基于OpenFlow的网络故障诊断系统

基于OpenFlow的网络故障诊断系统借助SDN网络管控分离、逻辑集中的架构特点，在控制器端实现动态灵活的、精细化的网络测量，并将测量的结果用于网络故障的诊断。同时，实现了网络物理故障的自动检测和恢复机制，增强了网络的自我保障性。

控制平面

数据平面

OpenFlow

网络测量模块

网络拓扑测量

链路参数测量

故障感知

故障诊断模块

物理故障诊断

性能故障诊断

故障恢复

图1 系统结构图

该系统共包含两个部分：参数测量部分和故障诊断部分，如图1所示。参数测量部分主要负责网络拓扑感知、链路性能参数测量和故障感知。因为SDN网络采用逻辑集中的控制器进行网络的控制，因此，这些功能均在控制器端实现，拓扑感知的主要功能是维护全网的拓扑视图，为其他功能提供拓扑数据支持，链路性能参数测量的主要功能是测量网络中任意链路的丢包、带宽和延迟情况。故障感知的功能是监测网络中的链路故障，在链路发生故障时及时的通知故障诊断模块进行处理。故障诊断部分的功能包括物理故障诊断，性能故障诊断以及故障恢复。本系统的实现共包含四个模块：网络资源感知模块、网络监控模块、网络路由模块和网络延迟测量模块。各模块的设计文档如下文所示。

## 1、网络资源感知模块

网络资源感知模块用于感知网络资源的实时变化，包括拓扑信息以及主机信息的变化。任何网络应用，可达性都是最基本的要求。SDN网络的集中控制，使得控制器可以根据全局的信息作出最佳决策而无需在交换节点上采用分布式的路由算法。所以感知网络资源是SDN应用最基础的一项服务。

### 1.1网络拓扑资源获取

网络拓扑发现是网络故障诊断与恢复的基础，在传统网络中，网络中的链路的发现由各个网络设备自动进行。而SDN网络将网络设备的控制层面剥离出来，使得网络设备变成一个单纯按照流表项进行转发的设备，所有的控制功能都统一集中到逻辑集中的控制器。因此，在SDN网络中，网络链路的发现有控制器完成。SDN控制器通过链路发现协议（Link Layer Discovery Protocol,LLDP）、packet\_in消息和packet\_out消息进行网络拓扑的发现。

链路层发现协议是一个厂商无关的二层协议，它可以被网络设备向局域网中的其他设备通告自己的设备标志和功能，包括设备的管理地址、设备标识、接口标识等信息。需要注意的是链路层发现协议的功能是用来进行信息通告，并不能用于远程设备的配置，它只提供了关于网络拓扑及管理配置的信息，这些信息可以被用来进行网络的配置和管理。LLDP的帧格式如图2所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DA | SA | LLDP Ethertype | Data+pad |  |
| LLDP\_multicast address | Mac address | 88-CC | LLDPDU | FCS |

图2 Ethernet II格式封装的LLDP帧

其中各个字段的含义如下：

DA：目的 MAC地址，为固定的组播 MAC地址 0x0180-C200-000E。

SA：源 MAC地址，为端口 MAC地址或设备MAC地址（如有端口地址则用端口MAC地址，否则用设备MAC地址）。

Type：帧类型，为 0x88CC。

Data：数据，为 LLDPDU。

FCS：帧检验序列。

在进行网络拓扑发现时，往往将链路源端的设备信息封装在链路发现协议数据报文的数据字段，即LLDPPDU内。LLDPDU采用了TLV的格式，即type+lenght+value的格式，其组成结构如图3所示。type表示TLV的类型，length是以字节为单位的TLV的长度，value是该TLV的值。其中Chassis ID TLV，Port ID TLV Time To Live TLV以及End Of LLDPDU TLV是强制的，必须包含的部分，除此之外在TLV Time To Live TLV和End Of LLDPDU TLV之间可以包含0个到多个可选的其它TLV。

图3 LLDPDU格式

在SDN网络中，采用主动测量的方式进行网络链路拓扑发现过程，具体过程如图4所示。



图4 SDN网络中的拓扑发现过程

控制器在和OpenFlow交换机进行握手时，交换机向控制器发送自己的状态信息，包括数据平面ID、端口状态等，但这些信息并不包括网络拓扑信息.为了获取OpenFlow的网络拓扑信息，通过链路发现协议（Link Layer Discovery Protocol,LLDP）进行拓扑发现。控制器将sw1的数据平面ID（datapath\_id）以及端口号（port2） 封装到LLDP数据包中，然后通过packet\_out消息指示sw1将该LLDP数据包通过端口2发送出去，当该数据包到达sw2的port1端口，会触发sw2发送packet\_in消息给控制器，该packet\_in消息中包含了由sw1的port2发送过来的LLDP数据包，控制器从收到的packet\_in消息中可以解析得到该packet\_in消息是由sw2的port1触发的，同时从packet\_in的LLDP数据包中可以解析出该数据包是从sw1的port2发送过来的，此时控制器就知道sw1的2端口和sw2的1端口是直连的。同理，控制器也可以指示sw2从其1端口发送LLDP数据包，这样控制器就知道sw2的1端口和sw1的2端口是直连的.由此，控制器完全获取了一条链路（（sw1，port2），（sw2，port1）的连接信息.推广到整个网络，控制器可以指示从所有交换机的所有端口发送LLDP数据包，从而，获取全网的拓扑信息。

### 1.2 终端主机发现

通过1.1节所述方法可以进行网络拓扑的发现，但是，通过这种方式发现的仅仅是网络中的交换机之间的连接情况，该拓扑图中并不包含终端主机的信息。然而，在我们进行路由规划是，如果能够知道网络中终端主机的具体位置，就可以进行更加快速、高效的路径规划。

因为终端主机并不能像网络交换机那样转发控制器下发的报文，也不能够出发packet\_in消息，因此，在SDN网络中终端主机往往是处于沉默状态，如果终端主机在网络中不进行任何通信，不产生任何流量，控制器是不能发现终端主机的。但是当终端主机进行通信时，由于接入交换机上并没有针对沉默终端主机的流表项，当沉默主机产生的初始通信数据包到达接入交换机时就会触发packet\_in事件，通过分析packet\_in事件，就可以获取到终端主机的详细位置，然后控制器可以利用这些信息进行网络拓扑的更新和完善。

