ISBN : 978-89-6211-681-6

한국과학기술정보연구원

플로우라우팅 기술 개발을 위한 NOX 분석

일 자: 2010년 11월 12일
부 서: 슈퍼컴퓨팅 본부/융합자원실
제출자: 권윤주, 홍원택

<목차>

1. 서론

미래 인터넷이라는 키워드가 등장하면서 네트워크 운영 기술에 많은 변화가 오고 있다. 특히 네트워크 가상화 또는 동적 라우팅이라는 개념이 실제 연구망에서 개발되고 시범 적용되어 가며 첨단 응용을 지원하기 위한 하나의 기술로서 자리잡고 있다.

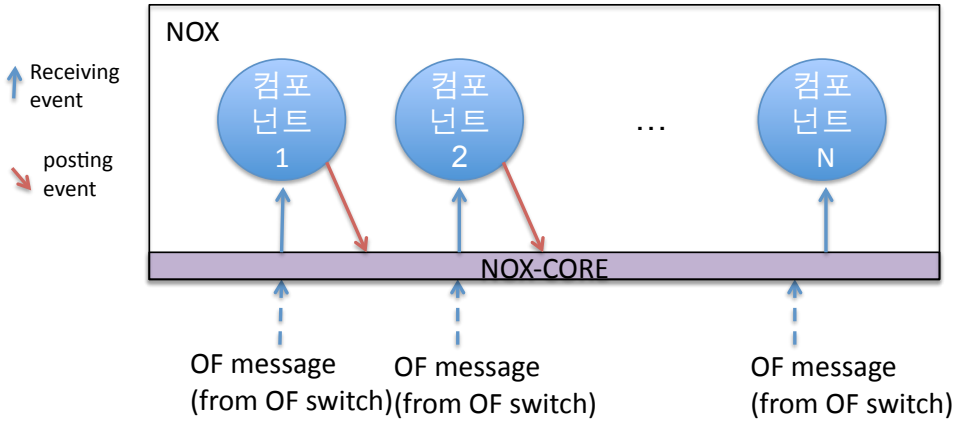
미래 인터넷의 큰 흐름에 기존 네트워크의 패러다임을 바꾸고 있는 OpenFlow라는 기술이 등장했다. 이 기술은 네트워크 상에서 IP 주소만을 가지고 패킷을 라우팅을 하는 것이 아닌 응용 특성에 따라 정의된 메트릭으로 패킷을 라우팅하고, 패킷 포워딩(switch) 역할과 라우팅 결정(controller : NOX) 역할을 분리하여 실제 망 운영에 적용하게 하였다.

우리 연구망에서는 첨단 응용들에 대한 차세대 연구망 서비스로서 OpenFlow 인프라를 구축하여 네트워크 자원할당 뿐만이 아닌 응용에 따른 플로우라우팅 기술을 제공하고자 하고 있다. 이에 본 문서에서는 지금까지 구현되어 있는 NOX의 구조를 파악하고 OF 스위치와의 연동 방식을 분석함으로써 NOX의 전체 구조 및 개발 방식을 정리하고자 한다.

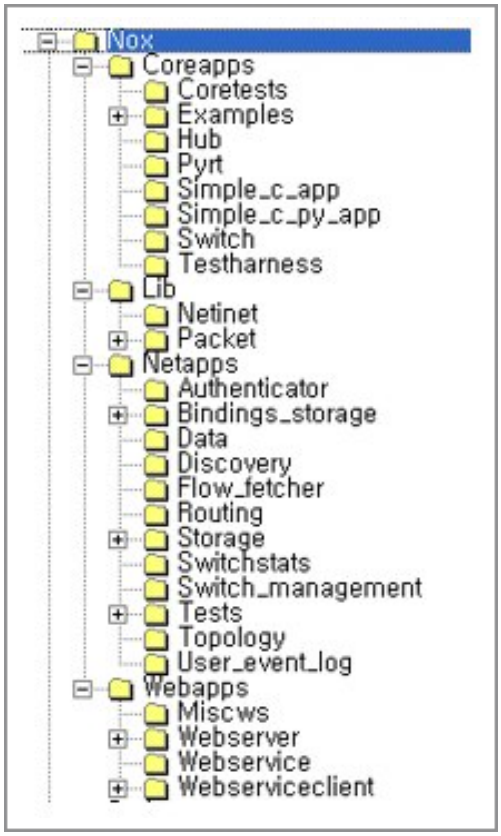
본 문서는 스탠포드 대학에서 배포한 NOX(버전 0.8.9) 구현 코드를 중심으로 NOX의 분석내용이 기술하는 것으로 범위가 한정되어있다.

2. NOX 의 구성 요소

OpenFlow Switch Network에서 데이터 전송 경로를 결정하는 NOX는 다양한 방식의 전송 경로에 대한 정의를 수용하기 위하여 하나의 프레임워크를 가지고 있다. 따라서 누구든지 NOX의 프레임워크를 이용하여 자신만의 방식의 데이터 전송 경로 결정 알고리즘을 구현할 수 있다.



NOX는 OF 스위치로부터 다양한 OF message들을 수신받아 네트워크 토폴로지 구성, 종단간 라우팅 경로등을 결정할 수 있다. NOX가 하는 하나하나의 역할은 컴포넌트로 구현되며, 컴포넌트는 NOX에 필요한 기능들을 구현한 단위 객체를 의미한다. 그리고 스위치와 컴포넌트간, 컴포넌트와 컴포넌트가 전달할 정보는 이벤트 형태로 보내진다. 모든 이벤트는 NOX-CORE에서 수신되어 해당 이벤트 수신을 원하는 컴포넌트로 전달된다. 이벤트들은 OpenFlow message에 의해 직접적으로 발생되는 다수의 core 이벤트들이 존재하고, 또 컴포넌트가 정의하고 발생시키는 컴포넌트 정의 이벤트(application event)가 존재할 수 있다.



