K-ONE 기술 문서 #30

MAS-Manv2: Mastership and Auto-Scaling Management System Version 2.0

Document No. K-ONE #30
Version 2.1
Date 2017-03-02
Author(s) 김우중

K-ONE

ONK
OpenNetworkingKorea

■ 문서의 연혁

버전	날짜	작성자	비고
초안 - 0.1	2016.12.01	김우중	개요부분
0.2	2016.12.25	김우중	설계부분
0.3	2017.01.15	김우중	구현부분
0.4	2017.03.05	김우중	데모부분
1.0	2017.04.30	김우중	초안작성
1.1	2017.05.12	김우중	교정
2.0	2018.02.28	김우중	초안작성
2.1	2018.03.02	김우중	교정

본 문서는 2017년도 정부(과학기술정보통신부)의 재원으로 정보통신 기술진흥센터의 지원을 받아 수행된 연구임 (No. 2015-0-00575, 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(MSIT) (No. 2015-0-00575, Global SDN/NFV OpenSource Software Core Module/Function Development)

기술문서 요약

본 기술문서에서는 현재 널리 사용되고 있는 분산형 컨트롤러에 대한 개념에 대해 살펴보고, 분산형 컨트롤러에서 발생하는 컨트롤러 배치 문제에 대하여 살펴본다. 또한, 본 기술문서에서는 기존에 이러한 문제를 해결하기 위한 본 연구진이 제안 및 개발한 효율적 마스터쉽 제어를 위한 소프트웨어의 설계 구현을 서술하였다. 특히 이 소프트웨어는 기존 기술문서(기술문서#23)의 성능적 문제를 해결하고, 마스터쉽뿐만 아니라 스케일링까지 제공한다. 본 소프트웨어는 관리자의 편의를 위해 커맨드 라인 인터페이스를 제공하고, 이를 기반으로 모니터링, 마스터쉽, 그리고 스케일링을 수행하는 기능을 구현하였다.

Contents

K-ONE #30. MAS-Manv2: Mastership and Auto-Scaling Management System Version 2.0

1. 개요 ····· 6
2. 기존에 제안된 마스터쉽 제어/관리 기술 ····· 10
3. 기존에 제안된 마스터쉽 제어/관리를 위한 소프트웨어 ····· 12
4. MAS-Manv2 ····· 32
5. 성능 ····· 36
6. 결론 ····· 39

그림 목차

그림 1. SDN의 구조 ····· 6
그림 2. 초기 SDN의 개념도 ····· 7
그림 3. 분산 SDN의 개념도 ····· 7
그림 4. 컨트롤러 배치 문제의 개념도 ····· 8
그림 5. ONOS 컨트롤러의 마스터쉽 예시도 ····· 10
그림 6. 오케스트레이터의 구조 ····· 11
그림 7. 오케스트레이터의 코어 기능/모듈 계층에 대한 개념도 ····· 12
그림 8. 오케스트레이터의 패키징 정보를 도식화한 개념도 ····· 14
그림 9. Dispatcher-servlet.xml의 내용 ····· 15
그림 10. Spring-context.xml의 내용 ····· 15
그림 11. GeneralConf.java 파일의 내용 일부 ····· 16
그림 12. GeneralConf.java 내의 포스트 생성자 ····· 16
그림 13. PMBean.java 내용 ····· 17
그림 14. VMBean.java 내용 ····· 18
그림 15. SWBean.java 내용 ····· 18
그림 16. DBBean.java 내용 ····· 19
그림 17. PMConfPool.java 내용 ····· 19
그림 18. VMConfPool.java 내용 ····· 19
그림 19. RESTConnectionUtil.java 중 REST API로 정보를 요청하는 코드 ····· 20
그림 20. RESTConnectionUtil.java 중 REST API로 정보를 전송하는 코드 ····· 20
그림 21. SSHConnectionUtil.java 내용 ····· 21
그림 22. ControlTrafficBean.java 내용 ····· 22
그림 23. CPULoadBean.java 내용 ····· 22
그림 24. NumSwitchesBean.java 내용 ····· 22
그림 25. GetControlTraffic.java 내용 ····· 23
그림 26. GetCPULoad.java 내용 ····· 24
그림 27. GetNumSwitches.java 내용 ····· 24
그림 28. PutControlTraffic.java의 쿼리 내용 ····· 24
그림 29. PutCPULoad.java의 쿼리 내용 ····· 24
그림 30. PutNumSwitches.java의 쿼리 내용 ····· 25
그림 31. ChangeMastership.java의 chageMastership 함수 내용 ····· 25
그림 32. GettingCPTraffic.java 내용 ····· 26

그림 33. GettingCPULoad.java 내용 ····· 27
그림 34. top툴에서 파싱하는 CPU 부하 ····· 27
그림 35. GettingTopology.java 내용 ····· 28
그림 36. RESTParser.java의 예시함수 ····· 28
그림 37. MastershipAndScalingThread.java 내용 ····· 30
그림 38. Design of MAS-Manv2 ····· 34
그림 39. 모니터링 결과 ····· 38
그림 40. MAS-Manv2의 성능 평가 결과 ····· 39

K-ONE #30. MAS-Manv2: Mastership and Auto-Scaling Management System Version 2.0

1. 개요

■ 최근 정보통신 기술의 발전으로 인해, 사용자들이 다양한 스마트 장비 및 컴퓨터를 활용하여 다양한 서비스를 제공받고 있다. 이러한 서비스는 기존에 사용되었던 인터넷 웹 서핑, 네트워크 게임, 음악 스트리밍 서비스, 방송, VoIP(Voice over IP)등이 있는데, 이들 서비스는 기본적으로 인터넷과 같은 네트워크에 연결이 되어야 한다. 그리고 위의 서비스들의 품질은 시간이 흐를수록 점차 증가하여, 네트워크의 규모가 비약적으로 증가하고 있는 추세이다.

■ 네트워크 규모가 증가함에 따라, 인터넷 서비스 제공업체(ISP: Internet Service Provider)와 같은 회사는 네트워크 장비의 제어 및 관리를 위해 네트워크 관리 시스템(NMS: Network Management System)을 도입하였다. 네트워크 관리 시스템은 네트워크 장비를 제어/관리할 수 있는 컨트롤러와, 컨트롤러와 네트워크 장비 사이의 통신을 위한 제어 통신 프로토콜, 그리고 관리자가 네트워크 제어/관리를 위한 어플리케이션으로 구성되어 있다. 이를 통해, 네트워크 운영자는 네트워크 관리 시스템을 통해, 대규모의 네트워크를 제어/관리할 수 있다.

■ 기존의 네트워크 관리 시스템의 경우, 네트워크 운영자는 많은 금전적, 시간적 비용을 소모해야 했다. 먼저 네트워크 관리 시스템은 다양한 솔루션 제조자(solution vendor)들에게 개발이 되었다. 네트워크 운영자는 이들 제조사들에게 막대한 양의 금전적 비용을 지불하여, 네트워크 관리 시스템을 도입해야 했다. 이때, 기존 네트워크 관리 시스템은 장비 종속성(vendor-dependency)이 존재하는 문제가 있었다. 즉, 네트워크 운영자가 특정 제조사의 네트워크 관리 시스템을 구매하고자 한다면, 이 네트워크 관리 시스템에 제어/관리를 받을 수 있는 특정 네트워크 장비만을 사용해야 했다. 한편, 네트워크 운영자는 운영하는 네트워크의 효율적 제어/관리를 위해 새로운 기능 및 어플리케이션이 필요로 할 수 있다. 이 경우, 네트워크 운영자는 네트워크 관리 시스템을 개발한 제조사에 새로운 기능을 요청해야 한다. 이 때에도, 네트워크 운영자는 제조사에 막대한 비용을 지불해야 함은 물론, 개발 완료시까지 기다려야 하므로 시간적 비용을 지출하게 되는 문제가 있다.

■ 이러한 문제점을 해결하기 위해, SDN(Software Defined Networking, 소프트웨어 정의 네트워킹)[1]-[3]이라는 개념이 등장한다. SDN이란 용어 그대로 소프트웨어를 사용하여 네트워크를 정의하고 이를 제어/관리하는 개념을 의미한다. SDN은 기존 제조사들에 종속적으로 네트워크를 구성하는 것과는 달리, 어떠한 네트워크 장비를 사용한다 하더라도, 네트워크 운영자가 제어/관리할 수 있도록 기능을 제공한다.

■ SDN은 그림 1과 같이 세 개의 개념적인 계층으로 구분될 수 있다[1]. 먼저 네트워크 인프라스트럭쳐(infrastructure) 계층은 스위치, 라우터 등과 같은 다양한 네트워크 장비들이 위치하는 계층이다. 이 때, 각 네트워킹 장비의 지능(intelligence)은 대부분 상위 계층인 제어(control) 계층으로 집중되어 있으며, 네트워킹 장비는 단순 데이터 전달이나 최소한의 기능만을 가지고 있다. 한편, 제어 계층은 네트워크 내에 존재하는 다양한 네트워크 장비들을 제어/관리하는 계층이다. 특히 네트워크를 제어/관리하기 위한 SDN 컨트롤러가 이 계층에 해당된다. 마지막으로 어플리케이션(application) 계층은 네트워크 운영자가 네트워크를 소프트웨어로 제어/관리하기 위한 다양한 어플리케이션(e.g., 비즈니스 어플리케이션, 네트워크 관리 어플리케이션, 모니터링 어플리케이션 등)들이 위치하는 계층이다.

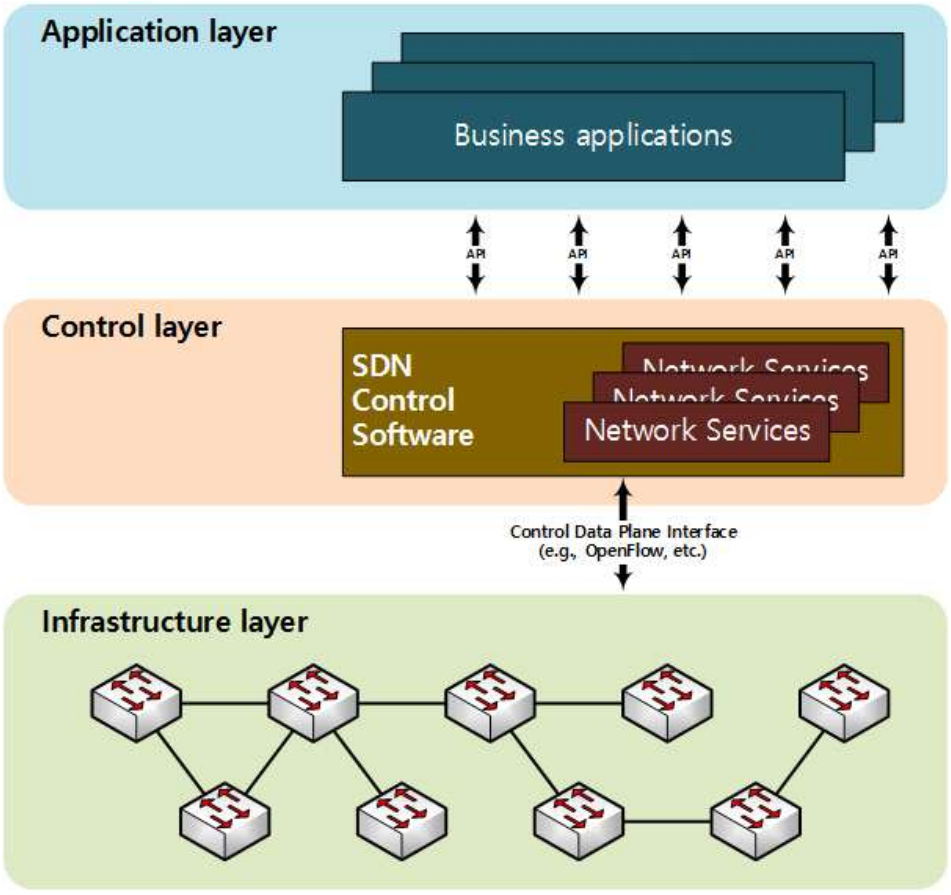


그림 1. SDN의 구조

■ 초기의 SDN 개념은 하나의 SDN 컨트롤러가 모든 네트워크 장비를 제어/관리하는 개념에서 시작하였다. 그림 2는 이를 도식화 한 개념도이다. 그림 2에서 보면, 총 9개의 스위치가 하나의 SDN 컨트롤러에게 제어/관리를 받고 있다. 만일 네트워크 내의 장비 수가 매우 적거나, SDN 컨트롤러 하나가 모두 처리할 컴퓨

팅 및 네트워킹 자원을 보유하고 있다면, 효율적으로 네트워크 운영자가 네트워크를 제어/관리할 수 있다. 하지만 네트워크 내의 장비 수가 점차 늘어나거나, 제어 트래픽의 수가 증가하게 된다면, 병목현상으로 인해 SDN 컨트롤러의 부하가 집중되어 네트워크의 성능이 저하될 것이다.

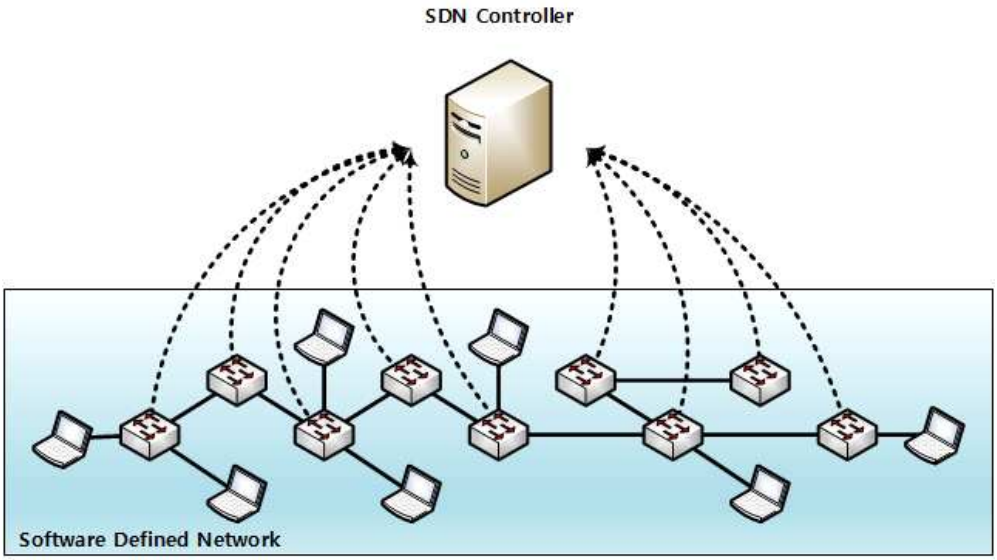


그림 2. 초기 SDN의 개념도

■ 이러한 문제를 해결하기 위해, ONOS(Open Network Operating System)[4]와 같은 분산형 SDN 컨트롤러의 개념이 등장한다. 분산형 SDN 컨트롤러란, 다수의 SDN 컨트롤러를 사용하여 네트워크를 제어/관리하는 개념이다[6]. 이를 사용하면, 초기의 SDN 개념에서 발생하는 병목현상을 해결할 수 있다. 그림 3은 분산형 SDN의 개념도이다.