### 1.3网络端口的分类

当进行网络拓扑的发现之后，控制器已经掌握的网络设备上的所有端口信息，如上文所述，SDN网络是管控分离、逻辑集中的架构，所有的控制功能均集中到控制器端完成。ARP广播风暴是在网络中普遍存在的问题，在传统网络中，通常使用生成树协议（Spanning Tree Protocol，STP）协议防止网络广播风暴的产生，但是该协议具有如下缺点：

1. 由于整个交换网络只有一棵生成树，在网络规模比较大的时候会导致较长的收敛时间，拓扑改变的影响面也较大。
2. 在网络结构对称的情况下，单生成树也没什么大碍。但是，在网络结构不对称的时候，单生成树就会影响网络的连通性。
3. 当链路被阻塞后将不承载任何流量，造成了带宽的极大浪费，这在环行城域网的情况下比较明显。

由于SDN网络中的控制器具有全局的网络视图，我们将网络中的所有端口划分成两类，内向端口和外向端口。当网络设备加入网络时，会向控制器汇报自身的状态信息，其中包含其端口信息，设从中获取的所有端口集合为*All\_port*，当通过LLDP协议进行网络拓扑发现时发现的链路端口定义为网络中的内向接口，设内向接口的集合为*interior\_port*。则网络中的外向接口集合为*access\_port=All\_port-interior\_port*。通过将网络中的所有接口分为内向接口和外向接口，当进行网络广播时，可以直接将网络广播报文通过外向接口发送到外向主机，而不用在网络内部进行传输，从而可以有效地防止网络广播风暴的产生。

## 2、网络监控模块

网络监测模块的主要功能是在网络资源感知模块获取的网络拓扑信息基础上，利用OpenFlow协议，采用被动测量的方式获取网络链路的带宽和丢包情况，从而为网络性能故障诊断提供依据。

### 2.1网络链路带宽测量

带宽的测量采用被动的方式进行，OpenFlow交换机上针对每条流、每个端口、每个队列维护计数器。带宽的测量可以利用交换机上维护的端口信息进行。由controller向链路两端的交换机发送ofp\_port\_stats\_request消息，询问构成链路的端口上的统计信息，交换机收到查询请求后返回ofp\_port\_stats消息，如图5所示。该消息包含了对应端口的全部统计信息。而带宽的计算需要使用*rx\_bytes*和*tx\_bytes*统计信息。

**struct** ofp\_port\_stats{

**uint16\_t**  port\_no;

**uint8\_t**  *pad*[6];

**uint64\_t**  *rx\_packets; /\*接收数据包个数\*/*

**uint64\_t** *tx\_packets*; */\*发送数据包个数\*/*

**uint64\_t** *rx\_bytes*; */\*接收数据包字节数\*/*

**uint64\_t** *tx\_bytes*; */\*发送数据包字节数\*/*

**uint64\_t** *rx\_dropped;*

**uint64\_t** *tx\_dropped;*

**uint64\_t** *rx\_errors*;

**uint64\_t** *tx\_errors;*

**uint64\_t** *rx\_frame\_err;*

**uint64\_t** *rx\_over\_err;*

**uint64\_t**  *rx\_crc\_err;*

**uint64\_t**  *collisions;*

};

图5 ofp\_port\_stats结构体

设时刻收集到的（sw1，port2）的*rx\_bytes*为，时刻收集到的（sw1，port2）的*rx\_bytes*为，则（sw1，port2）端口发送流量为：，同理，可以得到（sw2,port1）的接收流量为：.则（sw1,sw2）链路在（sw1，port2），（sw2,port1））方向的流量为。

### 2.2丢包率测量

链路丢包率测量与链路带宽测量类似，也是采取被动测量的方式，通过控制器发送ofp\_port\_stats\_request消息，获取交换机返回的ofp\_port\_stats消息，从中读取指定端口的发送数据包数目以及接受数据包数目，然后，利用读取到的带参数进行链路丢包率的计算。如图4所示，分别读取sw1交换机port2端口的*tx\_packets*和以及sw2交换机port1端口的*rx\_packets*.则链路（（sw1，port2），（sw2,port1））在（sw1,sw2）方向的丢包率为：(*tx\_packets-rx\_packets)/ tx\_packets*。

### 2.3拥塞度测量

链路拥塞度的测量主要利用链路端口的发送数据包统计参数和接收数据包统计参数计算，如图4所示，链路((sw1,port2）（sw2,port1）)的拥塞度可利用带宽测量模块计算的端口流量参数计算.设端口(sw1,port2)的发送流量为，端口（sw2,port1）的接收流量为，则链路的拥塞度为：，在链路没有出现拥塞的情况下，的值为1，链路出现拥塞，的值会大于1，并且会随着链路拥塞度的增加而变大。

## 3 网络路由模块

网络路由模块的主要功能是对网络中的数据流进行路由，并且监测网络链路的状态，当网络链路发生故障时，及时的将受故障链路影响的数据流切换到备用链路，保障网络业务的服务质量，同时，增强网络的自主保障性。故障检测及恢复流程如图6所示。



图6 故障检测及恢复流程

### 3.1故障检测

该模块主要用于检测链路的故障，检测的方法有两种，一种是当交换机的端口发生故障时，会触发OFPT\_PORT\_STATUS消息，该消息中包含reason字段，通过解析OFPT\_PORT\_STATUS[8]中的reason字段可以获得链路变化的具体原因。另一种方法是在拓扑发现时，会对发现的网络链路加上时间戳，并在下次发现时更新该时间戳，并计算当前时间和时间戳的差值，如果该差值超过了设定的阈值，则说明该条链路出现故障。

### 3.2故障链路提取

但链路发生故障时，会触发控制器中的EventLinkDelete，通过该事件可以提取出故障链路的源交换机dpid和宿交换机dpid，从而为清除失效的流表项提供信息。