(1)컴포넌트

NOX에 대한 기본적인 기능을 갖는 컴포넌트들은 왼쪽 그림과 같이 src/nox/apps에 구현되어 있다. 각각의 컴포넌트들은 C++, Python, C++과 Python 혼합 형태로 구현될 수 있다. 혼합된 형태의 컴포넌트는 본 문서에서는 논외로 다루고 본 장에서는 C++과 Python으로 컴포넌트 만드는 방법에 대하여 살펴본다.

1) C++ 컴포넌트 만들기

하나의 컴포넌트는 Component 클래스로부터 상속받아야 하고, dynamic loader를 돕기 위한 external linkage를 생성하는 REGISTER_COMPONENT macro를 포함해야 한다. “configure”와 “install”은 load time에 호출되며, 이벤트나 이벤트 핸들러를 등록하기 위해 사용된다.

또한 컴포넌트는 해당 컴포넌트와 같은 디렉토리에 meta.xml 파일이 있어야만 한다. 각 컴포넌트는 meta.xml을 통해 필요한 컴포넌트들을 정의하고, 의존관계를 정의함으로써, NOX에게 해당 컴포넌트 구동시 필요한 타 컴포넌트에 대한 정보를 알려준다.

컴포넌트는 이벤트를 통해서 또는 직접적으로 컴포넌트간 통신할 수 있다. 컴포넌트를 직접 접근하기 위해서, ‘resolve” method를 사용한다.

2) Python 컴포넌트 만들기

Python만으로 만들어진 컴포넌트는 C++ 컴포넌트에 비해 훨씬 간단하다. Python 컴포넌트는 다음의 구조를 가지고 있다. 선택적으로 C++에서처럼 configure() 를 추가해도 된다.

```
from nox.lib.core import *

class foo(Component):

    def __init__(self, ctxt):
        Component.__init__(self, ctxt)

    def install(self):
        # register for event here
        pass

    def getInterface(self):
        return str(foo)


def getFactory():
    class Factory:
        def instance(self, ctxt):
            return foo(ctxt)

    return Factory()
```

핵심 python API는 nox/lib/core.py, util.py에 존재하고, 다른 컴포넌트들에 대한 핸들을 얻기 위해서는 Component.resolve(...) method 사용한다.

(2) 이벤트

앞서 언급한 것처럼 이벤트(Event)들은 NOX를 구동시키는 역할을 한다. 이러한 이벤트들에는 OpenFlow메시지로부터 직접적으로 생성되는 core 이벤트들이 있고, 컴포넌트들이 기존 이벤트들을 기반으로 고수준으로 생성된 application 이벤트들이 있을 수 있다. 이번 장에서는 이벤트들의 종류와 이벤트 및 이벤트 핸들러 등록 방법에 대하여 설명한다.

1) 이벤트의 종류

▷ core 이벤트의 종류

✓ Datapath_join_event : 새로운 스위치가 OF 망에 감지될 때 발생하며, OFPT_FEATURES_REPLY메시지 내용을 담고 있다.

[표] Datapath_join_event 구조체(src/include/datapath-join.h)

```
struct Datapath_join_event
: public Event,
  public Ofp_msg_event
{
        Datapath_join_event(const ofp_switch_features *osf, std::auto_ptr<Buffer> buf);
        // only for use within python
        Datapath_join_event() : Event(static_get_name()){}

        static const Event_name static_get_name(){return "Datapath_join_event";}
        datapathid datapath_id;     /****                                    ****/
        uint32_t n_buffers;         /****                                          ****/
        uint8_t n_tables;           /**** OFPT_FEATURES_REPLY 의 메시지 내용 ****/
        uint32_t capabilities;              /****                                          ****/
        uint32_t actions;           /****                                          ****/
        std::vector<Ports> ports;   /****                                          ****/
        Datapath_join_event(const Datapath_join_event&);
}
```

✓ Datapath_leave_event : 새로운 스위치가 OF 망을 떠날 때 발생하며, 이 이벤트는 OF 스위치와 NOX간 secure channel이 close될 때 생성된다. Datapath_leave이벤트 생성코드와 Datapath_leave_event 구조체 내용은 다음 표와 같다.

[표] Datapath_leave 이벤트 생성 코드(builtin/nox.cc 중 일부)

```
bool Conn::close() {
        if(!closing){
                closing = true;
                datapathid dp_id = oconn->get_datapath_id();
                Datapath_leave_event * dple = new Datapath_leave_event(dp_id);
                connection_map.erase(dp_id);
                post_event(dple);
                main_loop->remove_pollable(this);
                if(!poll_cnt){
                        delete this;
                }
        }
}
```

[표] Datapath_leave_event 구조체 (src/include/datapath-leave.h)

```
struct Datapath_leave_event
        : public Event
{
        Datapath_leave_event(datapathid datapath_id)
                Event(static_get_name()), datapath_id(datapath_id_){}
        // only for use within python
        Datapath_leave_event() : Event(static_get_name()){}
                static const Event_name static_get_name(){ return "Datapath_leave_event";}

        datapathid datapath_id;
private:
        Datapath_leave_event(const Datapath_leave_event&);
        Datapath_leave_event& operator=(const Datapath_leave_event&);
}
```

✓ Packet_in_event : NOX에 새로운 패킷이 수신되었을 때, 즉 OF 스위치로부터 OFTP_PACKET_IN 메시지가 수신되었을 때 Packet_in 이벤트는 발생한다.