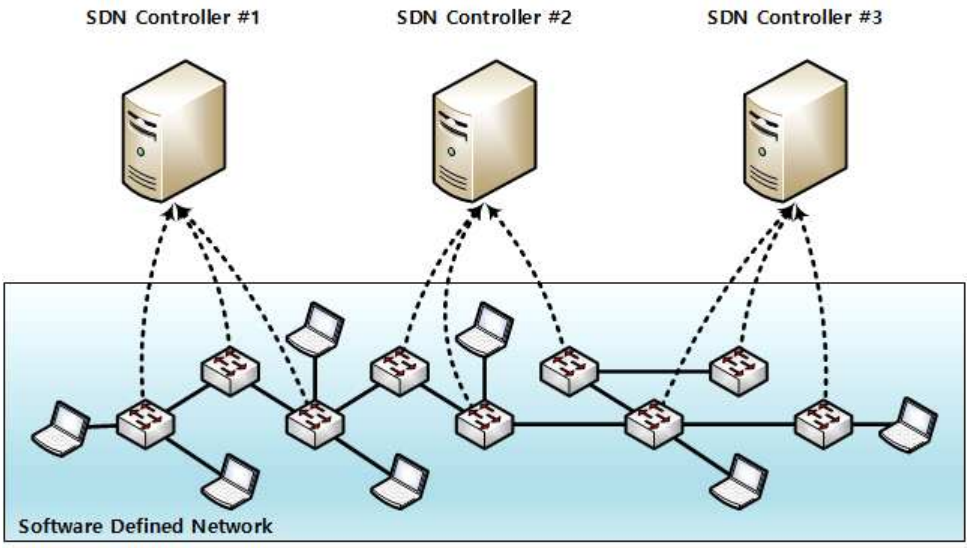


그림 3. 분산 SDN의 개념도

■ 그러나 분산형 SDN 컨트롤러를 사용한다 하더라도 여전히 하나의 문제가 남아있다. 수많은 네트워크 장비가 있는 네트워크 환경에서, 어떤 SDN 컨트롤러가 어떤 네트워크 장비를 제어/관리할 지를 결정(마스터쉽)하는, 이른바 컨트롤러 배치 문제(controller placement problem)[7]이다. 그림 4는 이와 같은 문제의 개념도이다. 그림 4에서 볼 수 있듯이, 만일 3개의 컨트롤러가 9개의 스위치를 제어/관리하는 환경에서 각 스위치가 과연 어떤 컨트롤러에게 제어/관리를 받아야 할지를 결정해야 한다.

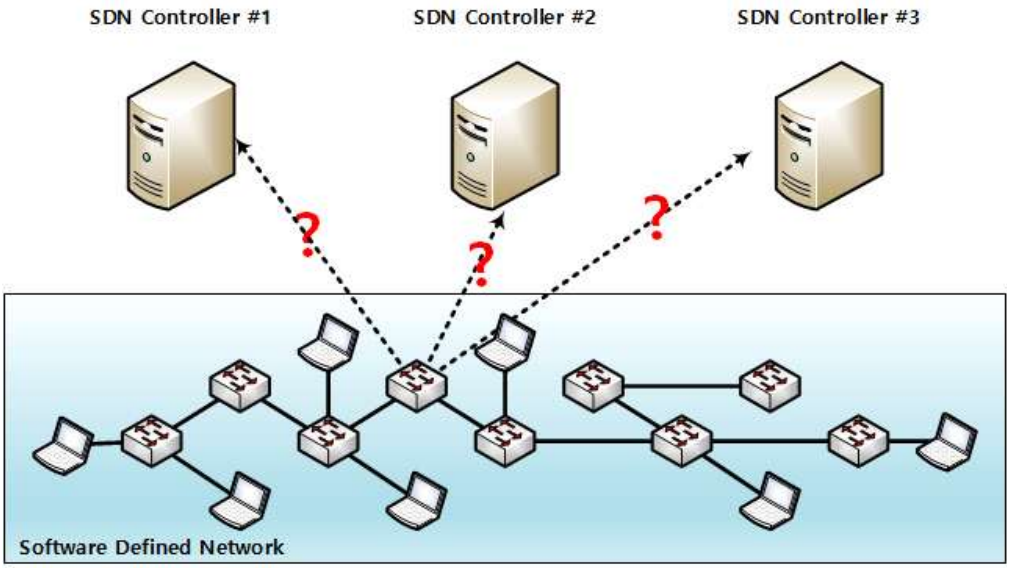


그림 4. 컨트롤러 배치 문제의 개념도

■ 본 기술문서에서는, 이러한 컨트롤러 배치 문제를 해결하기 위한 기존 기술들에 대해 알아본다. 그리고 이러한 문제를 실제로 해결하기 위한 소프트웨어의 설계 및 실행 결과를 서술한다.

2. 기존에 제안된 마스터쉽 제어/관리 기술

■ 최초에 이와 같은 마스터쉽 기술은 WAN(Wide-Area Network)환경에서 다수의 네트워크 장치가 다수의 컨트롤러에게 제어/관리를 받는 환경을 가정한다. 이와 같은 환경에서, 각 컨트롤러는 하나의 공간에 밀집되어있지 않고, 도처에 위치가 되어있다. 이 때, 각 네트워크 장비들은 RTT(Round-Trip Time)을 고려하여, RTT가 가장 짧은 컨트롤러에게 서비스를 받는 기술이 제안되었다[7].

■ 그리고 제어 평면 메시지의 수가 지속적으로 변화한다는 것에 착안하여, 긴 주기마다 한 번씩 마스터쉽을 수행하는 연구도 있다. 이 연구에서는 각 컨트롤러에서 요청하는 통계(statistics)를 위한 제어 메시지 수, 각 플로우들이 설정되는 비용(flow setup cost), 컨트롤러 사이의 동기화를 위한 비용(synchronization cost), 그리고 스위치 재 할당을 위한 비용(switch reassignment cost)을 고려하여 수행한다[8].

■ 이 외에도, 각 컨트롤러가 송/수신하는 제어 평면 메시지의 수를 분산하기 위한 기술도 제안되었다. 이 기술은 네트워크 장비가 처리하는 플로우의 수 및 기타 다른 특성에 따라, 서로 다른 제어 평면 메시지를 송/수신하는 것을 전제로 삼는다. 어떤 네트워크 장비는 다수의 새로운 플로우가 생성되어 많은 제어 평면 메시지를 송/수신하는 반면, 어떤 네트워크 장비는 그렇지 않는 경우가 발생한다. 만일 네트워크 장비의 수량대로 분산을 시키면, 특정 컨트롤러에 다수의 제어 평면 메시지가 집중되므로, 특정 컨트롤러에 병목현상을 일으킬 수 있다. 이러한 문제를 해결하기 위해, 각 컨트롤러가 송/수신하는 제어 평면 메시지의 수를 동일하도록 마스터쉽을 변경하는 연구가 수행되었다[9].

■ 한편, 현재 실제로 사용되고 있는 ONOS[4],[5]의 경우, 두 가지 방법으로 마스터쉽을 수행하고 있다. 첫 번째 방법은 FCFS(First-Come, First-Serve) 기반의 마스터쉽이다. 문자 그대로, 다수의 컨트롤러 중 가장 마스터쉽 요청/응답 메시지가 빨리 도착한 컨트롤러가 해당 요청/응답 메시지를 송신한 네트워크 장비를 제어/관리하는 기법이다. 그림 5는 이 기법에 대한 예시를 나타낸다. 그림 5에서는 하나의 네트워크 장비(미배정 노드)가 세 개의 컨트롤러(제 1 컨트롤러, 제 2 컨트롤러, 제 3 컨트롤러) 중 하나의 컨트롤러에게 제어/관리를 받아야 되는 환경이라고 가정한다. 그리고 컨트롤러와 네트워크 장비 사이의 제어 프로토콜은 오픈 플로우(OpenFlow)[1]를 사용한다고 가정한다. 이 때, 먼저 네트워크 장비는 각 컨트롤러들에게 자신을 알리며, 연결을 맺는 연결 과정을 갖는다. 이 후, 네트워크 장비를 제어/관리한 컨트롤러를 선출(master controller election)하게 된다. 이를 위해서, 오픈 플로우 표준에 있는 Role-request/response 메시지를 사용하며, ONOS 상에서는 다음과 같은 흐름으로 동작한다. 먼저 각 컨트롤러는

Role-request 메시지를 네트워크 장비에 전송을 하게 된다. 이를 받은 네트워크 장비는 Role-request 메시지에 대한 응답으로 Role-response 메시지를 각 컨트롤러에 전송하게 된다. 각 컨트롤러는 네트워크 장비로부터 Role-response 메시지를 받게 되는데, 네트워크 상태 혹은 다양한 이유로 인해 각 컨트롤러가 받는 Role-response 메시지의 시간이 다르게 된다. 그림 5에서는 제 3 컨트롤러가 가장 빨리 Role-response 메시지를 받은 컨트롤러이다. 이 경우, 네트워크 장비를 제어/관리하는 컨트롤러는 제 3 컨트롤러가 된다. 한편, 두 번째 방법은, 모든 컨트롤러가 동일한 수의 네트워크 장비를 제어/관리하도록 하는 방법(equalizing mastership)이 있다. 두 방법 사이의 차이는 언제 동작을 하는 시점에 있다. 첫 번재 방법의 경우, 처음 네트워크 장비와 컨트롤러 사이의 최초 연결 시점(initializing phase)에 동작되는 방법이다. 반면, 두 번째 방법은 동작이 되고 있는 어떠한 시점(runtime phase)에도 동작이 될 수 있는 방법이다.

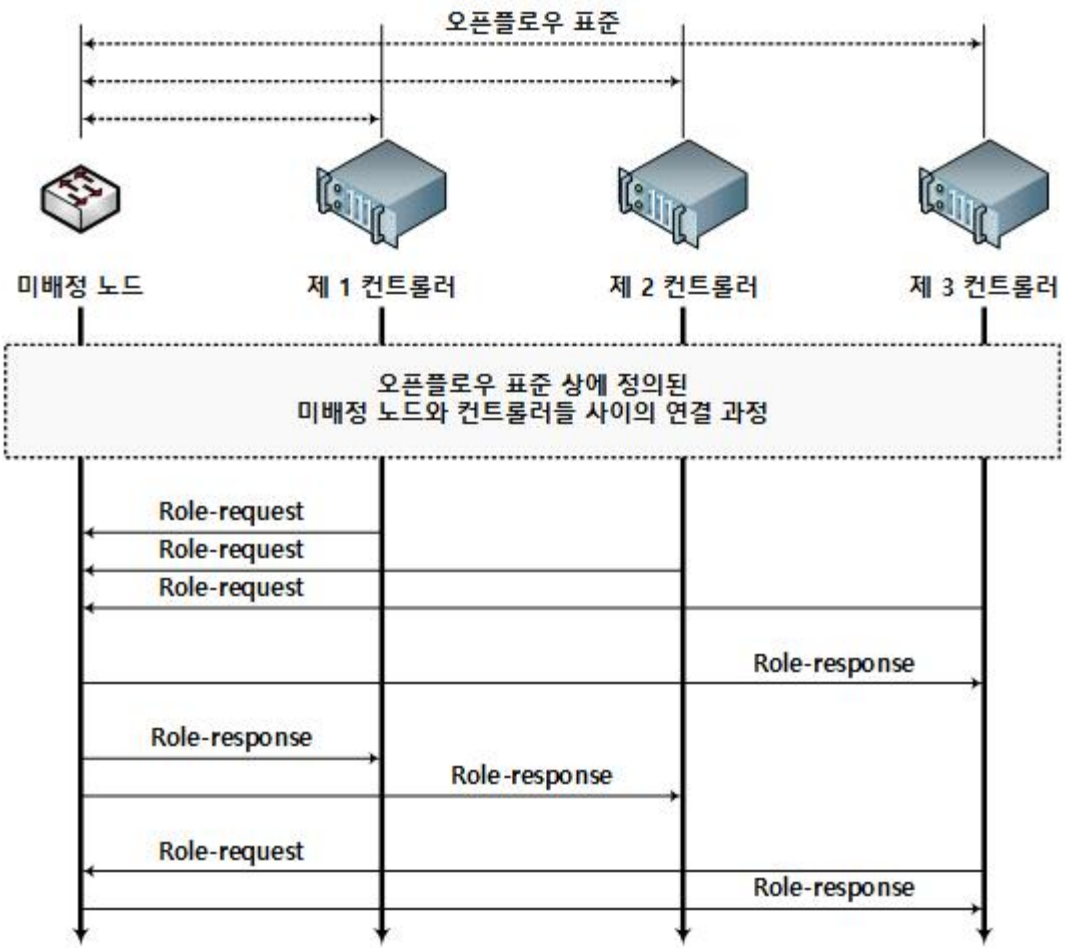


그림 5. ONOS 컨트롤러의 마스터쉽 예시도

3. 기존에 제안된 마스터쉽 제어/관리를 위한 소프트웨어

■ 앞서 살펴본 기술들은 각 환경에서 마스터쉽을 수행하기 위해 제안된 기술이다. 현재 ONOS와 같은 상용 컨트롤러 상에는 앞서 언급한 바와 같이 간단한 수준의 마스터쉽 기능이 구현되어 있다.