### 3.3失效流表项清理

该部分是故障快速恢复的关键，因为失效的流表项会一直将数据流按照错误的路径进行转发，直到流表项超时失效，故障才能恢复，但这样故障恢复的时间较长，所以，需要在链路故障时，清除失效的流表项。问题的关键如何查找失效的流表项，这需要在控制端维护所有留的路径信息，当故障发生时，找出受故障链路影响的路径，对于该路径上来网络流进行流表项清理、重新规划路径，以此快速的恢复故障链路。

### 3.4路由重规划

路由重规划是在清理完失效刘表项之后，对受影响的数据流进行网络路径的重规划。路径的规划借助python下的复杂网络编程包networkx来完成，将网络网络抽象为图，然后，利用networkx中的路径计算函数计算网络路径。具体过程如图7所示。



图7 最短路由流程图

首先，查询主机表，若查找成功，则查询主机位置表，之后直接由控制器将ARP数据包发送给对应的端口，此时控制器并不做ARP的代理。当目标主机回复ARP时，将数据包直接发送到源主机的接入端口。从而完成了ARP的学习过程。由于此时已经掌握了主机的接入信息以及网络信息，当ICMP或其他数据包触发packet\_in事件时，则可根据源目两个IP查询到接入交换机，再依据拓扑信息，计算最短路径，从而完成最短路由。

在网络初始化时，控制器并没有办法发现沉默的主机，原因在于我们没有进行DHCP分配，导致控制器没有掌握主机的IP/MAC信息。所以第一步我们需要处理的数据包是ARP。处理流程具体如图8所示：



图8 ARP处理流程图

首先，判断APR数据包的类型，如果是ARP应答包，则目的主机的位置必然是已知的，只需要查询目的主机的位置，然后将ARP应答包发送到目的主机即可。如果是ARP请求包，则查找目的主机的位置是否已经，如果已知，则将请求包发送到目的主机，否则通过所有的外部端口进行泛洪，将所有的端口分为内外端口和外部端口，与终端主机相连的端口为外部端口，在泛洪时，只通过外部端口进行泛洪，可以防止广播风暴的产生。

## 4 网络延迟测量模块

网络测量模块的功能是利用SDN网络逻辑集中的系统架构，控制器采用主动测量的方式获取网络中任意链路的延迟参数，而且，该方法不需要定制网络硬件设备，无需对现有的OpenFlow交换机进行任何改造升级，具有灵活可控，适用范围广，成本低廉的优点。

### 4.1基于OpenFlow的链路延迟测量架构

基于OpenFlow的链路延迟测量主要利用SDN网路管控分离，逻辑集中的架构特征。该方法的系统架构图如图9所示。在控制器端主要包括四个模块：规则下发模块，探测包生成模块，探测包监听模块和延迟计算模块。其中规则下发模块负责在测量路径上事先下发流表项规则，该规则负责将探测包按照测量路径转发，如图9所示，需要测量的路径为S1→S2→S3，为了使探测包沿着测量路径转发，需要在S1，S2和S3上提前下发转发规则，使得探测数据包按照S1→S2→S3的路径进行转发。探测包生成模块主要负责延迟探测报的生成，探测包可以是任意格式的数据包，只要与规则下发模块下发规则的匹配项相对应即可，控制器构造探测包之后可以通过packet\_out数据包将探测包发送到测量路径上的第一个交换机上，使得探测数据包进入网络，按照流表项规则进行转发。探测包监听模块负责在控制器端检测探测包，当探测包传送到测量路径的最后一个交换机时，该探测包会被上传到控制器，因此，需要在控制器上监测探测包。延迟测量模块利用探测包转发过程中的时间戳来计算测量路径上的延迟。

图9 基于OpenFlow的延迟测量示意图

### 4.2基于OpenFlow的链路延迟测量

SDN网络具有管控分离和逻辑集中地特性，因此，在SDN网络中可以进行更加灵活的延迟测量，本文给出链路延迟的定义是：数据包从到达链路第一个交换机到离开链路最后一个交换机所经历的时间。如图10所示，数据包到达交换机的时间为*t2，t4，t6*，离开交换的时间为*t3，t5，t7*，则S1→S2→S3链路的延迟为*t7- t3*。



图10 通过控制协议时间戳计算链路延迟

在传统网络中，如果想要测量网络中任意路径的延迟，就需要测量交换机传输数据包的时间，这往往会付出巨大的代价[1]，或者是需要修改交换机硬件[2]。而在SDN网络中，网络的控制层面变得更加的开放，灵活。本文提出的链路延迟测量方法通过OpenFlow中的控制消息来进行链路延迟的测量，包括packet\_out消息，packet\_in消息，echo\_request消息和echo\_reply消息。该方法不需要对网络设备进行任何修改，而且，借助SDN网络管控分离、集中控制的特性，能够测量网络任意路径的延迟。因此，该方法具有低开销、灵活可控的特点。

为了低开销的测量网络链路上的延迟，需要从控制器端向网络链路注入探测包，并记录相应的时间戳，这样可以不用对底层设备做修改，减少网络测量的开销。首先，在测试链路的交换机上预先下发“导航”流表项，使得探测数据包能够沿着测试链路进行转发，测试链路上的最后一个交换机应该将探测包上传至控制器，同时“导航”流表项还需要满足以下2点需求。

1. 预先下发的“导航”流表项不能对正常的网络流量产生干扰，这可以通过设定特殊的匹配项来避免，例如为“导航”流表项设定特殊的VLAN标签，指定特殊的TCP或者UDP端口号等。
2. “导航”流表项需要设定合适的硬超时时间，当测量任务完成后，“导航”流表项必须自动删除失效，以解释宝贵的TCAM (Ternary Content Addressable Memory，TCAM)资源。

在完成“导航”流表项下发之后，需要在控制器端生成延迟探测包，将其封装在packet\_out数据包中，注入到测试链路的起始交换机，指定packet\_out数据包的action为从链路起始端口转发，并记录发送探测报文的时间，即图10所示的*t1*。在探测报文发送到网络之后，会按照预先下发的“导航”流表项进行转发，在探测报文到达探测链路的最后一个交换机时，会被上传到控制器，控制器监测到包含探测包的packet\_in数据包时，会再次记录时间戳，即图10所示的*t8*。