[표] packet_in_event 구조체 (src/include/packet-in.hh)

```
struct Packet_in_event
   : public Event,  public Ofp_msg_event
{
   Packet_in_event(datapathid datapath_id_, uint16_t in_port_, std::auto_ptr<Buffer> buf_, size_t total_len_,
                uint32_t buffer_id_, uint8_t reason_)
           : Event(static_get_name()), Ofp_msg_event((ofp_header*) NULL, buf_), datapath_id(datapath_id_),
           in_port(in_port_), total_len(total_len_), buffer_id(buffer_id_), reason(reason_) {}

   Packet_in_event(datapathid datapath_id_, uint16_t in_port_, boost::shared_ptr<Buffer> buf_, size_t total_len_,
                uint32_t buffer_id_, uint8_t reason_)
           : Event(static_get_name()), Ofp_msg_event((ofp_header*) NULL, buf_), datapath_id(datapath_id_),
           in_port(in_port_), total_len(total_len_), buffer_id(buffer_id_), reason(reason_) {}

   Packet_in_event(datapathid datapath_id_, const ofp_packet_in *opi, std::auto_ptr<Buffer> buf_)
           : Event(static_get_name()), Ofp_msg_event(&opi->header, buf_), datapath_id(datapath_id_),
           in_port(ntohs(opi->in_port)), total_len(ntohs(opi->total_len)), buffer_id(ntohl(opi->buffer_id)),
           reason(opi->reason) {}

   virtual ~Packet_in_event() { }

   const boost::shared_ptr<Buffer>& get_buffer() const { return buf; }

   static const Event_name static_get_name() { return "Packet_in_event";  }

   datapathid datapath_id;
   uint16_t in_port;
   size_t   total_len;
   uint32_t buffer_id;
   uint8_t  reason;

   Packet_in_event(const Packet_in_event&);
   Packet_in_event& operator=(const Packet_in_event&);
};
```

✓ Port_status_event : 포트의 상태변화와 관련(dis/enabled, speed, port name)된 이벤트로서 OF 스위치로부터 OFPT_PORT_STATUS메시지를 수신하였을 때 발생되는 이벤트이다.

[표] Port_status_event 구조체(src/include/port_status.h)

```
struct Port_status_event
: public Event,  public Ofp_msg_event
{       Port_status_event(datapathid datapath_id_, uint8_t reason_, constPort& port_)
            : Event(static_get_name()), reason(reason_), port(port_), datapath_id(datapath_id_){}
        Port_status_event(datapathid datapath_id_, const ofp_port_status *ops, std::auto_ptr<Buffer> buf)
            :Event(static_get_name()),Ofp_msg_event(&ops->header,buf),  reason(ops->reason), port(&ops->desc), datapath_id(datapath_id_){}

        // only for use within python
        Port_status_event() : Event(static_get_name()){}
        static const Event_name static_get_name(){return "Port_status_event";}

        uint8_t reason;     /****  OFPT_PORT_STATUS 메시지 내용  ****/
        Port port;          /****  OFPT_PORT_STATUS 메시지 내용  ****/
        datapathid datapath_id;

        Port_status_event(const Port_status_event&);
        Port_status_event& operator=(const Port_status_event&);
}
```

▷ Application 이벤트

✓ Link_event : 새로운 링크가 발견될 때 발생하는 이벤트로서 Discovery 어플리케이션에서 생성한다.

※참고 (nox/netapps/discovery/link_event.h)

```
struct Link_event
: public Event,  boost::noncopyable
{       enum Action{ ADD, REMOVE  }
        Link_event(datapathid dpsrc_, datapathid dpdst_, uint16_t sport_, uint16_t dport_, Action action_);
        // -- only for use within python
        Link_event();

        datapathid dpsrc;
        datapathid dpdst;
        uint16_t sport;
        uint16_t dport;
        Action action;
}
```

✓ flow_in_event : routing 어플리케이션이 실행될 때, packet_in_event로 들어온 패킷내용을 플로우 라우팅을 위하여 flow_in_event로 생성

※참고 (nox/netapps/authenticator/flow_in.h)

```
struct Flow_in_event
    : public Event
{
    Flow_in_event(const timeval& received_,
                  const Packet_in_event& pi,
                  const Flow& flow_);

    Flow_in_event()
        : Event(static_get_name()) { }

    ~Flow_in_event() { }

    static const Event_name static_get_name() {
        return "Flow_in_event";
    }

   struct DestinationInfo {
        AuthedLocation              authed_location;
        bool                        allowed;
        std::vector<uint32_t>       waypoints;
        hash_set<uint32_t>          rules;
    };

    typedef std::vector<DestinationInfo> DestinationList;

    // 'active' == true if flow can still be "acted" upon else it has been
    // consumed by some part of the system.

    bool                                active;
    bool                                fn_applied; //if a function consumed the flow
    timeval                             received;
    datapathid                          datapath_id;
    Flow                                flow;
    boost::shared_ptr<Buffer>                   buf;
    size_t                              total_len;
    uint32_t                            buffer_id;
    uint8_t                             reason;

    boost::shared_ptr<Host>                     src_host;
    AuthedLocation                      src_location;
    boost::shared_ptr<std::vector<uint32_t> > src_addr_groups;

    bool                                dst_authed;
    boost::shared_ptr<Host>                     dst_host;
    DestinationList                     dst_locations;
    boost::shared_ptr<std::vector<uint32_t> > dst_addr_groups;

    boost::shared_ptr<Location>                 route_source;
    std::vector<boost::shared_ptr<Location> > route_destinations;
    uint32_t                            routed_to;

    Flow_in_event(const Flow_in_event&);
    Flow_in_event& operator=(const Flow_in_event&);

}; // class Flow_in_event
```

2) 이벤트 핸들러 등록 방법

▷ C++에서의 이벤트 핸들러 등록 방법

C++에서의 이벤트 핸들러 등록

```
Disposition handler(const Event&e)
{
        return CONTINUE;
}
void install()
{
        register_handler<Packet_in_event>(boost::bind(handler, this, _1));
}
```

각 컴포넌트에서는 Component::register_handler 메소드를 이용하여 대응하고자 하는 이벤트에 대한 핸들러를 등록한다. 모든 핸들러에서는 Disposition(src/include/event.hh에 정의) 형을 리턴하게 되어 있는데, 해당 이벤트를 다음 리스너(next listener)로 보내려면 **CONTINUE**를, event chain을 멈추도록 하기 위해서는 **STOP**을 반환한다.

▷ Python에서의 이벤트 핸들러 등록

Python에서 핸들러를 등록하는 방법은 C++과 유사하다. src/nox/lib/core.py에 정의되어 있는 register_handler를 이용하여 이벤트 핸들러를 등록한다.