■ 본 기술문서에서는 간단한 수준의 마스터쉽이 아닌, 기존에 제안된 기술들을 기반으로 동작할 수 있는 소프트웨어를 설계 및 구현한다. 이 때, 기반이 되는 기술은 앞서 기존 기술에서 언급한 컨트롤러들의 제어 평면 메시지를 동일하게 하는 기법을 사용한다[9]. 그러나 이 기법의 경우, 일회성으로 한번만 마스터쉽을 수행하도록 제안되어 있으며, 언제 동작하는지에 대한 연구는 되어있지 않다. 만일 제어 평면 메시지가 지속적으로 변화하는 환경이라면, 제어 평면 메시지의 분산이 잘 이루어지지 않게 된다. 이는 특정 컨트롤러가 또 다시 병목현상을 겪을 수 있게 된다. 따라서 본 기술문서에서 제안하는 소프트웨어는 이러한 문제를 해결하기 위해, 1분마다 한 번씩 마스터쉽을 수행하도록 구현하였다.

3.1. 디자인

■ 본 기술문서에서 제안하는 소프트웨어는 웹 GUI를 제공하는 웹 GUI 기반 오케스트레이터(orchestrator)이다. 이는 그림 6과 같이 디자인을 하였다.

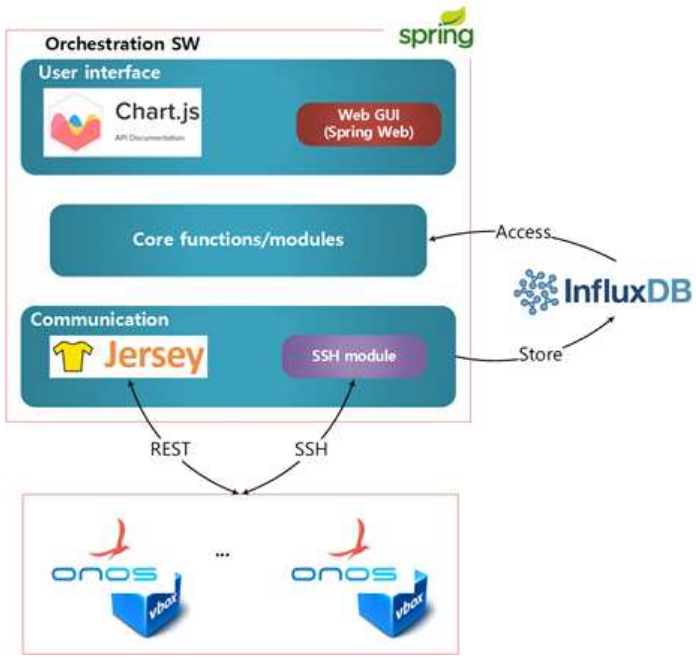


그림 6. 오케스트레이터의 구조

■ 먼저 오케스트레이터는 기본적으로 자바(Java) 언어를 통해 구현되었으며, 크게 세 개의 계층으로 구성이 되어 있다. 먼저 UI(User Interface) 계층은 사용자

들에게 웹 기반 GUI를 제공하기 위해 스프링 웹 프레임워크(Spring Web framework)를 포함한다. 그리고 웹 기반 GUI에서, 오케스트레이터는 사용자들에게 모니터링 결과를 시각화 하기 위해, 자바스크립트 라이브러리에 하나인 Chart.js를 포함한다. 한편, SDN 컨트롤러와 통신 및 모니터링 결과를 저장하는 데이터베이스와 통신하기 위해 커뮤니케이션(communication) 계층이 존재한다. 이 계층에는 SDN 컨트롤러와 REST(REpresentational State Transfer) API를 제공하기 위해 JERSEY/JACKSON 라이브러리를 사용하였고, 가상 머신 및 물리 머신, 그리고 데이터베이스와의 SSH(Secure SHell) 통신을 위해 SSH 통신 라이브러리를 포함한다. 마지막으로 오케스트레이터는 주기적으로 컨트롤러의 모니터링을 수행하고 마스터쉽을 주기적으로 동작시키기 위한 코어 기능/모듈 계층(Core functions/modules)이 존재한다. 한편, 오케스트레이터는 각 컨트롤러에서 수집한 모니터링 결과를 저장하기 위해 데이터베이스가 사용되었으며, 이 데이터베이스는 오픈소스 데이터베이스이며 시계열 데이터 저장에 적합한 인플럭스 데이터베이스(InfluxDB)를 사용하였다.

■ 오케스트레이터의 계층 중 코어 기능/모듈 계층에 대한 상세한 개념도는 그림 7과 같다.

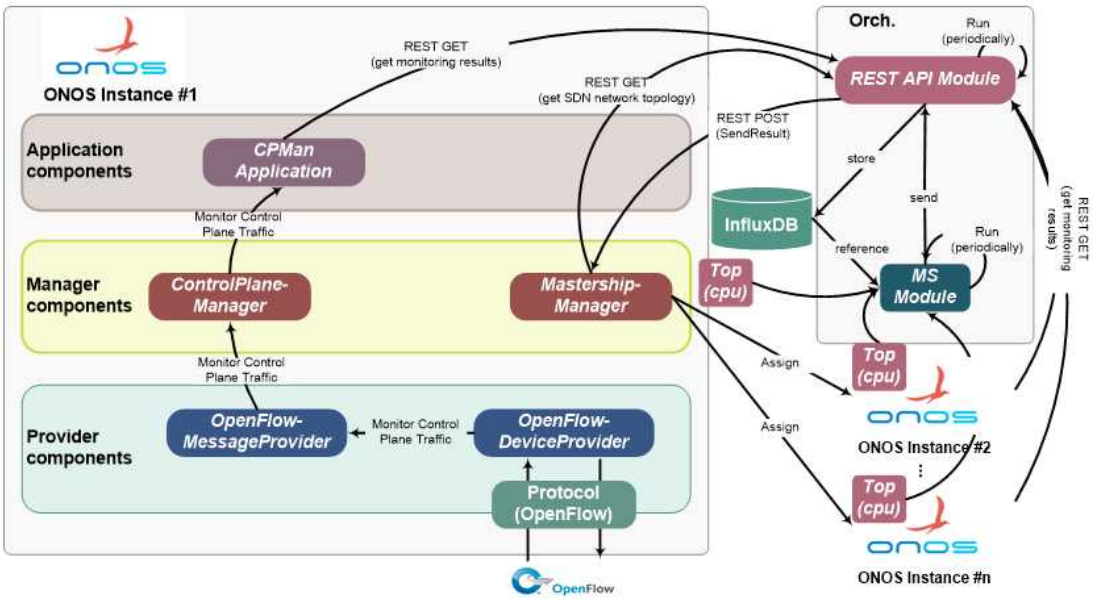


그림 7. 오케스트레이터의 코어 기능/모듈 계층에 대한 개념도

■ 먼저 컨트롤러는 제어 평면 메시지를 모니터링 하기 위한 어플리케이션(CPMan application)과 이를 위한 서브시스템 및 매니저(ControlPlaneManager, OpenFlowMessageProvider, OpenFlow Subsystem)가 필요하다. 오케스트레이터는 제어 평면 메시지를 매 1분마다 한 번씩 수집하는데, 이를 위해 REST API를 통해 컨트롤러에 요청한다. 요청을 받은 컨트롤러는 네트워크 장비와 컨트롤러 사

이에 얼마나 많은 제어 평면 메시지가 1분간 오갔는지를 알려준다. 이를 받은 오케스트레이터는 이 정보를 데이터베이스에 저장한다. 오케스트레이터 내의 마스터쉽 모듈(MS module)은 매 1분마다 모니터링이 종료되면 자동으로 동작이 된다. 마스터쉽 모듈은 모니터링 된 결과를 바탕으로 어떤 컨트롤러가 평균값 이상으로 제어 평면 메시지를 받았는지 계산한다. 계산 후, 마스터쉽 모듈은 평균값 이상으로 받은 컨트롤러에서 네트워크 장비를 평균값 이하로 받은 컨트롤러에게 마스터쉽을 변경한다. 이 때 사용되는 알고리즘은 휴리스틱(heuristic) 알고리즘의 하나인 욕심쟁이 알고리즘(greedy algorithm)을 사용하여 마스터쉽을 변경한다. 어떠한 네트워크 장비도 옮길 수 없는 상황이 되면, 마스터쉽 모듈의 기능을 종료한다. 이와 같은 방법은 기 제안된 연구 결과인 [9]의 알고리즘을 사용하였다. 이 후, 오케스트레이터는 REST API를 통해 컨트롤러에게 변경된 마스터쉽에 대한 정보를 알린다. 그러면, 컨트롤러들은 수신받은 마스터쉽 정보를 바탕으로 마스터쉽을 변경하게 된다.

■ 한편, 오케스트레이터에서 사용자가 더 많은 정보를 시각화할 수 있도록 하기 위해, 오케스트레이터는 각 컨트롤러의 CPU 정보를 수집하여 데이터베이스에 저장한다. CPU 정보를 수집하기 위해, 오케스트레이터는 각 컨트롤러에 SSH로 접속을 한다. 이후, 오케스트레이터는 각 컨트롤러에 top 툴을 동작시키게 된다. top 툴은 리눅스 운영체제에 포함된 컴퓨팅 자원 모니터링 툴의 하나이다. top툴을 통해 출력된 정보를 오케스트레이터는 파싱하여 데이터베이스에 저장한다.

■ 데이터베이스에 저장된 모든 정보들은 웹 GUI와 Chart.js를 통해 모두 웹 상에 그래프로 출력을 할 수 있다. 데이터베이스로부터 정보를 불러오기 위해, 오케스트레이터는 SSH 모듈을 사용하여 데이터베이스 머신에 접속한다. 이 후, 데이터베이스 쿼리(query)문을 사용하여 정보를 불러온다. 불러온 정보를 SSH를 통해 다시 오케스트레이터로 가져온 다음, 해당 정보를 파싱한다. 파싱된 정보는 스프링 웹 프레임워크와 Chart.js 라이브러리를 통해 웹상에 최종 출력된다.

3.2. 구현

■ 위의 디자인 결과를 구현하기 위해 다음과 같이 패키징을 수행하였다. 그림 8은 디자인 결과를 패키징 한 개념도이다. 먼저 크게 세 개의 계층으로 구분할 수 있다. 웹 어플리케이션 계층(webapp)은 사용자들에게 웹 GUI를 출력하기 위한 기본적인 요소들이 들어있다. 자원 계층(resource)은 스프링 웹 프레임워크를 설정하기 위한 설정 파일들이 들어 있다. 마지막으로 자바 계층(Java)은 서버단(server-side/backend)에서 수행하는 모든 함수 및 기능들이 구현된 계층이다.

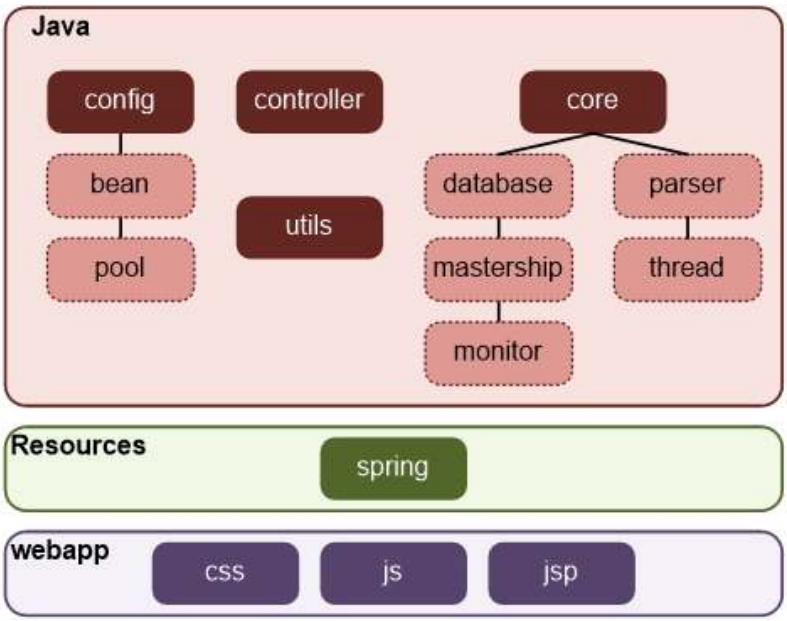


그림 8. 오케스트레이터의 패키징 정보를 도식화한 개념도

■ 각 계층을 좀 더 자세히 살펴보면, 웹 어플리케이션 계층에는 세 가지의 패키지가 존재한다. css 패키지는 웹 GUI의 디자인을 위한 css(cascading style sheets) 파일이 존재한다. js 패키지는 웹 GUI의 프론트엔드(frontend)단에서의 웹 기반 연산을 위한 자바스크립트(Javascript)라이브러리가 존재한다. 마지막으로 jsp는 실제로 사용자에게 웹 GUI를 보여주는 jsp(Java Server Pages)파일들이 존재한다.

■ 자원 계층은 spring 패키지 하나가 존재한다. 이 패키지 안에는 스프링 웹 프레임워크가 동작을 위한 설정 파일들이 위치하고 있으며, 해당 파일들은 xml(extensible markup language) 형태로 저장되어 있다.