通过packet\_in和packet\_out的时间戳之差，可以得到查找延迟、转发延迟和控制延迟之和，但是数据包在网络链路是经历的延迟不包含控制延迟。所以，为了精确的计算网络链路的延迟，必须计算SDN网络中的控制延迟。在SDN网络中，可以通过echo reques消息确认控制器和交换机的连通性，当交换机收到控制的echo request消息后，会立即回复echo reply消息。因此，可通过echo request消息和echo reply消息测量控制延迟，控制器通过记录发送echo request消息和收到echo reply消息的时间戳，并计算两者之差即可得到控制延迟，即图10中的*d1，d2*。

同时，由于控制器在生成和发送探测报文以及计算链路延迟时会消耗一定的时间，因此在利用以上方法计算链路延迟时需要添加一个矫正量△t。综上所述，基于OpenFlow的链路延迟计算公式为：*delay*= *（t8- t1）- （d1+ d2）*+*△t*。

## 5程序清单（开发语言Python）

### 5.1网络资源感知模块

# conding=utf-8

import logging

import struct

from operator import attrgetter

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from ryu.controller.handler import MAIN\_DISPATCHER, DEAD\_DISPATCHER

from ryu.controller.handler import CONFIG\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.ofproto import ofproto\_v1\_3

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet

from ryu.lib.packet import ipv4

from ryu.lib.packet import arp

from ryu.lib import hub

from ryu.topology import event, switches

from ryu.topology.api import get\_switch, get\_link

SLEEP\_PERIOD = 10

IS\_UPDATE = True

class Network\_Aware(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

\_NAME = 'network\_aware'

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(Network\_Aware, self).\_\_init\_\_(\*args, \*\*kwargs)

self.name = "Network\_Aware"

self.topology\_api\_app = self

# links :(src\_dpid,dst\_dpid)->(src\_port,dst\_port)

self.link\_to\_port = {}

# {(sw,port) :[host1\_ip,host2\_ip,host3\_ip,host4\_ip]}

self.access\_table = {}

# ports

self.switch\_port\_table = {} # dpid->port\_num

# dpid->port\_num (outer ports)

self.access\_ports = {}

# dpid->port\_num(interior ports)

self.interior\_ports = {}

self.graph = {}

self.pre\_link\_to\_port = {}

self.pre\_graph = {}

self.pre\_access\_table = {}

self.discover\_thread = hub.spawn(self.\_discover)

# show topo ,and get topo again

def \_discover(self):

i = 0

while True:

self.show\_topology()

if i == 5:

self.get\_topology(None)

i = 0

hub.sleep(SLEEP\_PERIOD)

i = i + 1

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

datapath = ev.msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

msg = ev.msg

self.logger.info("switch:%s connected", datapath.id)

# install table-miss flow entry

match = parser.OFPMatch()

actions = [parser.OFPActionOutput(ofproto.OFPP\_CONTROLLER,

ofproto.OFPCML\_NO\_BUFFER)]

self.add\_flow(datapath, 0, match, actions)

def add\_flow(self, dp, p, match, actions, idle\_timeout=0, hard\_timeout=0):

ofproto = dp.ofproto

parser = dp.ofproto\_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS,

actions)]

mod = parser.OFPFlowMod(datapath=dp, priority=p,

idle\_timeout=idle\_timeout,

hard\_timeout=hard\_timeout,

match=match, instructions=inst)

dp.send\_msg(mod)

def get\_switches(self):

return self.switches

def get\_links(self):

return self.link\_to\_port

# get Adjacency matrix from link\_to\_port

def get\_graph(self, link\_list):

for src in self.switches:

for dst in self.switches:

self.graph.setdefault(src, {dst: float('inf')})

if src == dst:

self.graph[src][src] = 0

elif (src, dst) in link\_list:

self.graph[src][dst] = 1

else:

self.graph[src][dst] = float('inf')

return self.graph

for sw in switch\_list:

dpid = sw.dp.id

self.switch\_port\_table.setdefault(dpid, set())

self.interior\_ports.setdefault(dpid, set())

self.access\_ports.setdefault(dpid, set())

for p in sw.ports:

self.switch\_port\_table[dpid].add(p.port\_no)

def create\_port\_map(self, switch\_list):

for sw in switch\_list:

dpid = sw.dp.id

self.switch\_port\_table.setdefault(dpid, set())

self.interior\_ports.setdefault(dpid, set())

self.access\_ports.setdefault(dpid, set())

for p in sw.ports:

self.switch\_port\_table[dpid].add(p.port\_no)

# get links`srouce port to dst port from link\_list,

# link\_to\_port:(src\_dpid,dst\_dpid)->(src\_port,dst\_port)

def create\_interior\_links(self, link\_list):

#self.link\_to\_port={}

for link in link\_list:

src = link.src

dst = link.dst

self.link\_to\_port[

(src.dpid, dst.dpid)] = (src.port\_no, dst.port\_no)

# find the access ports and interiorior ports

if link.src.dpid in self.switches:

self.interior\_ports[link.src.dpid].add(link.src.port\_no)

if link.dst.dpid in self.switches:

self.interior\_ports[link.dst.dpid].add(link.dst.port\_no)

@set\_ev\_cls(event.EventLinkDelete, MAIN\_DISPATCHER)

def update\_link\_delete(self,ev):

link = ev.link

src=link.src

dst=link.dst

if (src.dpid,dst.dpid) in self.link\_to\_port.keys():

del self.link\_to\_port[(src.dpid,dst.dpid)]

#print self.link\_to\_port

#print ev

self.graph[src.dpid][dst.dpid]=float('inf')

#print 'pre graph'

#print self.pre\_graph

#print 'update graph'

#print self.graph

# get ports without link into access\_ports

def create\_access\_ports(self):

for sw in self.switch\_port\_table:

self.access\_ports[sw] = self.switch\_port\_table[

sw] - self.interior\_ports[sw]

events = [event.EventSwitchEnter,

event.EventSwitchLeave, event.EventPortAdd,

event.EventPortDelete, event.EventPortModify,

event.EventLinkAdd, event.EventLinkDelete]