Python에서의 이벤트 핸들러 등록

```
def handler(self) :
        return CONTINUE
def install(self) :
        self.register_handler(Packet_in_event.static_get_name(), handler)
```

3) 이벤트 포스팅

어플리케이션은 다른 어플리케이션들이 다룰 수 있도록 이벤트들을 생성하여 포스팅할 수 있다. C++와 Python에서 이벤트들을 포스팅하는 방법은 각각 아래와 같다. 그 외, 타이머와 관련하여 post method를 이용하는 경우가 있다.

C++에서의 이벤트 등록

```
post(new Flow_in_event(flow, *src, *dst, src_dl_authed, src_nw_authed, dst_dl_authed, dst_nw_authed, pi));
```

Python에서의 이벤트 등록

```
e = Link_event(...)
self.post(e)
```

3. 핵심 NOX 네트워크 API

본 장은 핵심 NOX 네트워크 API 일부를 소개한다.

(1) Filtering input traffic : register_handler_on_match method

본 메소드는 NOX에서 수신된 트래픽들 중 관심 트래픽을 필터링 하기 위해 이용한다. 그리고 이것은 Packet_expr 과 함께 호출되어야 하는 데, Packet_expr은 매칭되어야하는 패킷 필드를 정의하는 역할을 한다. 참고로 이 핸들러는 기본 핸들러와 달리 Disposition 형을 반환하지 않는다.

예) TCP 패킷만을 받고 싶을 때

```
// only call on TCP packets
Packet_expr expr;
uint32_t val = ethernet::IP;
expr.set_field(Packet_expr::DL_TYPE,  &val);
val = ip_::proto::TCP;
expr.set_field(Packet_expr::NW_PROTO, &val);
register_handler_on_match(100, expr, boost::bind(&Example::tcp_packet_handler, this, _1));
```

(2) Managing switch flow tables : send_openflow_command method

어플리케이션은 스위치에 플로우 엔트리를 추가/삭제함으로써 네트워크상의 포워딩을 관리한다. NOX는 OpenFlow 프로토콜에 의해 지원되는 모든 기능들을 노출하고 있다. 일반적으로 아래와 같다.

```
int send_openflow_command(const datapathid&, const ofp_header*, bool block) const;
```

(3) Sending packets : send_openflow_packet

어플리케이션들은 아래의 method들을 이용하여 네트워크상에 패킷들을 전송할 수 있다. 첫 번째 method는 buffer_id를 받아들여 OF 스위치가 내에 buffer_id로 저장되어 있는 패킷을 데이터패스상에 전송하도록 명령을 줄 때 사용한다. 두 번째 method는 어플리케이션이 네트워크 상에 임의의 패킷을 구성하여 보내기 위해 사용한다. method이름은 같지만 전달받는 파라미터에 따라 기능이 다르므로 구별하여 사용할 수 있도록 해야 한다.

```
// Instruct a switch to send a packet it has buffered
int send_openflow_packet(const datapathid&, uint32_t buffer_id,  uint16_t out_port, uint16_t in_port, bool block)
    const;

// Send a packet in a Buffer object out on the network
int send_openflow_packet(const datapathid&, const Buffer&, uint16_t out_port, uint16_t in_port,  bool block) const;
```

(4) Accessing switch table and port statistics

스위치 통계 정보는 statistics 요구 메시지들을 보냄으로써 수집될 수 있다.

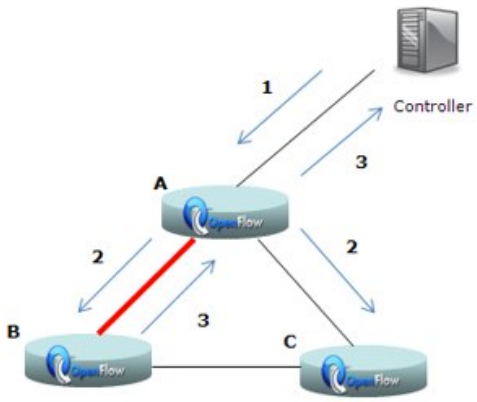
4. 주요 어플리케이션(컴포넌트) 분석

(1) Discovery

1) Discovery 어플리케이션 개요

Discovery는 OpenFlow 네트워크의 연결성 변화를 탐지하여 네트워크 토폴로지를 유추하고 “Link Event”를 생성하는 어플리케이션이다. OF 망에 존재하는 모든 스위치들에 대해 LLDP 패킷의 생성/파싱/해석을 모두 다룸으로써 링크에 대한 Discovery 역할을 수행한다.

[그림] *Discovery* 어플리케이션의 네트워크 *Discovery* 메카니즘



Discovery 어플리케이션은 위의 그림에서 보는 것과 같이 다음과 같은 프로세스로 네트워크 연결성을 체크한다.

① NOX는 연결된 OF 스위치들에게 packet_out 메시지를 이용하여 LLDP 패킷을 전송한다.

② A 스위치를 통하여 LLDP 패킷을 받은 B 스위치는 packet_in 메시지를 이용하여 LLDP 패킷을 NOX에 다시 전송한다.

③ 이러한 LLDP 패킷을 packet_in 메시지를 통해 받은 NOX는 A와 B 스위치가 연결되어 있음을 알게 된다.

④ 추가적으로 NOX는 연결된 링크에 대해 일정 간격마다 링크의 연결을 확인하고, LLDP가 packet_in 메시지로 돌아오지 않으면, 해당 링크는 단절되었음을 알게 된다.