■ 자바 계층은 크게 네 개의 패키지가 존재한다. 설정 패키지(config)는 오케스트레이터의 다양한 설정 파일들 및 전역 변수들을 자바 파일의 형태로써 담고 있으며, 각종 클래스들을 담고 있다. 특히 설정 패키지는 크게 두 개의 서브 패키지(sub-package)를 가지고 있다. 빈 서브 패키지(bean)는 네트워크 장비, 컨트롤러, 가상 머신, 데이터베이스등을 위한 클래스를 빈 파일의 형태로써 가진 서브 패키지이다. 한편, 풀 패키지(pool)는 빈 파일들을 저장하는 풀을 의미한다. 컨트롤러 패키지(controller)는 스프링 웹 프레임워크의 MVC(Model-View-Controller) 모델에서 컨트롤러를 의미한다. MVC 모델에서의 컨트롤러란, 웹 GUI에서 프론트엔드로 사용자에게 보이기 전에, 서버단(server-side/backend)에서 연산하는 백엔드 UI를 의미한다. 즉, 웹 어플리케이션 계층에서 jsp 패키지 내에 있는 jsp 파일들의 서버단에서의 연산을 위한 자바 파일들이 위치한다. 유틸리티 패키지(utils)는 트롤러 및 데이터베이스와의 통신을 위한 자바 파일등이 위치하는 유틸리티

패키지이다. 마지막으로 코어 패키지(core)는 오케스트레이터의 마스터쉽 등을 위한 핵심 기능이 포함된 자바 파일들이 위치한 패키지이다. 코어 패키지에는 총 다섯 개의 서브 패키지가 존재한다. 먼저 데이터베이스 서브 패키지(database)는 데이터베이스에 연관된 서브 패키지로, 데이터베이스에 정보를 넣거나, 모니터링 결과 출력을 위해 정보를 데이터베이스로부터 가져오는 자바 파일들이 위치한다. 마스터쉽 서브 패키지(mastership)의 경우, 컨트롤러와 네트워크 장비의 마스터쉽을 수행하는 기능이 있는 자바 파일이 위치하는 서브 패키지이다. 모니터링 서브 패키지(monitor)의 경우, 사용자들에게 웹 GUI로 출력하기 전에 모니터링 결과를 수집하여 가공하는 자바 파일이 위치하는 서브 패키지이다. 파서는 REST API나 SSH로부터 받은 파일들을 파싱하는 자바 파일이 위치한 서브 패키지이다. 마지막으로 스레드 서브 패키지(thread)는, 동작하는 다양한 자바 기능들, 특히 모니터링/마스터쉽 기능을, 스레드의 형태로 만들어 주는 자바 파일이 위치한 서브 패키지이다. 이 스레드는 오케스트레이터에서 백그라운드로 1분마다 동작한다.

■ 자원 계층에 있는 스프링 설정 파일은 다음의 그림 9-10과 같이 구현하였다. 먼저 스프링 설정 파일은 크게 두 개의 파일로 구성한다. Dispatcher-servlet.xml 파일은 그림 9와 같이 설정하며, Spring-context.xml의 파일은 다음과 같이 구현한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

</beans>
```

그림 9. Dispatcher-servlet.xml의 내용

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spri

    <mvc:annotation-driven/>
    <context:component-scan base-package="kr.postech.monet" />

    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

그림 10. Spring-context.xml의 내용

■ 한편, 웹 어플리케이션 계층은 다음과 같은 파일로 구성할 수 있다. 먼저 css 패키지 내에는 jsp 패키지 내에서 사용할 다양한 스타일 파일이 존재한다. js 패키지에는 Chart.js를 위한 chart.js 라이브러리 파일과, 웹 GUI를 비동기성으로 동

작시키기 위한 Jquery 라이브러리가 위치한다. 한편, jsp 패키지에는 크게 5개의 웹 GUI 파일이 존재한다. index.jsp 파일은 오케스트레이터를 처음 켰을 때 사용자에게 출력하는 jsp 파일이다. dashboard.jsp 파일은 오케스트레이터를 처음 켰을 때, index.jsp 파일 내의 본문을 출력하는 jsp 파일이다. monitoring.jsp 파일은 모니터링 결과를 오케스트레이터에 출력하는 jsp 파일이다. configuration.jsp 파일은 오케스트레이터의 설정값을 사용자에게 출력하는 jsp 파일이다. 마지막으로 mastership.jsp 파일은 마스터쉽과 관련된 정보를 사용자에게 시각화 하는 jsp 파일이다. 해당 파일에 대한 자세한 소스 코드는 github를 통해 확인할 수 있을 예정이다.

■ 마지막으로 자바 계층 중 설정 패키지는 다음과 같은 파일이 존재한다. 먼저 GeneralConf.java 파일은 오케스트레이터 내의 다양한 설정값을 전역변수로서 가지고 있다. 그림 11은 설정값을 저장한 해당 파일의 일부이다. 그림에서 볼 수 있듯이, 전역변수들에 대해 해당 정보를 저장하고 있다. 한편, 이 설정 파일에는 주기적으로 마스터쉽을 수행할 수 있도록, 처음 오케스트레이터를 켤 때 1분마다 모니터링 및 마스터쉽 스레드를 동작시키는 함수를 실행한다. 그림 12는 스프링 웹 프레임워크 내의 이와 같은 역할을 하는 포스트 생성자(post constructor)를 나타낸다.

```
@Component
public class GeneralConf {

    // System parameters
    public static final String REST_URL_POSTFIX = "/onos/v1";

    // Pool list for PMConfPool
    public static SiteConfPool siteConfPoolList;
    public static VMBean dbBean;
```

그림 11. GeneralConf.java 파일의 내용 일부

```
// Run at the start of web server
@PostConstruct
public void init() {

    // heartbeat check
    SSHConnectionUtil sshConn = new SSHConnectionUtil();
    sshConn.checkHeartBeatAllMachines();
    HeartbeatCheckThread.funcCheckHeartBeatAllMachineaThread();

    // initialize Database
    initDatabase();

    // periodically run mastership and autoscaling as a thread
    MastershipAndScalingThread.funcMastershipAndScalingThread();
}
```

그림 12. GeneralConf.java 내의 포스트 생성자

■ 또한, 자바 계층의 설정 패키지 중 빈 서브 패키지는 각 네트워크 장비, 컨트롤러, 가상 머신, 물리 머신등에 대한 정보를 빈의 형태로써 클래스화 한 파일이 있다. 먼저 PMBean.java파일의 경우, 물리 머신에 대한 정보를 클래스화 한 파일로써 그림 13과 같이 구성되어 있다. 해당 빈 파일에는 물리 머신을 오케스트레이터에서 사용할 별칭, 물리머신 내에서 구동되는 가상 머신 풀, IP 주소, SSH 포트, SSH 아이디, 비밀번호 등이 저장된다.

```
public class PMBean {

    private String pmAlias;
    private VMConfPool vmConfPool;

    private String ipAddress;
    private String sshPort;

    private String accessIPAddress;
    private String accessSSHPort;

    private String ID;
    private String PW;

    private int numCPU;

    private boolean alive;

    public PMBean(String pmAlias, List<VMBean> vmBeans) {
        this.pmAlias = pmAlias;
        this.vmConfPool = new VMConfPool();
        this.vmConfPool.setVmBeans(vmBeans);
        alive = false;
    }
```

그림 13. PMBean.java 내용

■ 빈 서브패키지의 VMBean.java파일의 경우, 가상 머신에 대한 정보를 클래스와 한 파일로, 그림 14와 같이 구성되어 있다. 먼저 가상 머신에 대한 별칭, IP 주소, SSH 포트번호, SDN 컨트롤러의 관리자 페이지에 대한 웹 포트 번호, SDN 컨트롤러의 ID, 비밀번호, SSH 아이디, 비밀번호, 현재 꺼졌는지 켜졌는지의 상태, 그리고 현재 SDN 컨트롤러가 구동 중인지 여부에 대한 상태를 갖고 있다.

그림 14. VMBean.java 내용

■ 빈 서브패키지의 SWBean.java파일의 경우, 네트워크 내에 있는 네트워크 장비를 클래스로 만든 파일로 그림 15와 같이 구성되어 있다. 각 네트워크 장비의 아이디인 DPID, 그리고 각종 제어 평면 메시지들의 타입을 모니터링 하기 위한 변수들이 저장되어 있다.

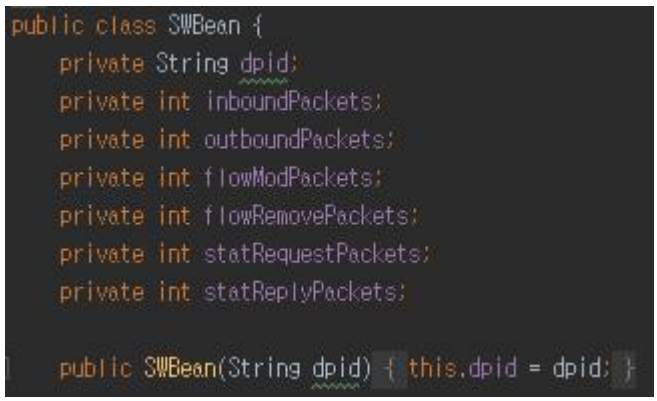


그림 15. SWBean.java 내용

■ 마지막으로 빈 서브 패키지의 DBBean.java파일의 경우, 오케스트레이터의 정보 및 모니터링 정보를 저장하기 위한 자바 빈 클래스로, 그림 16과 같이 구성되어 있다. 여기에는, 데이터베이스의 별칭, 데이터베이스의 IP 주소, 데이터베이스의 관리자 페이지 접속을 위한 포트 번호, 데이터베이스로 쿼리를 전송할 때 필요한 포트 번호가 포함되어 있다.

```java
public class DBBean {

    private String dbAlias;
    private String dbIPAddress;
    private String dbAdminPort;
    private String dbQueryPort;

    private String accessDBIPAddress;
    private String accessDBAdminPort;
    private String accessDBQueryPort;
```

그림 16. DBBean.java 내용

■ 한편 자바 계층의 설정 패키지 내 풀 서브 패키지는 물리 머신의 빈 파일 오브젝트를 저장할 수 있는 PMConfPool.java 파일과(그림 17), 가상 머신의 빈 파일 오브젝트를 저장할 수 있는 VMConfPool.java 파일(그림 18)이 있다. 각 풀 파일은 내부에 자바 언어에서 기본적으로 제공하는 리스트를 사용하여 풀을 만들었다.

```java
public class PMConfPool {
    private List<PMBean> pmBeans;

    public PMConfPool() { pmBeans = new CopyOnWriteArrayList<PMBean>(); }

    public List<PMBean> getPmBeans() { return pmBeans; }

    public void setPmBeans(List<PMBean> pmBeans) { this.pmBeans = pmBeans; }
}
```

그림 17. PMConfPool.java 내용

```java
public class VMConfPool {

    private List<VMBean> vmBeans;

    public VMConfPool() { vmBeans = new CopyOnWriteArrayList<VMBean>(); }

    public List<VMBean> getVmBeans() { return vmBeans; }

    public void setVmBeans(List<VMBean> vmBeans) { this.vmBeans = vmBeans; }
}
```

그림 18. VMConfPool.java 내용

■ 자바 계층의 유틸리티 패키지에는 컨트롤러와 REST API로 접속하여 정보를 가져오거나 정보를 전달하기 위한 RESTConnectionUtil.java 파일이 존재한다. 먼

저 해당 파일에서, 컨트롤러로부터 정보를 요청하기 위한 함수는 그림 19와 같이 구현되었다. 이 함수는 JERSEY/JACKSON 라이브러리를 통해, 각 컨트롤러에 접속을 하도록 한다. 이 후, 반환값은 REST API를 불러온 값을 파싱 하지 않고 직접 반환한다. 한편 정보를 단순히 오케스트레이터가 컨트롤러로 전송하기 위해서는 그림 20과 같이 구현하였다. 단순히 REST API로 정보를 전송하는 것은 반환값이 없으므로, void 타입으로 구현하였다.

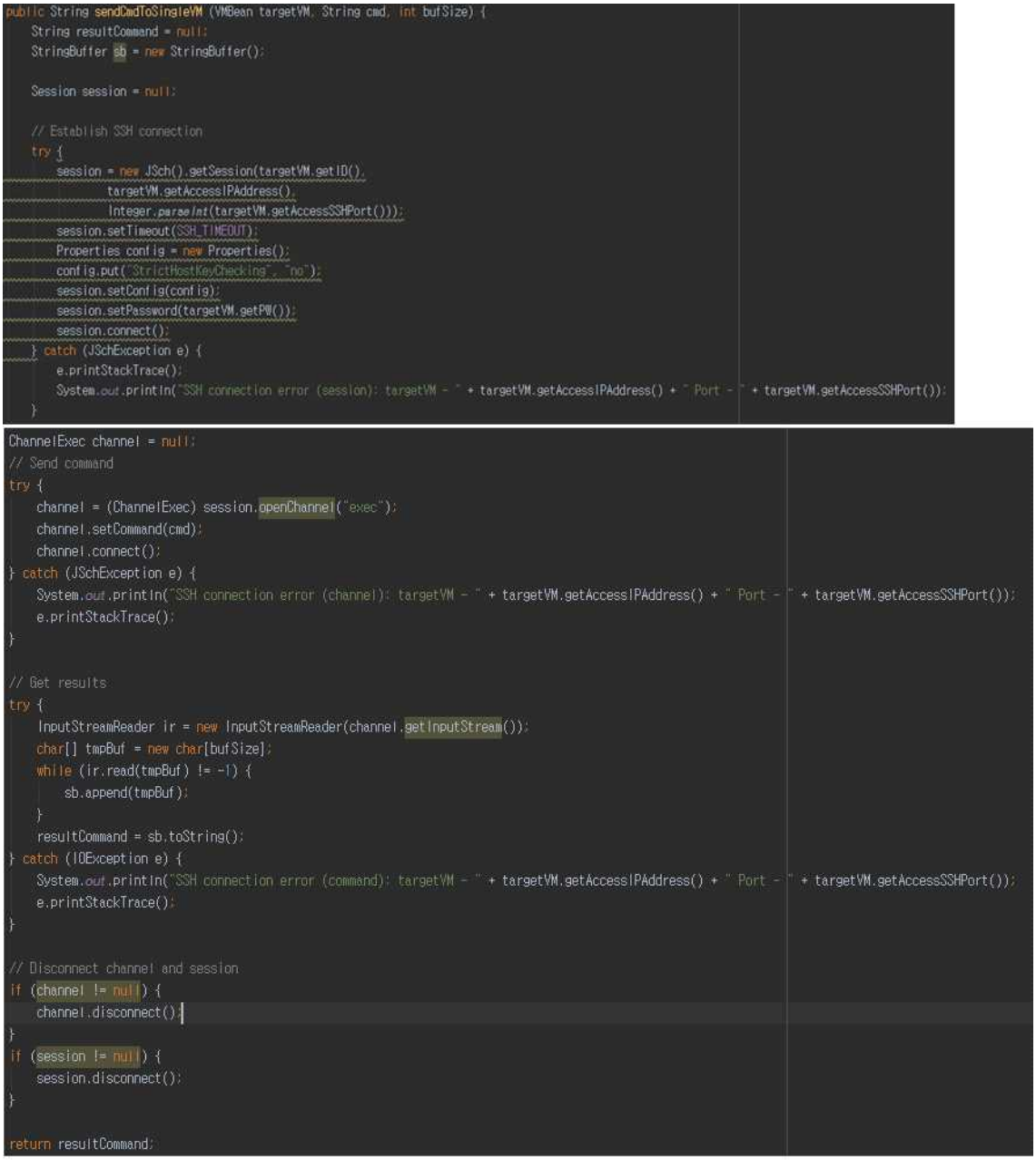
```
public String getRESTToSingleVM (VMBean targetVM, String url) {
    String resultJsonString = null;