@set\_ev\_cls(events)

def get\_topology(self, ev):

switch\_list = get\_switch(self.topology\_api\_app, None)

self.create\_port\_map(switch\_list)

self.switches = self.switch\_port\_table.keys()

links = get\_link(self.topology\_api\_app, None)

self.create\_interior\_links(links)

self.create\_access\_ports()

self.get\_graph(self.link\_to\_port.keys())

# self.show\_topology()

# show topo

def show\_topology(self):

switch\_num = len(self.graph)

if self.pre\_graph != self.graph or IS\_UPDATE:

#print self.graph

print "---------------------Topo Link---------------------"

print '%10s' % ("switch"),

for i in xrange(1, switch\_num + 1):

print '%10d' % i,

print ""

for i in self.graph.keys():

print '%10d' % i,

for j in self.graph[i].values():

print '%10.0f' % j,

print ""

self.pre\_graph = self.graph

# show link

if self.pre\_link\_to\_port != self.link\_to\_port or IS\_UPDATE:

print "---------------------Link Port---------------------"

print '%10s' % ("switch"),

for i in xrange(1, switch\_num + 1):

print '%10d' % i,

print ""

for i in xrange(1, switch\_num + 1):

print '%10d' % i,

for j in xrange(1, switch\_num + 1):

if (i, j) in self.link\_to\_port.keys():

print '%10s' % str(self.link\_to\_port[(i, j)]),

else:

print '%10s' % "No-link",

print ""

self.pre\_link\_to\_port = self.link\_to\_port

# each dp access host

# {(sw,port) :[host1\_ip,host2\_ip,host3\_ip,host4\_ip]}

if self.pre\_access\_table != self.access\_table and IS\_UPDATE:

print "----------------Access Host-------------------"

print '%10s' % ("switch"), '%12s' % "Host"

if not self.access\_table.keys():

print " NO found host"

else:

for tup in self.access\_table:

print '%10d: ' % tup[0], self.access\_table[tup]

self.pre\_access\_table = self.access\_table

### 5.2 网络监控模块

from \_\_future\_\_ import division

from operator import attrgetter

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from ryu.controller.handler import MAIN\_DISPATCHER, DEAD\_DISPATCHER

from ryu.controller.handler import CONFIG\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.ofproto import ofproto\_v1\_3

from ryu.lib import hub

from ryu.lib.packet import packet

from ryu.topology.api import get\_switch, get\_link

SLEEP\_PERIOD = 10

class Network\_Monitor(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

\_NAME = 'Network\_Monitor'

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(Network\_Monitor, self).\_\_init\_\_(\*args, \*\*kwargs)

self.datapaths = {}

self.port\_stats = {}

self.port\_speed = {}

self.flow\_stats = {}

self.flow\_speed = {}

# {"port":{dpid:{port:body,..},..},"flow":{dpid:body,..}

self.stats = {}

# {dpid:{port\_no:(config,state,cur),..},..}

self.port\_link = {}

self.link\_status = {}

#{srcswitch,switch port,speed,loss,congestion}

self.monitor\_thread = hub.spawn(self.\_monitor)

@set\_ev\_cls(ofp\_event.EventOFPStateChange,

[MAIN\_DISPATCHER, DEAD\_DISPATCHER])

def \_state\_change\_handler(self, ev):

datapath = ev.datapath

if ev.state == MAIN\_DISPATCHER:

if not datapath.id in self.datapaths:

self.logger.debug('register datapath: %016x', datapath.id)

self.datapaths[datapath.id] = datapath

elif ev.state == DEAD\_DISPATCHER:

if datapath.id in self.datapaths:

self.logger.debug('unregister datapath: %016x', datapath.id)

del self.datapaths[datapath.id]

def \_monitor(self):

while True:

self.stats['flow'] = {}

self.stats['port'] = {}

for dp in self.datapaths.values():

self.port\_link.setdefault(dp.id, {})

self.\_request\_stats(dp)

hub.sleep(SLEEP\_PERIOD)

if self.stats['flow'] or self.stats['port']:

self.show\_stat('flow', self.stats['flow'])

self.show\_stat('port', self.stats['port'])

hub.sleep(1)

def \_request\_stats(self, datapath):

self.logger.debug('send stats request: %016x', datapath.id)

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

req = parser.OFPFlowStatsRequest(datapath)

datapath.send\_msg(req)

req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP\_ANY)

datapath.send\_msg(req)

req = parser.OFPPortDescStatsRequest(datapath, 0)

datapath.send\_msg(req)

def \_save\_stats(self, dist, key, value, length):

if key not in dist:

dist[key] = []

dist[key].append(value)

if len(dist[key]) > length:

dist[key].pop(0)

def \_get\_speed(self, now, pre, period):

if period:

return (now - pre) / (period)

else:

return 0

def \_get\_time(self, sec, nsec):

return sec + nsec / (10 \*\* 9)

def \_get\_period(self, n\_sec, n\_nsec, p\_sec, p\_nsec):

return self.\_get\_time(n\_sec, n\_nsec) - self.\_get\_time(p\_sec, p\_nsec)

def \_get\_link\_status(self):

links\_list = get\_link(self.topology\_api\_app, None)

def show\_stat(self, type, bodys):

'''

type: 'port' 'flow'

bodys: port or flow `s information :{dpid:body}

'''

if(type == 'flow'):

print('datapath '' in-port ip-dst '

'out-port packets bytes flow-speed(B/s)')

print('---------------- '' -------- ----------------- '

'-------- -------- -------- -----------')

for dpid in bodys.keys():

for stat in sorted([flow for flow in bodys[dpid]

if flow.priority == 1],

key=lambda flow: (flow.match['in\_port'],

flow.match['ipv4\_dst'])):

print('%016x %8x %17s %8x %8d %8d %8.1f' % (

dpid,

stat.match['in\_port'], stat.match['ipv4\_dst'],

stat.instructions[0].actions[0].port,

stat.packet\_count, stat.byte\_count,

abs(self.flow\_speed[

(stat.match['in\_port'],

stat.match['ipv4\_dst'],

stat.instructions[0].actions[0].port)][-1])))

print '\n'

if(type == 'port'):

print('datapath port ''rx-pkts rx-bytes rx-error '