2) Discovery 어플리케이션의 기능 분석

본 어플리케이션은 install()에 작성되어 있는 이벤트들을 처리함으로써 OpenFlow 네트워크의 변화를 반영하여 네트워크 정보를 관리한다. Discovery 어플리케이션에서 처리하는 이벤트들은 다음 표와 같다. 특히 LLDP 패킷을 받아들이기 위하여 register_for_packet_match()라는 함수를 사용하였다.(register_for_packet_match()는 core.py에 정의되어 있음))

```
def install(self):
    self.register_for_datapath_join ( lambda dp,stats : discovery.dp_join(self, dp, stats) )
    self.register_for_datapath_leave( lambda dp       : discovery.dp_leave(self, dp) )
    self.register_for_port_status( lambda dp, reason, port : discovery.port_status_change(self, dp, reason, port) )
    # register handler for all LLDP packets
    match = {DL_DST : array_to_octstr(array.array('B',NDP_MULTICAST)),\
            DL_TYPE: ethernet.LLDP_TYPE}
    self.register_for_packet_match(lambda
        dp,inport,reason,len,bid,packet:
        discovery.lldp_input_handler(self,dp,inport,reason,len,bid,packet),
        0xffff, match)

    self.start_lldp_timer_thread()
```

그림 *Discovery* 어플리케이션에서 처리하는 이벤트들

▷ 주요 인스턴스 변수들

✓ self.dps: OF 노드마다 생성되는 datapathid를 리스트화 시켜 유지
✓ self.lldp_packets: OF 노드에 존재하는 각 포트마다 생성되는 lldp packets 리스트 유지
✓ self.adjacency_list: linktuple 단위로 링크 리스트 정보유지, timestamp와도 관련

▷ Discovery 어플리케이션의 core functions

✓ add_link() : Link_event.ADD action을 갖고, Link_event를 발생시킴
✓ delete_links() : deleteme에 있는 링크들에 대해 adjacency_list에서 삭제하고, Link_event.REMOVE action을 갖고, Link_event를 발생시킴.
✓ create_discovery_packet() : lldp packet을 생성하는 유틸리티 함수

▷ Discovery 어플리케이션의 이벤트 처리 루틴

✓ Datapath_join 이벤트(dp_join()) : datapath join 시, 포트마다 새로운 LLDP 패킷을 생성함 (create_discovery_packet 호출)
✓ Datapath_leave 이벤트(dp_leave(self, dp)) : datapath leave 시, 모든 관련된 링크들을 삭제함(delete_links 호출). self.dps, self.lldp_packets, self.adjacency_list에서 dp와 관련된 정보를 모두 삭제함.
✓ LLDP packet_in 이벤트(lldp_input_handler()) : LLDP 패킷의 수신시 호출되는 packet_in 핸들러 함수. 네트워크 링크들은 인스턴스 변수인 adjacency_list에 저장됨. 유효한 LLDP packet_in 인 경우 아래와 같은 단계를 거침. self.adjacency_list에 존재하지 않는 linktuple을 add_link()로 추가하고, timestamp를 업데이트함.

```
# print 'LLDP packet in from',longlong_to_octstr(chassid),' port',str(portid)

linktuple = (dp_id, inport, chassid, portid)

if linktuple not in self.adjacency_list:
    self.add_link(linktuple)
    lg.warn('new link detected ('+longlong_to_octstr(linktuple[0])+' p:'\
            +str(linktuple[1]) +' - > '+\
            longlong_to_octstr(linktuple[2])+\
            ' p:'+str(linktuple[3])+')')


# add to adjaceny list or update timestamp
self.adjacency_list[(dp_id, inport, chassid, portid)] = time.time()
```

✓ port_status_change 이벤트(port_status_change(), nox5.0까지는 포함되어 있지 않은 내용) : 포트들이 추가/제거되는 경우 LLDP 패킷 리스트를 업데이트함. dp는 해당포트의 datapath ID에 해당

▷ Discovery 어플리케이션의 Discovery 루틴

Discovery 어플리케이션은 이벤트 처리 외에 일정 주기마다 LLDP 패킷을 보내어 네트워크의 연결상태 변화를 관리한다. start_lldp_timer_thread()가 주기적인 네트워크 연결상태 관리 작업을 관장한다.

✓ start_lldp_timer_thread() : LLDP_SEND_PERIOD마다 LLDP 패킷을 보내는 역할. send_lldp, build_lldp_generator를 매번 실행하는 구조.

✓ send_lldp() : timeout(LLDP_SEND_PERIOD) 주기마다 timer에서 호출되는 generator 함수. 관리하는 네트워크 상의 모든 포트들에 대해 반복하고, 각 호출시 LLDP 패킷을 전송함. LLDP 패킷은 chassis ID, outgoing 스위치/포트 정보를 포함. --> self.send_openflow_packet 호출

```
def send_lldp (packets):
  for dp in packets:
    # if they've left, ignore
    if not dp in self.dps:
      continue
    try:
      for port in packets[dp]:
        #print 'Sending packet out of ',longlong_to_octstr(dp), ' port ',str(port)
        self.send_openflow_packet(dp, packets[dp][port].tostring(), port)
        yield dp
    except Exception, e:
      # catch exception while yielding
      lg.error('Caught exception while yielding'+str(e))
```

✓ build_lldp_generator: g()안에서 send_lldp를 호출함. 그 외에, start_lldp_timer_thread에서 두 개의 post_callback 각각 설정(build_lldp_generator, discovery.timeout_links)

✓ timeout_links(): TIMEOUT_CHECK_PERIOD 주기로 self.adjacency_list에 있는 linktuple들의 값(시간)을 확인하여 LINK_TIMEOUT을 넘어선 linktuple들을 deleteme 리스트에 등록해놓은 후, adjacency_list의 모든 link_tuple에 대한 확인이 종료되면 delete_links() 호출하여 일정시간(LINK_TIMEOUT)이상 응답이 없는 링크에 대한 삭제를 수행한다.

(2) Topology

1) Topology 어플리케이션 개요

Topology는 OpenFlow 네트워크의 구조를 유지 및 관리하는 어플리케이션이다.

Topology의 meta.xml은 다음과 같이 정의되어 있다.