    // Connection through REST API
    Client client = ClientBuilder.newClient();
    client.register(HttpAuthenticationFeature.basic(targetVM.getONOSID(), targetVM.getONOSPW()));
    WebTarget target = client.target(url);

    Invocation.Builder builder = target.request(MediaType.APPLICATION_JSON);
    Response response = builder.get();

    // Does REST server working?
    if (response.getStatus() != 200) {
        System.out.println("REST Connection Error - VM: " + targetVM.getVmAlias() + " (Status: " + response.getStatus() + ")");
        return null;
    }

    resultJsonString = builder.get(String.class);

    return resultJsonString;
}
```

그림 19. RESTConnectionUtil.java 중 REST API로 정보를 요청하는 코드

```
public void putRESTToSingleVM (VMBean targetVM, String url, JsonObject parameterJson) {
    // Connection through REST API
    Client client = ClientBuilder.newClient();
    client.register(HttpAuthenticationFeature.basic(targetVM.getONOSID(), targetVM.getONOSPW()));
    WebTarget target = client.target(url);

    Invocation.Builder builder = target.request(MediaType.APPLICATION_JSON);
    Response response = builder.put(Entity.entity(parameterJson.toString(), MediaType.APPLICATION_JSON));

    // Does REST server working?
    if (response.getStatus() != 200) {
        System.out.println("REST Connection Error - VM: " + targetVM.getVmAlias() + " (Status: " + response.getStatus() + ")");
        return;
    }
}
```

그림 20. RESTConnectionUtil.java 중 REST API로 정보를 전송하는 코드

■ 자바 계층 유틸리티 패키지에서 SSH 연결을 수행하는 함수는 SSHConnectionUtil.java 파일에 그림 21과 같이 구현되었다. 해당 함수는 JSch라는 라이브러리를 사용하여 세션을 연결한다. 이 후, SSH 명령어를 세션으로 보낸 후, 버퍼에 실행 결과를 불러오는 형식으로 구현이 되었다. 이 함수의 반환값은 버퍼에 실행 결과를 그대로 반환하도록 구현하였다.



```
public String sendCmdToSingleVM (VMBean targetVM, String cmd, int bufSize) {
    String resultCommand = null;
    StringBuffer sb = new StringBuffer();

    Session session = null;

    // Establish SSH connection
    try {
        session = new JSch().getSession(targetVM.getID(),
                targetVM.getAccessIPAddress(),
                Integer.parseInt(targetVM.getAccessSSHPort()));
        session.setTimeout(SSH_TIMEOUT);
        Properties config = new Properties();
        config.put("StrictHostKeyChecking", "no");
        session.setConfig(config);
        session.setPassword(targetVM.getPW());
        session.connect();
    } catch (JSchException e) {
        e.printStackTrace();
        System.out.println("SSH connection error (session): targetVM - " + targetVM.getAccessIPAddress() + " Port - " + targetVM.getAccessSSHPort());
    }
```

```
ChannelExec channel = null;
// Send command
try {
    channel = (ChannelExec) session.openChannel("exec");
    channel.setCommand(cmd);
    channel.connect();
} catch (JSchException e) {
    System.out.println("SSH connection error (channel): targetVM - " + targetVM.getAccessIPAddress() + " Port - " + targetVM.getAccessSSHPort());
    e.printStackTrace();
}

// Get results
try {
    InputStreamReader ir = new InputStreamReader(channel.getInputStream());
    char[] tmpBuf = new char[bufSize];
    while (ir.read(tmpBuf) != -1) {
        sb.append(tmpBuf);
    }
    resultCommand = sb.toString();
} catch (IOException e) {
    System.out.println("SSH connection error (command): targetVM - " + targetVM.getAccessIPAddress() + " Port - " + targetVM.getAccessSSHPort());
    e.printStackTrace();
}

// Disconnect channel and session
if (channel != null) {
    channel.disconnect();
}
if (session != null) {
    session.disconnect();
}

return resultCommand;
```

그림 21. SSHConnectionUtil.java 내용

■ 자바 계층 코어 패키지 내 데이터베이스 서브패키지에는 데이터베이스를 클래스화 한 빈 파일이 있다. 데이터베이스 내에는 제어 평면 트래픽의 양, CPU 부하의 양, 그리고 각 컨트롤러가 몇 개의 네트워크 장비를 가지고 있는지를 나타내는 총 3개의 테이블이 있다. 데이터베이스 서브패키지 내의 빈 파일들은 각 테이블들의 정보를 클래스화 한 파일이다. 그림 22는 제어 평면 메시지를 모니터링하는 테이블을 클래스화 하였다(ControlTrafficBean.java). 해당 테이블에는 측정한 시간 값, 각종 제어 평면 메시지 타입의 송/수신 횟수, 그리고 전체 송/수신 제어 평면 메시지의 수를 담고 있다. 그림 23과 같이 CPULoadBean.java 파일은

CPU 모니터링 정보를 저장하는 테이블을 클래스화 한 파일이다. 해당 파일에는 기록한 시간과 CPU 부하를 측정한 값이 저장된다. 마지막으로 그림 24는 NumSwitchesBean.java 파일의 내용으로, 각 컨트롤러가 몇 개의 네트워크 장비를 서비스하고 있는지 기록하는 파일이다. 해당 파일 내에는 시간값과 네트워크 장비의 수를 표시하고 있도록 클래스를 작성하였다.

```
public class ControlTrafficBean {
    private String time;
    private int numInbound;
    private int numOutbound;
    private int numFlowMod;
    private int numFlowRem;
    private int numStatReq;
    private int numStatRep;
    private int numTotalPackets;

    public ControlTrafficBean(String time, int numInbound, int numOutbound, int numFlowMod, int numFlowRem, int numStatReq, int numStatRep, int numTotalPackets) {
        this.time = time;
        this.numInbound = numInbound;
        this.numOutbound = numOutbound;
        this.numFlowMod = numFlowMod;
        this.numFlowRem = numFlowRem;
        this.numStatReq = numStatReq;
        this.numStatRep = numStatRep;
        this.numTotalPackets = numTotalPackets;
    }
```

그림 22. ControlTrafficBean.java 내용

```
public class CPULoadBean {
    private String time;
    private float cpuLoad;

    public CPULoadBean(String time, float cpuLoad) {
        this.time = time;
        this.cpuLoad = cpuLoad;
    }
```

그림 23. CPULoadBean.java 내용

```
public class NumSwitchesBean {
    private String time;
    private int numSwitches;

    public NumSwitchesBean(String time, int numSwitches) {
        this.time = time;
        this.numSwitches = numSwitches;
    }
```

그림 24. NumSwitchesBean.java 내용

■ 그리고 데이터베이스 서브클래스에는 각 테이블에서 정보를 불러오는 함수들이 있는 파일과, 테이블에 정보를 저장하는 함수가 있는 파일이 추가로 있다. 먼저 그림 25는 GetControlTraffic.java 파일로, 데이터베이스에서 특정 가상 머신의 제어 평면 메시지의 수를 불러오는 함수이다. 그림 26은 특정 가상 머신의 CPU

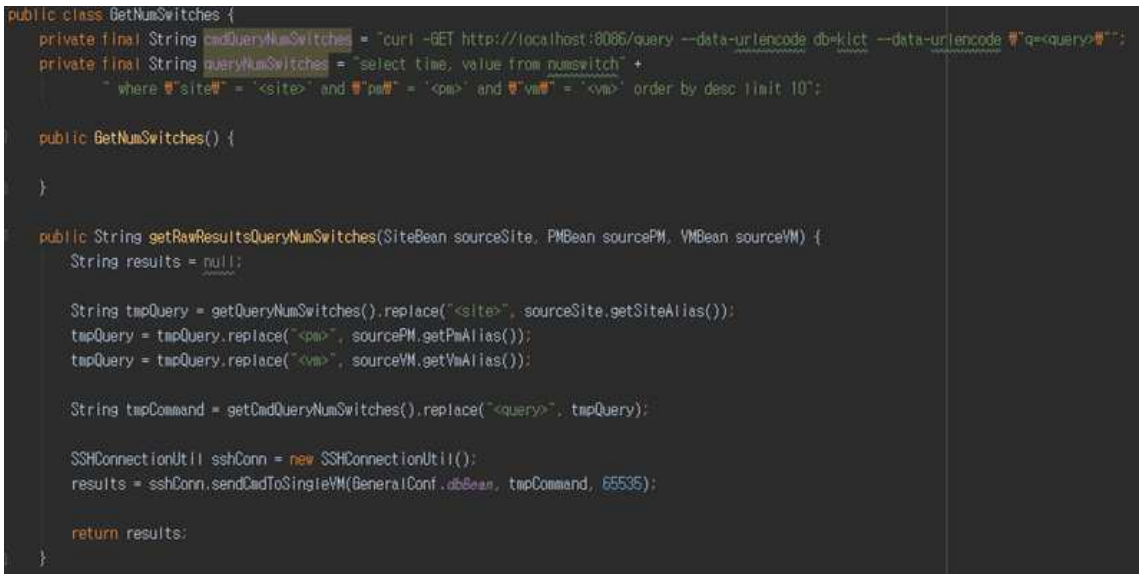
부하를(GetCPULoad.java), 그리고 그림 27은 특정 가상 머신 상의 컨트롤러가 서비스하는 네트워크 장비 수(GetNumSwitches.java)를 데이터베이스에서 불러오는 함수이다. 각 데이터베이스에 접속해서 정보를 불러오기 위해서는 오케스트레이터가 SSH 연결하는 함수를 사용하여 쿼리를 데이터베이스가 구동중인 머신에 전송한다. 각 파일들 내에 위치한 변수 중 접두사(prefix)가 cmd로 시작되는 문자열이 실제 데이터베이스에 SSH로 접속하여 실행하는 SSH 실행문이다. 그리고 각 파일들 내에 위치한 변수 중 접두사가 query로 시작되는 문자열이 실제 데이터베이스에 전송하는 쿼리이다. cmd로 시작하는 문자열 내에 query로 시작되는 문자열을 첨가하여 SSH로 구동하게 되는 원리로 작성하였다. 한편, 데이터베이스에 정보를 넣는 과정은, 그림 25-27과 동일하나, 데이터베이스 쿼리문만 서로 다르다. 제어 평면 메시지의 모니터링 정보를 데이터베이스에 넣는 파일은 PutControlTraffic.java, CPU 모니터링 정보를 데이터베이스에 넣는 파일은 PutCPULoad.java, 그리고 각 컨트롤러가 서비스하는 네트워크 장비의 수를 넣는 파일은 PutNumSwitches.java이다. 각 파일은 그림 25-27과 쿼리를 제외하고 동일하게 구성되어 있다. 그림 28, 29, 30은 PutControlTraffic.java, PutCPULoad.java, 그리고 PutNumSwitches.java의 쿼리문을 나타낸 그림이다.

```
public class GetControlTraffic {
    private final String cmdQueryControlTraffic = "curl -GET http://localhost:8086/query --data-urlencode db=kict --data-urlencode \"q=<query>\"";
    private final String queryControlTraffic = "select time, inbound, outbound, flowmod, flowrem, statreq, statrep, value from controltraffic" +
            " where \"site\" = '<site>' and \"pm\" = '<pm>' and \"vm\" = '<vm>' order by desc limit 10";

    public GetControlTraffic() {

    }

    public String getRawResultsQueryControlTraffic(SiteBean sourceSite, PMBean sourcePM, VMBean sourceVM) {
        String results = null;

        String tmpQuery = getQueryControlTraffic().replace("<site>", sourceSite.getSiteAlias());
        tmpQuery = tmpQuery.replace("<pm>", sourcePM.getPmAlias());
        tmpQuery = tmpQuery.replace("<vm>", sourceVM.getVmAlias());

        String tmpCommand = getCmdQueryControlTraffic().replace("<query>", tmpQuery);

        SSHConnectionUtil sshConn = new SSHConnectionUtil();
        results = sshConn.sendCmdToSingleVM(GeneralConf.dbBean, tmpCommand, 65535);

        return results;
    }
```

그림 25. GetControlTraffic.java 내용

```
public class GetCPULoad {
    private final String cmdQueryCPULoad = "curl -GET http://localhost:8086/query --data-urlencode db=kict --data-urlencode \"q=<query>\"";
    private final String queryCPULoad = "select time, value from cpuload" +
            " where \"site\" = '<site>' and \"pm\" = '<pm>' and \"vm\" = '<vm>' order by desc limit 10";

    public GetCPULoad() {
    }

    public String getRawResultsQueryCPULoad(SiteBean sourceSite, PMBean sourcePM, VMBean sourceVM) {
        String results = null;

        String tmpQuery = getQueryCPULoad().replace("<site>", sourceSite.getSiteAlias());
        tmpQuery = tmpQuery.replace("<pm>", sourcePM.getPmAlias());
        tmpQuery = tmpQuery.replace("<vm>", sourceVM.getVmAlias());

        String tmpCommand = getCmdQueryCPULoad().replace("<query>", tmpQuery);

        SSHConnectionUtil sshConn = new SSHConnectionUtil();
        results = sshConn.sendCmdToSingleVM(GeneralConf.dbBean, tmpCommand, 65535);

        return results;
    }
```

그림 26. GetCPULoad.java 내용

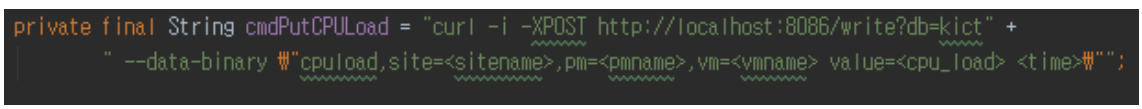