'tx-pkts tx-bytes tx-error port-speed-tx(B/s) port-speed-rx(B/s)'

' current-capacity(Kbps) '

'port-stat ')

print('---------------- -------- ''-------- -------- -------- '

'-------- -------- -------- '

'---------------- ---------------- ---------------- '

' ----------- ')

format = '%016x %8x %8d %8d %8d %8d %8d %8d %8.1f %8.1f %16d %16s '

for dpid in bodys.keys():

for stat in sorted(bodys[dpid], key=attrgetter('port\_no')):

if stat.port\_no != ofproto\_v1\_3.OFPP\_LOCAL:

print(format % (

dpid, stat.port\_no,

stat.rx\_packets, stat.rx\_bytes, stat.rx\_errors,

stat.tx\_packets, stat.tx\_bytes, stat.tx\_errors,

abs(self.port\_speed[(dpid, stat.port\_no)][-1][0]),

abs(self.port\_speed[(dpid, stat.port\_no)][-1][1]),

self.port\_link[dpid][stat.port\_no][2],

self.port\_link[dpid][stat.port\_no][0],

))

print '\n'

@set\_ev\_cls(ofp\_event.EventOFPFlowStatsReply, MAIN\_DISPATCHER)

def \_flow\_stats\_reply\_handler(self, ev):

body = ev.msg.body

self.stats['flow'][ev.msg.datapath.id] = body

for stat in sorted([flow for flow in body if flow.priority == 1],

key=lambda flow: (flow.match['in\_port'],

flow.match['ipv4\_dst'])):

key = (

stat.match['in\_port'], stat.match['ipv4\_dst'],

stat.instructions[0].actions[0].port)

value = (

stat.packet\_count, stat.byte\_count,

stat.duration\_sec, stat.duration\_nsec)

self.\_save\_stats(self.flow\_stats, key, value, 5)

# Get flow's speed.

pre = 0

period = SLEEP\_PERIOD

tmp = self.flow\_stats[key]

if len(tmp) > 1:

pre = tmp[-2][1]

period = self.\_get\_period(

tmp[-1][2], tmp[-1][3],

tmp[-2][2], tmp[-2][3])

speed = self.\_get\_speed(

self.flow\_stats[key][-1][1], pre, period)

self.\_save\_stats(self.flow\_speed, key, speed, 5)

@set\_ev\_cls(ofp\_event.EventOFPPortStatsReply, MAIN\_DISPATCHER)

def \_port\_stats\_reply\_handler(self, ev):

body = ev.msg.body

self.stats['port'][ev.msg.datapath.id] = body

for stat in sorted(body, key=attrgetter('port\_no')):

if stat.port\_no != ofproto\_v1\_3.OFPP\_LOCAL:

key = (ev.msg.datapath.id, stat.port\_no)

value = (

stat.tx\_bytes, stat.rx\_bytes, stat.rx\_errors,

stat.duration\_sec, stat.duration\_nsec)

self.\_save\_stats(self.port\_stats, key, value, 5)

# Get port speed.

pre\_tx = 0

pre\_rx = 0

period = SLEEP\_PERIOD

tmp = self.port\_stats[key]

if len(tmp) > 1:

#port\_stats every key contain five vaule,every value is like above.

#this shoud seperate to two,represent the transtion and receive

pre\_tx = tmp[-2][0]

pre\_rx = tmp[-2][1]

period = self.\_get\_period(

tmp[-1][3], tmp[-1][4],

tmp[-2][3], tmp[-2][4])

#this speed is the sum of trasmition and receive, i want to seperate it to two,and then

#transrate it to the \_save\_stats()

speed\_tx = self.\_get\_speed(

self.port\_stats[key][-1][0],

pre\_tx, period)

speed\_rx = self.\_get\_speed(

self.port\_stats[key][-1][1],

pre\_rx, period)

self.\_save\_stats(self.port\_speed, key, [speed\_tx,speed\_rx], 5)

@set\_ev\_cls(ofp\_event.EventOFPPortDescStatsReply, MAIN\_DISPATCHER)

def port\_desc\_stats\_reply\_handler(self, ev):

msg = ev.msg

dpid = msg.datapath.id

ofproto = msg.datapath.ofproto

config\_dist = {ofproto.OFPPC\_PORT\_DOWN: "Down",

ofproto.OFPPC\_NO\_RECV: "No Recv",

ofproto.OFPPC\_NO\_FWD: "No Farward",

ofproto.OFPPC\_NO\_PACKET\_IN: "No Packet-in"}

state\_dist = {ofproto.OFPPS\_LINK\_DOWN: "Down",

ofproto.OFPPS\_BLOCKED: "Blocked",

ofproto.OFPPS\_LIVE: "Live"}

ports = []

for p in ev.msg.body:

ports.append('port\_no=%d hw\_addr=%s name=%s config=0x%08x '

'state=0x%08x curr=0x%08x advertised=0x%08x '

'supported=0x%08x peer=0x%08x curr\_speed=%d '

'max\_speed=%d' %

(p.port\_no, p.hw\_addr,

p.name, p.config,

p.state, p.curr, p.advertised,

p.supported, p.peer, p.curr\_speed,

p.max\_speed))

if p.config in config\_dist:

config = config\_dist[p.config]

else:

config = "up"

if p.state in state\_dist:

state = state\_dist[p.state]

else:

state = "up"

port\_feature = (config, state, p.curr\_speed)

self.port\_link[dpid][p.port\_no] = port\_feature

#self.logger.debug('OFPPortDescStatsReply received: %s', ports)