[표] Topology의 meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<components:components xmlns:components="http://www.noxrepo.org/components.xsd">
<component>
<name>topology</name>
<library>topology</library>
<dependency>
<name>discovery</name>
</dependency>
</component>
...
</components:components>
```

meta.xml에서 보는 것처럼, Topology 어플리케이션은 discovery 어플리케이션에 의존적이다.

2) Topology 어플리케이션 기능 분석

본 어플리케이션에 대한 분석은 netapps/topology 디렉토리 내에 있는 topology.cc를 토대로 수행되었다. 본 어플리케이션은 configure()에 작성되어 있는 이벤트들을 처리함으로써 OpenFlow 네트워크의 변화를 반영하여 topology 정보를 관리한다. Topology 어플리케이션에서 처리하는 이벤트들은 다음 표와 같다.

[표] Topology 어플리케이션에서 처리하는 event

```
void
Topology::configure(const Configurations*)
{
        register_handler<Link_event> (boost::bind(&Topology::handle_link_event, this, _1));
        register_handler<Datapath_join_event> (boost::bind(&Topology::handle_datapath_join, this, _1));
        register_handler<Datapath_leave_event> (boost::bind(&Topology::handle_Datapath_leave, this, _1));
        register_handler<Port_status_event> (boost::bind(&Topology::handle_port_status, this, _1));
}
```

Topology 어플리케이션에서는 기본적인 이벤트 외에 “Datapath_leave_event”와 “Link_event”와 를 처리한다. 이 때, “Link_event”는 의존관계에 있는 ‘discovery' 어플리케이션으로부터 생성되고 “Datapath_leave_event”는 OF switch와 controller간 secure channel이 close될 때(Conn::close()) 생성된다. 참고로 configure()에서 사용하고 있는 register_handler()는 본 어플리케이션에서 작성한 이벤트 핸들러를 등록하기 위해 사용되었다.

▷ Topology 어플리케이션 내에 사용되는 구조체들

- ✓ LinkPorts : 양 스위치간 연결되어 있는 포트 번호
  - struct LinkPorts{uint16_t src; uint16_t dst;}
- ✓ LinkSet
  - std::list<LinkPorts>
- ✓ PortVector
  - std::vector<Port>
- ✓ PortMap : 각 port의 연결성 유무
  - hash_map<uint16_t, std::pair<uint16_t, uint32_t>
- ✓ DatapathLinkMap : datapath 연결 정보
  - hash_map<datapathid, LinkSet>
- ✓ DpInfo : datapath 구성 정보
  - struct DpInfo{PortVector ports; PortMap internal; DatapathLinkMap outlinks; bool active;}
- ✓ NetworkLinkMap : datapathid와 해당 datapathid의 구성정보를 매핑, 특히 Topology 어플리케이션에서 전역변수로 사용되는 topology의 자료형
  - hash_map<datapathid, DpInfo>

▷ Topology 어플리케이션의 이벤트 처리 루틴

a. Disposition Topology::handle_datapath_join(const Event & e)

datapath가 조인하게 되면(새 OF switch의 등장), topology 구조체에 해당 datapathid와 datapath정보(DpInfo)의 쌍으로 삽입된다. 구현된 내용은 다음과 같다.

[표] Topology 어플리케이션의 Datapath_join 이벤트 구현 내용

```
Disposition
Topology::handle_datapath_join(const Event & e) {
        /* ① 이벤트 구조체 캐스팅 */
        const Datapath_join_event& dj = assert_cast<const Datapath_join_event&> (e);
        /* ② 현재 topology 구조체에 해당 datapathid가 존재하는 지 확인 */
        NetworkLinkMap::iterator nlm_iter = topology.find(dj.datapath_id);
        /* ③ 해당 datapathid가 존재하지 않는다면 topology 구조체에 insert */
        nlm_iter = topology.insert(std::make_pair(dj.datapath_id, DpInfo())).first;
        /* ④ datapath 정보(DpInfo) 설정, DpInfo 구조체 내의 active와 ports 정보 */
        nlm_iter->second.active = true;
        nlm_iter->second.ports=dj.ports;
}
```

구현내용에서 보는 것처럼, Datapath_join 이벤트가 발생되면 hash_map 형태를 갖는 nlm_iter가 다음과 같이 구성되어 topology 구조체에 삽입된다.

[표] 삽입되는 nlm_iter의 내용

datapathid	active = true; ports = dj.ports; internal=?; outlinks=?;

b. Disposition Topology::handle_datapath_leave(const Event & e)

Datapath_leave 이벤트를 받으면 Topology 어플리케이션에서는 'datapathid'로 검색된 topology의 하나의 요소를 삭제한다. Datapath_leave 이벤트에 대한 Topology 어플리케이션의 처리 루틴은 다음 표와 같이 구현되어 있다.

[표] Topology 어플리케이션의 Datapath_leave 이벤트 구현 내용

```
Disposition
Topology::handle_datapath_leave(const Event & e){
        /* ① 이벤트 구조체 캐스팅 */
        const Datapath_join_event& dj = assert_cast<const Datapath_join_event&> (e);
        /* ② topology 구조체에서 해당 datapathid 찾기 */
        NetworkLinkMap::iterator nlm_iter =topology.find(dl.datapath_id);
        /* ③ topology 구조체 내에 해당 datapathid가 존재하지 않으면 에러발생,
        존재하면 다음과 같이 topology 구조체에서 해당 엔트리 삭제 */
        if(!(nlm_iter->second.internal.empty()&& nlm_iter->second.outlinks.empty()))
        { /*** 의문사항 : internal과 outlinks가 존재할 때에는 해당 datapathid 엔트리를 topology 구조체에서 삭제안하나? ***/
                nlm_iter->second.active = false;
                nml_iter->second.ports.clear();
        } else{
                topology.erase(nlm_iter);
        }
}
```

c. Disposition Topology::handle_port_status(const Event & e)

Port_status 이벤트를 Topology 어플리케이션이 받으면, Port_status 메시지 내의 “reason" 필드에 따라 Topology 구성의 변화를 반영한다. 구현 내용은 다음과 같다.

[표] Topology 어플리케이션의 Port_status 이벤트 구현 내용