```
public class GetNumSwitches {
    private final String cmdQueryNumSwitches = "curl -GET http://localhost:8086/query --data-urlencode db=kict --data-urlencode \"q=<query>\"";
    private final String queryNumSwitches = "select time, value from numswitch" +
            " where \"site\" = '<site>' and \"pm\" = '<pm>' and \"vm\" = '<vm>' order by desc limit 10";

    public GetNumSwitches() {

    }

    public String getRawResultsQueryNumSwitches(SiteBean sourceSite, PMBean sourcePM, VMBean sourceVM) {
        String results = null;

        String tmpQuery = getQueryNumSwitches().replace("<site>", sourceSite.getSiteAlias());
        tmpQuery = tmpQuery.replace("<pm>", sourcePM.getPmAlias());
        tmpQuery = tmpQuery.replace("<vm>", sourceVM.getVmAlias());

        String tmpCommand = getCmdQueryNumSwitches().replace("<query>", tmpQuery);

        SSHConnectionUtil sshConn = new SSHConnectionUtil();
        results = sshConn.sendCmdToSingleVM(GeneralConf.dbBean, tmpCommand, 65535);

        return results;
    }
```

그림 27. GetNumSwitches.java 내용

```
private final String cmdPutControltraffic = "curl -i -XPOST http://localhost:8086/write?db=kict" +
        " --data-binary \"controltraffic,site=<sitename>,pm=<pmname>,vm=<vmname>,inbound=<num_inbound>," +
        "outbound=<num_outbound>,flowmod=<num_flowmod>,flowrem=<num_flowrem>,statreq=<num_statreq>," +
        "statrep=<num_statrep> value=<total_packets> <time>\"";
```

그림 28. PutControlTraffic.java의 쿼리 내용



```
private final String cmdPutCPULoad = "curl -i -XPOST http://localhost:8086/write?db=kict" +
        " --data-binary \"cpuload,site=<sitename>,pm=<pmname>,vm=<vmname> value=<cpu_load> <time>\"";
```

그림 29. PutCPULoad.java의 쿼리 내용

```
private final String cmdPutNumSwitches = "curl -i -XPOST http://localhost:8086/write?db=kict" +
        " --data-binary @"numswitch,site=<sitename>,pm=<pmname>,vm=<vmname> value=<num_switches> <time>@"";
```

그림 30. PutNumSwitches.java의 쿼리 내용

■ 자바 계층 코어 패키지 내 마스터쉽 서브 패키지에는 마스터쉽을 변경하도록 REST API를 통해 명령하는 함수인 changeMastership 함수가 위치한다. 해당 함수는 ChangeMasteship.java 파일에 위치해 있다. 해당 함수는 그림 31과 같이 구형되어 있다. 해당 함수 안에는 REST API 접속을 위한 컨트롤러의 HTTP 주소와 REST API를 불러오는 주소가 명시되어 있다. 그리고 컨트롤러가 마스터쉽을 변경하라는 명령은 JSON을 통해 전달하며, 이를 REST API를 통해 컨트롤러로 전송한다. JSON안에는 네트워크 장비의 아이디(deviceId), 컨트롤러의 아이디(nodeId), 그리고 마스터라는 역할을 명시해 전송한다.

```
public void changeMastership (SWBean movingSW, VMBean targetVM) {
    String mastershipRESTURL = "http://" + targetVM.getAccessIPAddress() + ":" + targetVM.getAccessHTTPPort() + RESTURL_CHANGEMASTERSHIP;

    JsonObject rootObj = new JsonObject();
    rootObj.add("deviceId", movingSW.getDpid());
    rootObj.add("nodeId", targetVM.getIpAddress());
    rootObj.add("role", "MASTER");

    RESTConnectionUtil restConn = new RESTConnectionUtil();
    restConn.putRESTToSingleVM(targetVM, mastershipRESTURL, rootObj);

}
```

그림 31. ChangeMastership.java의 chageMastership 함수 내용

■ 자바 계층 코어 패키지의 모니터링 서브 패키지에는 제어 평면 메시지의 모니터링 결과를 컨트롤러로부터 불러오는 파일인 GettingCPTraffic.java파일(그림 32)과 CPU 정보를 불러오는 GettingCPULoad.java 정보를 불러오는 파일(그림 33), 그리고 컨트롤러가 어떤 네트워크 장비를 서비스하는지에 대한 정보를 불러오는 GettingTopology.java파일(그림 35)이 위치한다. 먼저 제어 평면 메시지 모니터링 결과를 불러오는 파일에서는 REST API를 통해, 각 컨트롤러에 접속하여, 자신이 서비스하는 네트워크 장비와 자신 사이의 제어 평면 메시지 송/수신 모니터링 결과를 요청하여 받아온다. 반환값은 REST API의 JSON 내용을 파싱하여 각 네트워크 장비 클래스의 오브젝트에 저장하고 이 리스트를 반환한다. CPU 부하 정보를 불러오는 것은 각 머신에 SSH로 접속하여 top툴을 파싱하여 가져온다. 그림 34는 top툴에서 머신의 CPU 부하를 불러올 때 파싱한 정보를 나타낸다. 빨간 줄로 표시된 부분은 지난 1분간의 CPU 부하를 표기한 정보인데, GettingCPULoad.java 파일은 해당 정보를 불러온다. 마지막으로 컨트롤러와 네트워크 장비 사이의 토폴로지는 각 컨트롤러의 REST API를 통해 불러오도록 구현하였다.

```java
public class GettingCPTraffic {

    private final String CPTrafficRESTURL = "http://"
            + "<controllerIP>"
            + ":"
            + "<controllerPort>"
            + "/onos/cpman/controlmetrics/messages";

    public GettingCPTraffic() {
    }

    public String getCPTrafficRESTURL() { return CPTrafficRESTURL; }

    public List<SWBean> getCPTraffic(VMBean sourceVM, List<SWBean> sourceSWes) {

        String tmpRESTUrl = CPTrafficRESTURL.replace("<controllerIP>", sourceVM.getAccessIPAddress());
        tmpRESTUrl = tmpRESTUrl.replace("<controllerPort>", sourceVM.getAccessHTTPPort());

        RESTConnectionUtil restConn = new RESTConnectionUtil();
        String rawResults = restConn.getRESTToSingleVM(sourceVM, tmpRESTUrl);

        RESTParser restParser = new RESTParser();
        sourceSWes = restParser.parseGetCPTraffic(rawResults, sourceSWes);

        return sourceSWes;
    }

}
```

그림 32. GettingCPTraffic.java 내용

```java
public class GettingCPULoad {
    //private final String SSHCommandCPULoad = "vboxmanage metrics query '*' CPU/Load/User:avg,CPU/Load/Kernel:avg";
    private final String SSHCommandCPULoad = "top -n 1 -b | head -1 | awk '{print $12}'";
    public GettingCPULoad() {
    }

    public String getSSHCommandCPULoad() { return SSHCommandCPULoad; }

    public HashMap<VMBean, Float> getCPULoad(PMBean sourcePM) {
        HashMap<VMBean, Float> resultmap = new HashMap<>();

        SSHConnectionUtil sshConn = new SSHConnectionUtil();
        SSHParser sshParser = new SSHParser();
        if(sourcePM.isAlive()) {

            List<VMBean> vmBeanList = sourcePM.getVmConfPool().getVmBeans();
            for (int index1 = 0; index1 < vmBeanList.size(); index1++) {
                VMBean tmpVm = vmBeanList.get(index1);
                if(tmpVm.isAlive()) {
                    String tmpResult = sshConn.sendCmdToSingleVM(tmpVm, SSHCommandCPULoad, 4096);
                    StringReader sr = new StringReader(tmpResult);
                    BufferedReader br = new BufferedReader(sr);
                    try {
                        tmpResult = br.readLine();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    tmpResult = tmpResult.replace("%n", "").replace(",", "");
                    resultmap.put(tmpVm, Float.valueOf(tmpResult));
                }
            }

            //String tmpResults = sshConn.sendCmdToSinglePM(sourcePM, SSHCommandCPULoad, 4096);
            //resultmap = sshParser.parseCPULoad(tmpResults, sourcePM);
        }

        return resultmap;
    }
}
```

그림 33. GettingCPULoad.java 내용

```
top - 18:00:29 up 2 days, 27 min,  1 user,  load average: 0.13, 0.08, 0.05
Tasks: 128 total,   2 running, 126 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5.0 us,  2.1 sy,  0.0 ni, 91.2 id,  0.6 wa,  0.0 hi,  1.2 si,  0.0 st
KiB Mem:   2048948 total,  1844244 used,   204704 free,   200404 buffers
KiB Swap:  2095100 total,    12948 used,  2082152 free.   697788 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2799 sdn       20   0 3236284 694792  21664 S   2.9 33.9  93:57.60 java
 1180 lightdm   20   0  441744  23704  17800 S   1.7  1.2   0:00.41 indicator-keybo
  957 root      20   0  230176  30528  18520 S   1.1  1.5   0:01.92 Xorg
  962 root      20   0  287460   6040   5560 S   1.1  0.3   0:00.11 accounts-daemon
 1135 lightdm   20   0  781472  30188  23332 S   1.1  1.5   0:08.25 unity-greeter
   18 root      20   0       0      0      0 R   0.6  0.0   0:11.43 rcuos/1
 1164 lightdm   20   0  178180   6724   4224 S   0.6  0.3   0:00.05 dconf-service
```

그림 34. top툴에서 파싱하는 CPU 부하

```
public class GettingTopology {
    public GettingTopology() {
    }

    public List<SWBean> getSwitches(VMBean sourceVM) {

        RESTConnectionUtil restConn = new RESTConnectionUtil();
        String cmdURL = "http://" + sourceVM.getAccessIPAddress() + ":" +
                sourceVM.getAccessHTTPPort() + "/onos/v1/devices";
        String restraw = restConn.getRESTToSingleVM(sourceVM, cmdURL);
        RESTParser parser = new RESTParser();
        return parser.parseGetSwitches(restraw);
    }

    public List<String> getMasterRoleSwitches(VMBean sourceVM) {

        RESTConnectionUtil restConn = new RESTConnectionUtil();
        String cmdURL = "http://" + sourceVM.getAccessIPAddress() + ":" + sourceVM.getAccessHTTPPort()
                + "/onos/v1/mastership/" + sourceVM.getIpAddress() + "/device";
        String restraw = restConn.getRESTToSingleVM(sourceVM, cmdURL);
        RESTParser parser = new RESTParser();

        return parser.parseGetMasterRoleSwitches(restraw);
    }
}
```

그림 35. GettingTopology.java 내용

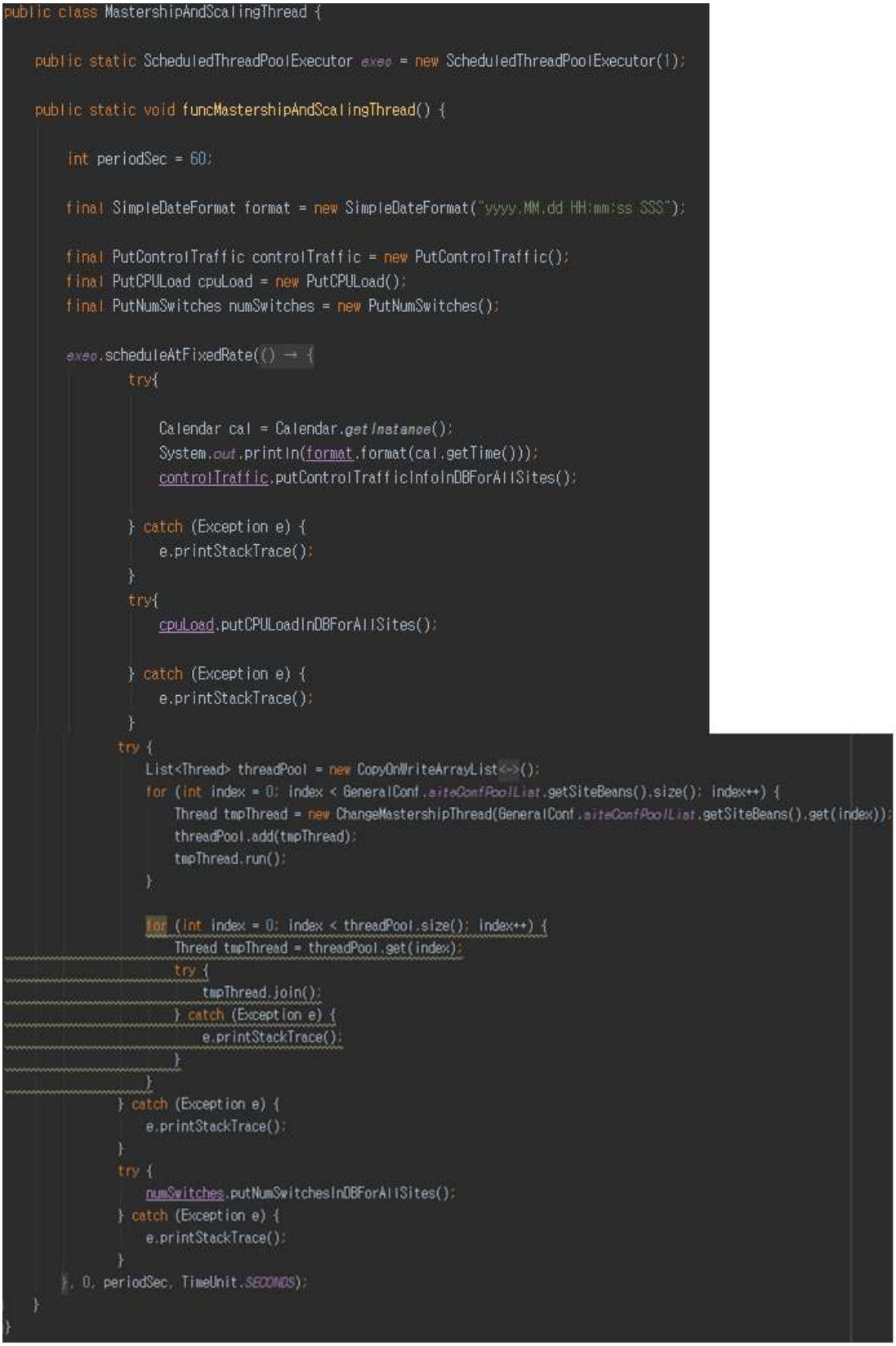
■ 자바 계층 코어 패키지 내 파서 서브 패키지에는 모니터링 정보 등으로 불러온 REST API의 JSON 문자열과 SSH로부터 받아온 문자열을 파싱하는 함수를 가지고 있다. JSON 문자열을 파싱하는 것은 JERSEY/JACSON 라이브러리를 활용하여 파싱하였다. SSH역시 원하는 정보만 자바의 문자열 파싱을 통해 파싱하였다. 그림 36은 RESTParser.java 파일 중 컨트롤러와 네트워크 장비 사이의 토폴로지를 파싱하는 예시 함수이다.

