHOT    | onos-openflow
182 | Active |   80 | 1.4.1.SNAPSHOT    | onos-lisp-api
183 | Active |   80 | 1.4.1.SNAPSHOT    | onos-lisp-ctl
184 | Active |   80 | 1.4.1.SNAPSHOT    | onos-lisp-device-provider
onos>
```

그림 20. ONOS에서 LISP 프로바이더 실행

ONOS에서는 Karaf[9]의 로그 시스템을 이용하고 있으며 tl 툴을 이용하면 로그를 확인할 수 있다. 그림 21은 LISP 프로바이더 실행 후 로그 내용으로 LISP 프로바이더가 ONOS에 설치가 되었고 동작하고 있다는 것을 나타내고 있다. 로그 시스템을 이용하여 ONOS 개발에 필요한 정보를 확인 할 수 있다.

```
2016-05-08 17:01:43,225 | INFO  | l for user karaf | Controller
      | 183 - org.onosproject.onos-lisp-ctl - 1.4.1.SNAPSHOT | Started
2016-05-08 17:01:43,225 | INFO  | l for user karaf | Controller
      | 183 - org.onosproject.onos-lisp-ctl - 1.4.1.SNAPSHOT | init
2016-05-08 17:01:43,240 | INFO  | l for user karaf | Controller
      | 183 - org.onosproject.onos-lisp-ctl - 1.4.1.SNAPSHOT | Listening port 43
42
2016-05-08 17:01:43,240 | INFO  | l for user karaf | LispControllerImpl
      | 183 - org.onosproject.onos-lisp-ctl - 1.4.1.SNAPSHOT | Started
2016-05-08 17:01:43,243 | INFO  | l for user karaf | LispDeviceProvider
      | 184 - org.onosproject.onos-lisp-device-provider - 1.4.1.SNAPSHOT | LISP
provider started
```

그림 21. LISP 프로바이더 실행 시 로그 내용

# 3. 결론

SDN에 대한 관심이 높아짐과 더불어 기존 네트워크를 SDN으로 교체하기 위한 노력이 계속되고 있다. ONOS는 다른 컨트롤러들에 비해서 높은 가용성과 네트워크 속도를 보장하여 캐리어 레벨의 네트워크를 SDN으로 운용할 수 있을 것으로 기대되고 있다. 하지만 기존 인터넷을 SDN으로 대체하기 위해서는 오랜 시간이 소요될 것으로 보이며 현재 설치되어 있는 네트워크 장비를 모두 SDN 장비로 교체하기에는 무리가 따른다. 따라서 기존 인터넷과 SDN이 공존하면서 차츰차츰 SDN의 영역을 넓혀가면서 대체해 가야 할 것으로 보인다. 장비 제조사에서는 OpenFlow와 기존 IP 라우팅을 지원하는 장비를 선보이거나 펌웨어 업데이트를 통해서 OpenFlow를 지원하고 있는 추세이다. 이러한 인프라스트럭쳐 영역에서의 노력뿐만 아니라 SDN 컨트롤러에서도 기존 인터넷에서 사용하는 제어 프로토콜을 지원하여 SDN의 지원 범위를 넓혀 가고 있다. 본 기술문서에서는 상대적으로 적은 수의 프로토콜을 SBI로 지원하고 있는 ONOS에 LISP을 추가 하는 것을 목적으로 하였으며, 구현을 위해 설계한 프로바이더와 프로토콜 계층에 대한 설명과 현재까지 구현된 LISP 프로토콜에 대한 소개를 하였다. 현재 개발된 LISP 프로바이더는 LISP에서 매핑 정보를 교환하기 위한 4개의 메시지 처리를 중심적으로 다루었고, 동작을 확인하기 위해서 ONOS의 구조에 알맞지 않은 설계를 이용하였다. ONOS에 LISP 서브시스템을 구현하는데 알맞게 현재 LISP 프로바이더의 구조를 수정하도록 할 예정이다.

## References

[1] 유재형, 김우성, 윤찬형, "SDN/OpenFlow기술 동향 및 전망," KNOM Review, Vol. 15, No. 2, Dec. 2012
[2] ONF, https://www.opennetworking.org
[3] OpenFlow, https://www.opennetworking.org/sdn-resources/openflow
[4] ONOS, http://onosproject.org/
[5] ODL, https://www.opendaylight.org/
[6] ODL features list, https://www.opendaylight.org/opendaylight-features-list
[7] LISP, D. Farinacci *et al.*, "The Locator/ID Separation Protocol (LISP)," IETF RFC 6830, Oct, 2015
[8] Provider tutorial, https://wiki.onosproject.org/display/ONOS/Provider+Tutorial
[9] Apache Karaf, http://karaf.apache.org/

# *K-ONE 기술 문서*



작성기관: K-ONE Consortium
작성년월: 2016/05