```
Disposition
Topology::handle_port_status(const Event& e)
{
        /* ① 이벤트 구조체 캐스팅 */
        const Port_status_event& ps = assert_cast<const Port_status_event&>(e);
        /* ② "reason" 필드에 따른 topology 어플리케이션의 행위 정의*/
        if(ps.reason == OFPPR_DELETE){
                delete_port(ps.datapath_id, ps.port);
        }else{
                add_port(ps.datapath_id, ps.port, ps.reason != OFPPR_ADD)
        }
}
```

d.Disposition Topology:::handle_link_event(const Event & e)

Link 연결정보인 Link 이벤트를 받으면, Topology 어플리케이션은 topology(le.dpsrc)->second->outlinks, topology(le.dpdst)->second->internal 내용을 갱신하여, 해당 datapathid에 대한 연결정보를 관리한다.

[표] Topology 어플리케이션의 link 이벤트 구현 내용

```
Disposition
Topology::handle_link_event(const Event & e)
{
        const Link_event& le = assert_cast<const Link_event&>(e);
        if(le.action == Link_event::ADD)
        {
                add_link(le);
        }else if(le.action == Link_event::REMOVE){
                remove_link(le);
        }else{
                lg.err("unknown link action %u", le.action);
        }
        return CONTINUE;
}
```

(3) Routing

1) Routing 어플리케이션 개요

Routing 어플리케이션은 shortest path 알고리즘을 이용하여 유입된 플로우의 경로를 계산하여 OpenFlow Switch에 플로우 엔트리를 추가시켜 주는 역할을 한다. 이 어플리케이션은 routing_module과 sprouting 컴포넌트로 구성되었으며, 기타 다른 컴포넌트들도 사용되었다.

[표] meta.xml 내용의 일부

```
<component>
  <name>routing</name>
  <library>sprouting</library>
  <dependency>
    <name>routing_module</name>
  </dependency>
  <dependency>
    <name>authenticator</name>
  </dependency>
  <dependency>
    <name>configuration</name>
  </dependency>
</component>

<component>
  <name>routing_module</name>
  <library>routing_module</library>
  <dependency>
    <name>topology</name>
  </dependency>
  <dependency>
    <name>nat_enforcer</name>
  </dependency>
</component>
```

2) routing_module의 기능 분석

routing_module은 최단 경로를 처리하고, 플로우 엔트리들을 설정하기 위해 호출되는 유틸리티 컴포넌트이다. 본 어플리케이션은 아래표에서처럼 configure()에 작성되어 있는 “Link_event”를 처리하여 링크의 변화에 따라 최단 경로를 계속 갱신해 놓는 역할을 하고, 또 타 컴포넌트의 요청에 따라 가지고 있는 route를 리턴하거나 유효성을 체크해준다.

[표] routing_module에서 처리하는 이벤트

```
void
Routing_module::configure(const container::Configuration*)
{
    resolve(topology);
    resolve(nat);
    register_handler<Link_event>  (boost::bind(&Routing_module::handle_link_change, this, _1));
}
```

▷ 주요 구조체들

✓ 기본 구조체

구조체명	구성요소	주석
struct Link	datapathid dst uint16_t outport uint16_t inport	destination dp of link starting at current dp outport of current dp link connected to inport of destination dp link connected to
struct RouteId	datapathid src datapathid dst	Source dp of a route Destination dp of a route
struct Route	RouteId id std::list<Link> path	Start/End datapath links connecting datapaths

✓ 주요 구조체

- typedef boost::shared_ptr<Route> RoutePtr
- typedef std::list<RoutePtr> RouteList
- typedef hash_set<RoutePtr, routehash, routeq> RouteSet
- typedef hash_map<RouteId, RoutePtr, ridhash, rideq> RouteMap;
- RouteMap shortest;

▷ routing 어플리케이션의 이벤트 처리 루틴

✓ handle_link_change: link 상태의 변화 시에 최단 경로를 update 함. 즉, Link_event::REMOVE시와 Link_event::ADD에 따라 All-pairs shortest path 알고리즘을 구현한 내용을 수행 함. (eg. cleanup, fixup, add 등등)

그 외에, All-pairs shortest path 알고리즘("A New Approach to Dynamic All Pairs Shortest Paths" by C. Demetrescu)을 위한 여러 가지 function들이 구현되어 있다.

▷ routing 어플리케이션의 API (타 컴포넌트로부터의 요청받을 수 있는 함수들)

✓ get_route() : 주어진 RouteID 'id'를 갖고, 두 datapaths 사이의 경로를 'route'로 설정. 둘 사이에 경로가 존재하면 'true', 아니면 'false'. 대부분의 경우, 경로는 check_route로 보내져서 경로의 유효성을 검사함.(들어왔던 같은 포트로 빠져나가는지?)

✓ setup_route() : 'route', 'inport', 'outport'에 따라 'flow'를 라우팅 하는데 필요한 스위치 엔트리들을 설정함. 엔트리들을 'flow_timeout'후에 유효하지 않게 됨. 'actions'이 non-empty이면, flow를 라우팅하는데 사용되는 OFPAT_OUTPUT action외에, 각 datapath에 적용할 OpenFlow actions들을 명시해야 함. 즉, action[i]는 i번째 datapath에 위치하는 actions의 집합임. 또한 dp마다 (즉, 노드마다) send_openflow_command(dp, &ofm->header, false)를 수행하여 ofp_flow_mod를 통해 Flow setup 및 teardown을 수행함.

✓setup_flow() : setup_route와 같은 일을 수행하는 데, 처리 대상이 setup_route는 route전체인 반면 본 함수는 dp 하나임. set_openflow, check_openflow, set_action등을 순차적으로 수행하고, send_openflow_command(dp, &ofm->header, false)도 수행함.

✓check_route() : 주어진 'route', 'inport', 'outport'에 대해 들어왔던 같은 포트로 다시 빠져나가는지를 검사함. (inport == outport)

▷ routing 어플리케이션의 OF message 구성 관련 함수들

✓ init_openflow() : ofp_flow_mod *ofm을 초기화 시킴. (eg. ofm->header.type = OFPT_FLOW_MOD, ofm->command = htons(OFPFC_ADD) 등등)

✓check_openflow() : openflow 메시지(?)의 유효성 검사함.

✓set_openflow() : flow를 받아들여 ofp_match& match를 초기화시킴. setup_flow와 setup_route에서 호출됨.

✓set_openflow_actions() : ofm->actions을 파라미터로 하여 set_action을 호출함.

✓modify_match() : const Buffer& actions을 받아들여 ofp_match& match를 action type에 따라 초기화시킴. 왜 필요할까 ??? setup_route에서 호출됨.

✓set_action() : NAT 적용이 되었는지에 따라, nx_action_snat& snat 또는 ofp_action_output& output을 초기화시킴. 함수 이름에서 기능을 유추하기 어려움.

✓send_packet() : 결국, Packet_out을 하기 위한 함수. Topology쪽과도 연관 있는 듯. check_openflow, set_action등을 순차적으로 수행하고, send_openflow_packet(dp, bid, ofm->actions, actions_len, inport, false)도 수행함.

3) Sprouting

SamPle Routing 컴포넌트로써, Flow_in_events를 받으면 routing_module의 도움을 받아 종단간 최단경로를 설정한다. 경로가 존재하지 않을 경우, 패킷을 브로드캐스트하고, 경로가 존재하지만 패킷이 같은 포트로 빠져 나갈 때는 해당 플로우가 drop되는 형태로 전송 경로를 결정한다. 본 어플리케이션의 경우, 아래 configure()에서 보는 것처럼 Bootstrap_complete_event와 Flow_in_event에 대하 처리 루틴을 가지고 있다.