```
public List<String> parseGetMasterRoleSwitches(String jsonRaw) {
    List<String> parsingResults = new CopyOnWriteArrayList<>();

    if(jsonRaw == null) {
        return new CopyOnWriteArrayList<String>();
    }

    JsonObject parser = JsonObject.readFrom(jsonRaw);
    JsonArray switchObjs = parser.get("deviceIds").asArray();
    for (int index = 0; index < switchObjs.size(); index++) {
        String tmpSwitchID = switchObjs.get(index).asString();
        parsingResults.add(tmpSwitchID);
    }
    return parsingResults;
}
```

그림 36. RESTParser.java의 예시함수

■ 자바 계층 코어 패키지의 스레드 서브패키지에는 주기적으로 마스터쉽을 백그라운드에서 수행할 수 있도록하는 파일인 MastershipAndScalingThread.java 파일이 존재한다. 그림 37은 해당 파일의 일부를 가져온 내용이다. 스레드를 매 1분마다 구동하기 위해, scheduleFixRate라는 자바 함수를 사용하였다. 내부에는 먼저 제어 평면 메시지와 CPU 부하를 모니터링한 후, 이후에 마스터쉽을 변경하는 순서로 구성되어 있다. 마스터쉽을 변경하는 함수는 ChangeMastershipThread라는 함수로 구성이되어 있는데, 해당 함수는 앞서 언급한바와 같이, 욕심쟁이 알고리즘을 통해 각 평균 이상의 제어 평면 메시지를 송/수신하는 컨트롤러의 일부 네트워크 장비를 평균 이하의 제어 평면 메시지를 송/수신하는 컨트롤러로 이동시키는 알고리즘으로 구현되어있다. 자세한 사항은 Github 코드를 참조하면 될 것이다.

■ 마지막으로 각종 UI 및 컨트롤러 패키지에 있는 파일들은 저마다 UI를 어떻게 구성하는지에 따라 다르게 구현될 것이다. 본 기술문서에서는 해당 내용에 대한 자세한 설명은 생략하며, 해당 코드는 Github를 통해 공개되니 참조하면 될 것이다.



```
public class MastershipAndScalingThread {

    public static ScheduledThreadPoolExecutor exeo = new ScheduledThreadPoolExecutor(1);

    public static void funcMastershipAndScalingThread() {

        int periodSec = 60;

        final SimpleDateFormat format = new SimpleDateFormat("yyyy.MM.dd HH:mm:ss SSS");

        final PutControlTraffic controlTraffic = new PutControlTraffic();
        final PutCPULoad cpuLoad = new PutCPULoad();
        final PutNumSwitches numSwitches = new PutNumSwitches();

        exeo.scheduleAtFixedRate(() -> {
                try{

                    Calendar cal = Calendar.getInstance();
                    System.out.println(format.format(cal.getTime()));
                    controlTraffic.putControlTrafficInfoInDBForAllSites();

                } catch (Exception e) {
                    e.printStackTrace();
                }
                try{
                    cpuLoad.putCPULoadInDBForAllSites();

                } catch (Exception e) {
                    e.printStackTrace();
                }
               try {
                   List<Thread> threadPool = new CopyOnWriteArrayList<>();
                   for (int index = 0; index < GeneralConf.siteConfPoolList.getSiteBeans().size(); index++) {
                       Thread tmpThread = new ChangeMastershipThread(GeneralConf.siteConfPoolList.getSiteBeans().get(index));
                       threadPool.add(tmpThread);
                       tmpThread.run();
                   }