@set\_ev\_cls(ofp\_event.EventOFPPortStatus, MAIN\_DISPATCHER)

def \_port\_status\_handler(self, ev):

msg = ev.msg

reason = msg.reason

port\_no = msg.desc.port\_no

dpid = msg.datapath.id

ofproto = msg.datapath.ofproto

reason\_dict = {ofproto.OFPPR\_ADD: "added",

ofproto.OFPPR\_DELETE: "deleted",

ofproto.OFPPR\_MODIFY: "modified", }

if reason in reason\_dict:

print "switch%d: port %s %s" % (dpid, reason\_dict[reason], port\_no)

else:

print "switch%d: Illeagal port state %s %s" % (port\_no, reason)

### 5.3网络路由模块

# conding=utf-8

import logging

import struct

from operator import attrgetter

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from ryu.controller.handler import MAIN\_DISPATCHER, DEAD\_DISPATCHER

from ryu.controller.handler import CONFIG\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.ofproto import ofproto\_v1\_3

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet

from ryu.lib.packet import ipv4

from ryu.lib.packet import arp

from ryu.controller import dpset

from ryu.topology import event, switches

from ryu.topology.api import get\_switch, get\_link

import string

import hashlib

import network\_aware

import network\_monitor

import time

class Shortest\_Route(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

\_CONTEXTS = {

"Network\_Aware": network\_aware.Network\_Aware,

"Network\_Monitor": network\_monitor.Network\_Monitor,

'dpset' : dpset.DPSet

}

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(Shortest\_Route, self).\_\_init\_\_(\*args, \*\*kwargs)

self.network\_aware = kwargs["Network\_Aware"]

self.network\_monitor = kwargs["Network\_Monitor"]

self.mac\_to\_port = {}

self.datapaths = {}

self.path\_list=[]

self.dpset=kwargs['dpset']

# links :(src\_dpid,dst\_dpid)->(src\_port,dst\_port)

self.link\_to\_port = self.network\_aware.link\_to\_port

# {sw :[host1\_ip,host2\_ip,host3\_ip,host4\_ip]}

self.access\_table = self.network\_aware.access\_table

# dpid->port\_num (ports without link)

self.access\_ports = self.network\_aware.access\_ports

self.graph = self.network\_aware.graph

self.i=0

#caculate the recovery time 2015.10.24

#self.detect\_time=0.0

@set\_ev\_cls(ofp\_event.EventOFPStateChange,

[MAIN\_DISPATCHER, DEAD\_DISPATCHER])

def \_state\_change\_handler(self, ev):

datapath = ev.datapath

if ev.state == MAIN\_DISPATCHER:

if not datapath.id in self.datapaths:

self.logger.debug('register datapath: %016x', datapath.id)

self.datapaths[datapath.id] = datapath

elif ev.state == DEAD\_DISPATCHER:

if datapath.id in self.datapaths:

self.logger.debug('unregister datapath: %016x', datapath.id)

del self.datapaths[datapath.id]

def add\_flow(self, dp, p, match, actions, idle\_timeout=0, hard\_timeout=0):

ofproto = dp.ofproto

parser = dp.ofproto\_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS,

actions)]

mod = parser.OFPFlowMod(datapath=dp, priority=p,

idle\_timeout=idle\_timeout,

hard\_timeout=hard\_timeout,

match=match, instructions=inst)

dp.send\_msg(mod)

def install\_flow(self, path, flow\_info, buffer\_id, data, priority):

'''

path=[dpid1, dpid2, dpid3...]

flow\_info=(eth\_type, src\_ip, dst\_ip, in\_port)

'''

# first flow entry

in\_port = flow\_info[3]

assert path

datapath\_first = self.datapaths[path[0]]

ofproto = datapath\_first.ofproto

parser = datapath\_first.ofproto\_parser

out\_port = ofproto.OFPP\_LOCAL

# inter\_link

if len(path) > 2:

for i in xrange(1, len(path) - 1):

port = self.get\_link2port(path[i - 1], path[i])

port\_next = self.get\_link2port(path[i], path[i + 1])

if port:

src\_port, dst\_port = port[1], port\_next[0]

datapath = self.datapaths[path[i]]

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

actions = []

actions.append(parser.OFPActionOutput(dst\_port))

match = parser.OFPMatch(

in\_port=src\_port,

eth\_type=flow\_info[0],

ipv4\_src=flow\_info[1],

ipv4\_dst=flow\_info[2])

self.add\_flow(

datapath, priority, match, actions,

idle\_timeout=10, hard\_timeout=120)

# inter links pkt\_out

msg\_data = None

if buffer\_id == ofproto.OFP\_NO\_BUFFER:

msg\_data = data

out = parser.OFPPacketOut(

datapath=datapath, buffer\_id=buffer\_id,

data=msg\_data, in\_port=src\_port, actions=actions)

datapath.send\_msg(out)

if len(path) > 1:

# the first flow entry

port\_pair = self.get\_link2port(path[0], path[1])

print port\_pair

out\_port = port\_pair[0]

actions = []

actions.append(parser.OFPActionOutput(out\_port))

match = parser.OFPMatch(

in\_port=in\_port,

eth\_type=flow\_info[0],

ipv4\_src=flow\_info[1],

ipv4\_dst=flow\_info[2])

self.add\_flow(datapath\_first,

priority, match, actions, idle\_timeout=10, hard\_timeout=120)

# the last hop: tor -> host

datapath = self.datapaths[path[-1]]

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

actions = []

src\_port = self.get\_link2port(path[-2], path[-1])[1]

dst\_port = None

for key in self.access\_table.keys():

if flow\_info[2] == self.access\_table[key]:

dst\_port = key[1]

break

actions.append(parser.OFPActionOutput(dst\_port))

match = parser.OFPMatch(

in\_port=src\_port,

eth\_type=flow\_info[0],

ipv4\_src=flow\_info[1],

ipv4\_dst=flow\_info[2])

self.add\_flow(

datapath, priority, match, actions, idle\_timeout=10, hard\_timeout=120)

# first pkt\_out

actions = []

actions.append(parser.OFPActionOutput(out\_port))

msg\_data = None

if buffer\_id == ofproto.OFP\_NO\_BUFFER:

msg\_data = data

out = parser.OFPPacketOut(

datapath=datapath\_first, buffer\_id=buffer\_id,

data=msg\_data, in\_port=in\_port, actions=actions)

datapath\_first.send\_msg(out)