```
void
SPRouting::configure(const container::Configuration*)
{
    resolve(routing);
    resolve(tstorage);
    register_handler<Bootstrap_complete_event> (boost::bind(&SPRouting::handle_bootstrap, this, _1));
    register_handler<Flow_in_event> (boost::bind(&SPRouting::handle_flow_in, this, _1));
}
```

✓ configure() : resolve 함수를 이용하여 Routing_module(routing)에 대한 핸들을 얻고, Bootstrap_complete_event와 Flow_in_event에 대한 이벤트 핸들러인 handle_bootstrap, handle_flow_in을 등록함.

▷ sprouting 어플리케이션의 이벤트 처리 루틴

✓ handle_flow_in() : Flow_in_event에 대해 route_flow를 호출. 단, Flow_in 이벤트는 packet_in 이벤트 시에, authenticator 컴포넌트에 의해 생성됨.

✓ route_flow() : set_route 호출 후, routing->setup_route 호출. route상에 해당 플로우의 datapath_id가 존재하면, on_route를 true로 설정. on_route가 true일 경우, routing->send_packet을 통해 packet_out 메시지 처리(OFPP_TABLE). on_route가 false일 경우, routing->send_packet을 통해 flooding 시킴(OFPP_FLOOD).

✓ set_route() : inport, outport를 설정하고, routing->get_route을 통해 경로를 얻고, routing->check_route를 통해 경로의 유효성을 검사함. 그러나, break에 걸려, while 밖으로 나온 경우 fi.routed_to를 Flow_in_event::BROADCASTED로 설정함. 추가적으로, fi.flow.dl_dst.is_broadcast()에 따라 routing->setup_flow 또는 routing->send_packet 처리

▷ sprouting 을 Flow_util 컴포넌트와 함께 이용하는 루틴

✓ install() : resolve 함수를 이용하여 Flow_util(flow_util)에 대한 핸들을 얻고, register_handler에서와 유사하게 필요한 핸들러(?)들을 등록함. 네 가지 경우(rewrite_headers, http_proxy_redirect, http_proxy_undo_redirect, allow_no_nat)가 존재하는데, 각 경우가 어떠한 것을 의미하는지 정확히 파악 안됨. 기본적으로 allow_no_nat쪽으로 가지 않을까??

✓ validate_args(): args에 대한 파리미터의 유효성 검사하는 함수

✓ validate_redirect_args() : args를 입력받아 "SRC", "NO_NAT"를 삽입하고, 다시 validate_args를 호출

✓ generate_rewrite() : args를 받아들여, validate_args를 검사함. validate_args를 통해 얻어진 eth, ip, tport 등을 이용하여 파라미터를 조작(?)하고, boost::bind를 통해 &SPRouting::rewrite_and_route와 연결함.

✓ rewrite_and_route() : flow_in을 파라미터로 route_flow 호출. route_no_nat에서 호출됨.

✓ route_no_nat() : rewrite_and_route 호출

5.결론

우리가 가지고 있는 OpenFlow 인프라에서는 IP주소뿐만이 아니라 다양한 메트릭을 이용하여 패킷을 라우팅 할 수도 있고, 특정 시점마다 패킷 전송 경로를 변경할 수도 있다. 그러나 이러한 유연한 네트워크 운영기술을 이용하기 위해서는 네트워크 연구자 또는 응용 연구자 스스로 응용의 특성에 맞는 형태로의 네트워크 운영기술 개발이 따라줘야 한다.

본 문서는 현재 스탠포드 대학에서 제공하고 있는 NOX에 대한 구현 코드 분석 내용을 기술하였다. NOX에서는 각각의 필요기능을 컴포넌트 단위로 구현하였고, OF스위치가 OF 메시지를 통해 컴포넌트로 전달하고자 하는 데이터 또는 컴포넌트간 전달하고자 하는 데이터를 '이벤트'라는 형태의 구조체를 만들어서 상당히 복잡할 수 있는 NOX의 구조를 단순하게 파악할 수 있도록 구현하였다. 즉, 각각의 컴포넌트는 네트워킹을 위해 필요한 단위 정보를 생성/갱신/소멸하는 역할을 수행하고 그 수행에 필요한 정보들을 이벤트를 통해 전달한다.

미래연구망으로서 KREONET이 OpenFlow가 추구하는 네트워크 운영의 개방성을 잘 적용시킬 수 있는 모델이 되면 아이폰이 등장하면서 'IT 생태계'라는 말이 나온 것처럼 '네트워크 생태계' 또는 그에 준하는 이름으로 진화할 수 있을 거라고 생각한다.