                   for (int index = 0; index < threadPool.size(); index++) {
                       Thread tmpThread = threadPool.get(index);
                       try {
                           tmpThread.join();
                       } catch (Exception e) {
                           e.printStackTrace();
                       }
                   }
               } catch (Exception e) {
                   e.printStackTrace();
               }
               try {
                   numSwitches.putNumSwitchesInDBForAllSites();
               } catch (Exception e) {
                   e.printStackTrace();
               }
        }, 0, periodSec, TimeUnit.SECONDS);
    }
}
```

그림 37. MastershipAndScalingThread.java 내용

4. MAS-Manv2

■ 상기에서 서술한 소프트웨어(이하 MAS-Manv1)은 멀티 스레드를 지원하나, 최초 디자인 시, 멀티 스레드 환경을 고려하지 않고 디자인되었다. 이러한 이유로, 모니터링 정보를 불러오거나 마스터쉽을 수행함에 있어 다소간의 시간이 낭비되었다. 한편, 기존의 MAS-Manv1의 경우, 모니터링 및 마스터쉽 변경은 지원하나 스케일링을 지원하지는 않는 문제가 있다. 이러한 문제를 해결하기 위해, 본 연구진은 멀티 스레드 기반의 동작 및 자동 스케일링 기능을 모두 지원하는 MAS-Manv2를 디자인 및 새로이 구현하였다.

4.1. 자동 스케일링 기능

■ 자동 스케일링이란, 소프트웨어 정의 네트워크 환경 하에서 컨트롤러의 자원이 부족하거나 과도한 경우 자동으로 컨트롤러를 생성하거나 종료하는 기능을 의미한다. 기존의 자동 스케일링 기능은 크게 두 가지의 타입으로 구분할 수 있다. 하나는 컨트롤러의 실제적인 동작은 멈추지 않은 채, 스위치의 마스터쉽만 변경하여, 특정 컨트롤러들이 서비스하는 스위치를 0개로 만드는 방법이다. 또 다른 방법은, 실제로 컨트롤러의 전원을 종료하는 방법이다.

■ 두 기법의 결정적인 차이는 실제적인 컨트롤러의 전원을 제어하느냐의 차이이다. 실제적으로 전원을 종료하는 경우, 물리 머신의 CPU 자원 낭비를 막을 수 있는 장점이 있다. 반면, 전원을 종료하지 않고 스위치의 마스터쉽을 제어하는 경우, 스케일링을 수행하는 시간이 줄어들어 실시간성/즉시성을 보장하는 장점이 있다.

■ 본 소프트웨어에서는 위의 기법을 모두 적용하여 구현하였다. 한편, 위 기법들의 성능적인 교환(trade-off)가 있음을 인지하고, S-DCORAL이라는 기법을 제안하였다. 그러나 본 기법의 경우, 아직 논문 작성 중인 관계로, 출판 후 본 기술문서에 상세히 기술 할 예정이다.

4.2. 디자인

■ MAS-Manv2의 기본적인 디자인은 그림 38과 같다. 사용자(user)가 본 소프트웨어를 사용한다고 할 때, 본 소프트웨어는 User Interface 모듈을 통해 사용자와 상호작용(interaction)을 수행한다. 현재, User Interface 모듈은 CLI(Command Line Interface)만을 지원하고 있으나 추 후 GUI기반 모듈을 구현할 계획이다.

■ User Interface 모듈은 본 소프트웨어의 모든 기능을 제어/관리하고 접근할 수 있는 Controller 모듈과 상호작용을 하도록 하였다. Controller 모듈은 기본적으로 2개의 기능을 제공한다. 하나는 모니터링 기능이고 하나는 알고리즘 동작 기능이다. 먼저 모니터링 기능은 말 그대로 컨트롤러의 마스터쉽 변경 및 스케일링을 제어를 위한 메트릭(metric)들을 모니터링 하는 기능이다. 한편 알고리즘 동작 기능은 실제적으로 마스터쉽 및 스케일링이 동작 해야하는지에 대한 의사 결정을

하는 알고리즘의 동작 기능을 의미한다.

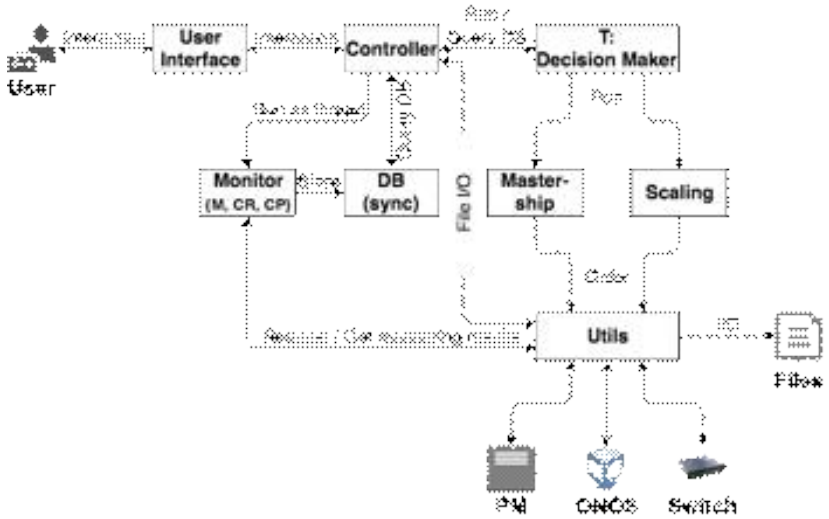


그림 38 Design of MAS-Manv2

■ 먼저 모니터링을 위한 기능은 크게 2개의 모듈로 구성되어 있다. Monitor와 DB가 그것이다. Monitor 모듈은 말 그대로 각 모니터링 메트릭을 실제적으로 불러오는 모듈을 의미한다. 이 때 모니터링 메트릭은 크게 네트워크 관련 메트릭과 컴퓨팅 관련 메트릭, 그리고 SDN 관련 메트릭으로 구분할 수 있다. 먼저 네트워크 관련 메트릭은 각 SDN 컨트롤러가 송/수신하는 메시지/패킷의 수 및 바이트이며, 컴퓨팅 관련 메트릭은 각 SDN 컨트롤러 및 이를 구동하는 물리 머신의 CPU 자원 및 메모리 자원 소모량을 의미한다. 마지막으로 SDN 관련 메트릭은 어떤 컨트롤러가 어떤 스위치를 서비스하는지를 나타내는 마스터쉽 토폴로지를 의미한다. 각 모니터링의 결과를 불러오기 위해서, Monitor 모듈은 Utils 모듈과 상호작용을 한다. Utils 모듈은 각 물리머신, 스위치, 그리고 SDN 컨트롤러와의 REST API 및 SSH 통신을 수행할 수 있는 통신 모듈과 위 통신 모듈을 통해 받아온 결과를 파싱하는 파싱 모듈로 나눌 수 있다. Monitor 모듈은 위의 Utils 모듈에서 받아온 정보를 DB 모듈에 저장한다.

■ 한편, 알고리즘 동작 모듈은 크게 Decision Maker, Mastership, 그리고 Scaling 모듈들로 구성되어 있다. Decision Maker 모듈에서는 현재 스케일링 및 마스터쉽 변경을 수행 해야하는지 여부를 결정하는 모듈이다. 만일 이 모듈이 마스터쉽 변경을 수행해야 한다고 결정하는 경우, Mastership 모듈을 호출하여 이를 변경한다. 반면, 스케일링을 수행해야 한다고 결정하는 경우, Scaling 모듈을 수행해서 스케일링을 수행한다.

■ Mastership 모듈이 마스터쉽 변경을 수행해야 하는 경우, MAS-Manv2는 REST API를 통해 SDN 컨트롤러에 명령을 전달해야 한다. 이를 위해, Mastership 모듈은 Utils 모듈에 있는 통신 모듈에 REST API 전달을 지시한다. 이 명령을 전달받은 통신 모듈은 SDN 컨트롤러에게 마스터쉽 변경을 지시한다.

■ 이와 달리, Scaling 모듈이 스케일링을 수행해야 하는 경우, Utils 모듈에 있는 통신 모듈에 REST API 전달 지시뿐만 아니라 SSH를 통한 명령 전달을 수행한다. Scaling 모듈이 특정 컨트롤러의 종료를 지시하는 경우, 해당 컨트롤러에 서비스를 받고 있는 스위치들의 마스터쉽을 다른 컨트롤러로 변경해야 하며, 이 후 컨트롤러의 서비스를 종료해야 한다. 전자를 위해서는 REST API를 통해 컨트롤러에 지시해야 하며, 후자를 위해서는 SSH를 통해 서비스 종료를 명령해야 한다. 한편, Scaling 모듈이 특정 컨트롤러의 실행을 지시하는 경우, 종료하는 경우와 반대로, SSH를 통해 서비스 시작을 지시하고, REST API를 통해 컨트롤러에 일정 수의 스위치를 할당해야 한다. 상기의 두 경우를 위하여, Scaling 모듈은 Utils 모듈과 상호작용 하여 스케일링을 수행한다.

■ 한편, Controller 모듈에서 수행하는 두 기능(모니터링과 알고리즘 동작 기능)은 병렬적으로 수행이 되며, 각 기능은 스레드 기반으로 구현되어 있다. 실제로, MAS-Manv2 내의 각 기능들은 하나의 SDN 컨트롤러 클러스터와 이 클러스터가 서비스하는 스위치 토폴로지만을 서비스하지 않고 다양한 위치에 있는 복수개 이상의 클러스터를 동시에 서비스 할 수 있도록 만든 오케스트레이터이다. 따라서, 위의 두 기능들이 각 컨트롤러 클러스터를 동시에 수행할 수 있도록 멀티 스레드 기반으로 구현이 되어 있다.

■ 멀티 스레드 기반으로 구동이 되는 경우, 가장 큰 문제는 동기화 문제와 경쟁 상태에 대한 핸들링이다. MAS-Manv2의 경우, Controller 모듈이 제공하는 두 기능이 동시에 접근하고자 하는 모듈은 바로 데이터베이스 모듈이다. 만일 모니터링 결과에 대한 저장과 의사 결정 모듈이 데이터베이스에서 정보를 불러오는 기능이 동시에 수행된다면, 동기화의 문제가 발생하게 된다. 이러한 문제를 해결하기 위해, 세마포어를 사용하여 동기화 문제 및 경쟁 상태 문제를 해결하였다.

4.3. Requirement list

■ 모니터링 기능 부분

- 각 컨트롤러의 제어 평면 메시지 수 모니터링
- 각 컨트롤러의 제어 평면 메시지의 byte수 모니터링
- 각 컨트롤러를 구동하는 가상 머신의 CPU 부하 모니터링
- 각 컨트롤러를 구동하는 가상 머신의 메모리 부하 모니터링
- 각 컨트롤러를 구동하는 가상 머신의 네트워킹 부하 모니터링
- 가상 머신을 실제 구동하는 물리 머신의 CPU 부하 모니터링
- 가상 머신을 실제 구동하는 물리 머신의 메모리 부하 모니터링
- 가상 머신을 실제 구동하는 물리 머신의 네트워킹 부하 모니터링
- 스위치들과 컨트롤러 사이의 마스터쉽 토폴로지 모니터링

■ 스케일링 기능 부분

  - 스케일 아웃 기능
  - 스케일 인 기능
- 마스터쉽 기능 부분
  - CPMan 알고리즘 구현
  - 마스터쉽 변경 기능 구현
- 모니터링 결과 기록 부분
  - 시간에 흐름에 따른 모니터링 결과 저장
  - 세마포어를 사용한 경쟁상태 해소
- 연결 모듈 부분
  - SSH 연결 기능
  - REST API 연결 기능
- 파싱 부분
  - SSH 결과 파싱
  - REST API로부터 받아온 JSON 파싱
- I/O 처리 부분
  - 데이터베이스 결과를 파일에 저장
  - 설정 파일을 데이터베이스에 저장
- 설정 부분
  - 물리 머신 정보 저장
  - 컨트롤러 정보 저장
  - 동작을 위한 각종 파라미터 값 저장 (예: 모니터링 주기 등)
  - 설정 파일은 JSON 포맷으로 관리

4.4. Packages and Classes

■ MAS-Manv2의 패키지는 다음과 같이 구성되어 있다. 먼저 사용자와의 상호 작용을 위한 UI, Controller 모듈을 위한 Controller package, Monitoring 모듈을 위한 Monitor package, 모니터링 결과를 저장하고 설정 파일을 저장하기 위한 database package, 알고리즘 구동 기능의 의사 결정을 위한 decision_maker package, mastership 변경을 위한 mastership package, 스케일링 기능을 위한 scaling package, 각종 유틸리티(예: 연결 모듈, 파싱 모듈, IO 처리 모듈)를 포함한 Utils package, 그리고 각 스위치/컨트롤러/물리 머신 등의 정보를 담기 위한 Beans package가 있다.

■ UI pakcage에는 cli.java 파일이 위치하며, 사용자와의 상호 작용을 위한 커맨드 라인 인터페이스를 제공한다.

■ Controller package에는 MAS-Manv2 소프트웨어의 핵심 모듈 및 모든 기능을 관할하는 controller.java 파일이 위치한다.

■ Database package에는 데이터베이스 서비스를 제공하며, 모든 테이블을 관리

하는 database.java파일, 각 튜플 값을 관리하는 Tuple.java (child: Mastershiptuple, ControlPlaneTuple, ComputingResourceTuple) 파일, 각 튜플 값을 특정 시간 단위로 저장하는 State.java 파일, 그리고 MAS-Manv2의 설정 값을 가지고 있는 Configure.java파일이 위치한다.

■ decisoon_maker package에는 알고리즘 동작의 의사 결정을 수행하는 DecisionMaker.java 파일이 위치한다.

■ Mastership package에는 컨트롤러가 서비스하는 스위치들의 마스터쉽을 변경할 수 있는 Mastership.java 파일이 위치하며, 하위 자식 클래스로 CPMan 알고리즘이 포함된 MastershipCPMan.java 파일이 위치한다.

■ Scaling package에는 스케일링을 수행할 수 있는 ControllerScaling.java 파일이 위치한다. 그리고 해당 파일의 자식 클래스로 ControllerScalingL1, ControllerScalingL2, 그리고 ControllerScalingL3 클래스가 존재한다.

■ Utils package에는 통신을 위한 Connection subpackage, 파싱을 위한 Parser subpackage, 그리고 파일 입출력을 위한 file_io subpackage가 존재한다. Connection subpackge에는 SSH 통신을 위한 SSHConnection.java 파일과 REST API 기반 통신을 위한 RESTConnection.java 파일이 존재한다. Parser subpackage에는 SSH 결과를 분석하는 SSHParser.java와 REST API 통신의 결과인 Json 파일을 분석하는 JsonParser.java 파일이 있다. 마지막으로 파일 입출력을 위한 file_io subpackage에는 FileIOUtil.java파일이 위치한다.

■ 각 장치들의 정보를 담는 클래스는 beans package에 위치하는데, 해당 패키지에는 Beans.java 파일이 위치한다. 이 파일의 자식 클래스는 물리 머신을 추상화한 PMBean.java, 컨트롤러를 추상화한 ControllerBean.java, 그리고 스위치 장비를 추상화한 SwitchBean.java파일이 위치한다.

5. 성능

■ 성능 평가를 위한 첫 번째 테스트는 MAS-Manv2를 포스텍 K-BOX에 두고 실험하였다. 4-POD Fat-tree 토폴로지를 6개 구동하여 대규모 네트워크를 에뮬레이션 하였다. 이 때 사용된 소프트웨어는 현재 널리 사용되고 있는 Mininet을 사용하였다. 그리고 이들 네트워크를 제어/관리하기 위해 ONOS 컨트롤러를 9개 구동하여 하나의 클러스터를 구성하였다. 매 3초마다 모니터링 결과를 바탕으로 마스터쉽 변경을 수행토록 한 결과, 아래의 그림과 같이 지연 없이 수행됨을 확인할 수 있었다.

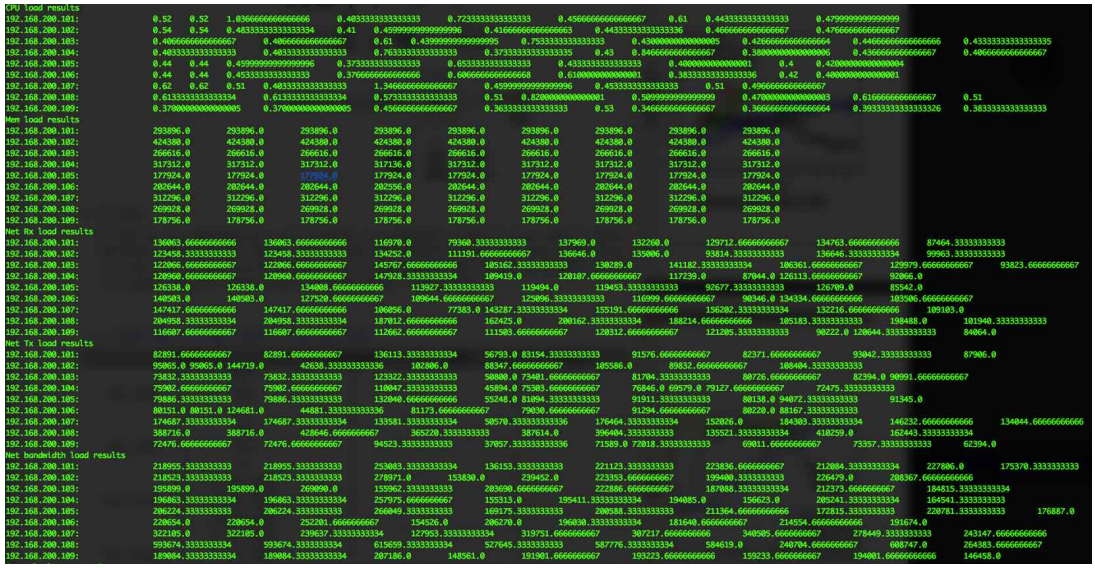


그림 39 모니터링 결과

■ 두 번째 테스트는, 72개의 스위치를 구동시키고 매 3초마다 Packet-in 메시지를 {30, 60, 90, 120, 150, 180, 210, 240}으로 점차 증가시켜 발생시켰다. 이러한 환경에서 MAS-Manv2의 스케일링 결과를 실험하였고, 그 결과는 다음과 같다. 이 때, ElastiCon Level은 스케일링의 정도를 의미한다. Level1은 SDN 컨트롤러를 종료하지 않고 마스터쉽만 변경한 경우(컨트롤러는 클러스터 내에 여전히 잔류), Level2는 SDN 컨트롤러 소프트웨어를 종료한 경우(VM은 여전히 구동 중이나, 해당 컨트롤러는 클러스터에서 배제된 상태), 마지막으로 Level3은 VM까지 완전히 종료된 것을 의미한다. 아래의 결과를 통해 MAS-Manv2가 S-DCORAL(해당 기법은 논문 출판 이후 업데이트 예정) 및 각 Level별 ElastiCon 기법이 정상적으로 구동되고 있음을 보여준다. 특히, 플로우 수가 시간의 흐름에 따라 점차 증가하더라도, CDF 그래프(하단 우측)가 2~3초 후부터 기울기가 일정해 짐을 알 수 있었다. 즉 컨트롤러의 스케일링이 동작하여, 각 컨트롤러가 받는 부하가 그 이후부터 점차 일정해짐을 확인 할 수 있다.

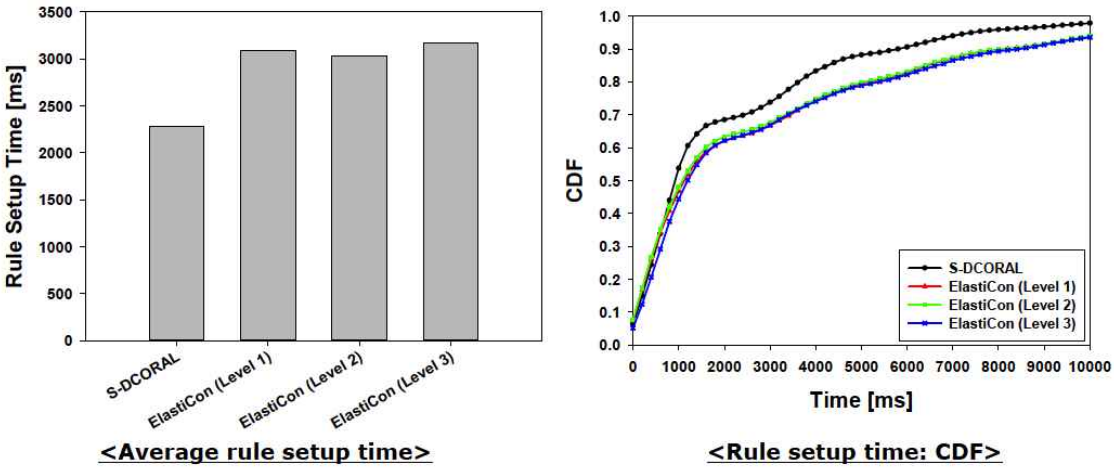


그림 40 MAS-Manv2의 성능 평가 결과

6. 결론

■ 본 기술문서에서는 분산형 SDN 컨트롤러의 개념 및 해당 컨트롤러에서 발생할 수 있는 문제에 대해 간단히 알아보았다. 그리고 이러한 문제를 해결하기 위해, 전년차에서 개발한 소프트웨어의 서술을 하였고, 이를 발전시킨 MAS-Manv2 소프트웨어의 서술을 수행하였다.

■ 결과에서 볼 수 있드시, MAS-Manv2는 기존의 버전 1과 달리 처음부터 멀티 스레드를 기반으로 디자인되어, 짧은 시간마다 모니터링을 수행해도(성능 섹션에서는 매 3초마다 진행) 끊김없는 모니터링 결과를 확인할 수 있었다.

■ 한편, 본 소프트웨어에 적용된 자동 스케일링 기법인 ElastiCon 기법 및 DCORAL 기법 역시 모두 정확히 동작함을 확인할 수 있었다.

■ 본 연구진은 해당 소프트웨어를 K-ONE GitHub에 기 업로드 하였으며, 현재 실행을 위한 문서화 작업을 진행 중이다. 이와 함께, 현재 본 소프트웨어에 적용된 DCORAL 기법에 대한 연구 및 논문 작업을 진행하고 있다.

■ 추 후, 좀 더 정교한 방법의 스케일링 알고리즘을 디자인하여, 본 소프트웨어에 추가할 예정이다. 이와 별도로, 현재 구현된 마스터쉽 기법은 현재 ONOS 컨트롤러에 통합을 위해 실제 구현 진행 중에 있다.

References

[1] Open Networking Foundation, "OpenFlow Switch Specification," *Technical Specification*, Apr. 2015.

[2] Open Networking Foundation (ONF), [online] Available: https://www.opennetworking.org.

[3] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *IEEE Proceddings of the IEEE*, vol. 103, no. 1, pp. 14-76, Jan. 2015.

[4] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Cornnor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Toward an Open, Distributed SDN OS," in Proc. *ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2014.

[5] ON. Lab, "ONOS Architecture Guide," *Technical Document*, Jan. 2015.

[6] D. Levin, A. Wundsam, Br. Heller, N. Handigol, and A. Feldmann, "Logically Centralized?: State Distribution Trade-offs in Software Defined Networks," in Proc. ACM Hot Topics in Software Defined Networks (HotSDN), pp. 1-6, 2012.

[7] B. Heller, R. Sherwood, and N. McKeown, "The Controller Placement Problem," in Proc. ACM Hot Topics in Software Defined Networks (HotSDN), pp. 7-12, 2012.

[8] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic Controller Provisioning in Software Defined Networks," in Proc. International Conference of Network and Service Management (CNSM), pp. 18-25, 2013.

[9] J. Li, J.-H. Yoo, and J. W.-K. Hong, "Dynamic Control Plane Management for Software-defined Networks," International Journal of Network Management, vol. 26, no. 2, pp. 111-130, 2016.

K-ONE 기술 문서

- 이 문서의 기술적인 내용은 프로젝트의 진행과 함께 별도의 예고 없이 변경될 수 있습니다.

- 본 문서와 관련된 문의 사항은 아래의 정보를 참조하시길 바랍니다.
  (Homepage: http://opennetworking.kr/projects/k-one-collaboration-project/wiki, E-mail: k1@opennetworking.kr)

작성기관: K-ONE Consortium
작성년월: 2018/03