# last pkt\_out

actions = []

actions.append(parser.OFPActionOutput(dst\_port))

msg\_data = None

if buffer\_id == ofproto.OFP\_NO\_BUFFER:

msg\_data = data

out = parser.OFPPacketOut(

datapath=datapath, buffer\_id=buffer\_id,

data=msg\_data, in\_port=src\_port, actions=actions)

datapath.send\_msg(out)

else: # src and dst on the same

out\_port = None

actions = []

for key in self.access\_table.keys():

if flow\_info[2] == self.access\_table[key]:

out\_port = key[1]

break

actions.append(parser.OFPActionOutput(out\_port))

match = parser.OFPMatch(

in\_port=in\_port,

eth\_type=flow\_info[0],

ipv4\_src=flow\_info[1],

ipv4\_dst=flow\_info[2])

self.add\_flow(

datapath\_first, priority, match, actions,

idle\_timeout=10, hard\_timeout=120)

# pkt\_out

msg\_data = None

if buffer\_id == ofproto.OFP\_NO\_BUFFER:

msg\_data = data

out = parser.OFPPacketOut(

datapath=datapath\_first, buffer\_id=buffer\_id,

data=msg\_data, in\_port=in\_port, actions=actions)

datapath\_first.send\_msg(out)

def get\_host\_location(self, host\_ip):

for key in self.access\_table:

if self.access\_table[key] == host\_ip:

return key

self.logger.debug("%s location is not found." % host\_ip)

return None

def get\_path(self, graph, src):

result = self.dijkstra(graph, src)

if result:

path = result[1]

#add\_path(path)

return path

self.logger.debug("Path is not found.")

return None

#store the path in pathlist

#next should consider the path removed, this should get the flow\_removed message and get the flow\_info,

#then seach it in path\_list, and then delete the relevent path.

def store\_path(self,path,flow\_info):

keyword = ''.join(map(str,path))

key = hashlib.md5(keyword).hexdigest()

if len(self.get\_path\_through\_key(key)) == 0:

tempPath = []

tempPath.append(path)

tempPath.append(flow\_info)

tempPath.append(key)

self.path\_list.append(tempPath)

#get\_path through the key

def get\_path\_through\_key(self,key):

temp = []

for x in self.path\_list:

if key == x[-1]:

temp = x[0]

break

return temp

#return the path attribute through the key,including the [path,flow\_info]

def get\_affect\_Path\_attribute(self,key):

temp = []

for x in self.path\_list:

if key == x[-1]:

temp.append(x[0])

temp.append(x[1])

break

return temp

#judge weather the delete link is in the path or not, if yes then return the key or then NULL

def haspath(self,dpids):

result = ""

for x in self.path\_list:

if dpids[0] in x[0] and dpids[-1] in x[0]:

a = x[0].index(dpids[0])

b = x[0].index(dpids[-1])

if abs(a-b) == 1:

result = x[-1]

#there may be multi infective path,but we now just consider one of them, if there more than one inffective path, we should store them in the []

break

return result

#detect the delete link ,and install new flow entry

@set\_ev\_cls(event.EventLinkDelete, MAIN\_DISPATCHER)

def link\_del\_handler(self, ev):

#install the detect time 2015.10.24

detect\_time=time.time()

link=ev.link

src\_dpid=link.src.dpid

dst\_dpid=link.dst.dpid

#update the graph and link2port

del self.link\_to\_port[(src\_dpid,dst\_dpid)]

self.graph[src\_dpid][dst\_dpid]=float('inf')

path\_key=self.haspath([src\_dpid,dst\_dpid])

if self.i==1:

if path\_key:

#install the direct path flow

affect\_path\_attribute=self.get\_affect\_Path\_attribute(path\_key)

flow\_info=affect\_path\_attribute[1]

#flow info:(eth\_type, ip\_src, ip\_dst,in\_port)

ip\_src = flow\_info[1]

ip\_dst = flow\_info[2]

result = None

src\_sw = None

dst\_sw = None

src\_location = self.get\_host\_location(ip\_src)

dst\_location = self.get\_host\_location(ip\_dst)

if src\_location:

src\_sw = src\_location[0]

if dst\_location:

dst\_sw = dst\_location[0]

result = self.dijkstra(self.graph, src\_sw)

if result:

path = result[1][src\_sw][dst\_sw]

path.insert(0, src\_sw)

self.logger.info(

" because the delete link , Reconfigure the PATH[%s --> %s]:%s\n" % (ip\_src, ip\_dst, path))

print self.graph

#store the path and the flow infomation in the path\_list

self.store\_path(path,flow\_info)

datapath= self.dpset.get(path[0])

ofproto=datapath.ofproto

self.install\_flow(path, flow\_info, ofproto.OFP\_NO\_BUFFER, "111",2)

#print "test the access\_table"

#print self.accss\_table

else:

# Reflesh the topology database.

self.network\_aware.get\_topology(None)

#install the reverse path flows,

flow\_info\_reverse=(flow\_info[0],flow\_info[2],flow\_info[1],flow\_info[3])

ip\_src\_reverse=flow\_info[2]

ip\_dst\_reverse=flow\_info[1]

result\_reverse=None

src\_sw\_reverse=None

dst\_sw\_reverse=None

src\_location\_reverse = self.get\_host\_location(ip\_src\_reverse)

dst\_location\_reverse = self.get\_host\_location(ip\_dst\_reverse)

if src\_location\_reverse:

src\_sw\_reverse = src\_location\_reverse[0]

if dst\_location\_reverse:

dst\_sw\_reverse = dst\_location\_reverse[0]

result\_reverse = self.dijkstra(self.graph,src\_sw\_reverse)

if result\_reverse:

path\_reverse = result\_reverse[1][src\_sw\_reverse][dst\_sw\_reverse]

path\_reverse.insert(0, src\_sw\_reverse)

self.logger.info(

" because the delete link , Reconfigure the PATH[%s --> %s]:%s\n" % (ip\_src\_reverse, ip\_dst\_reverse, path\_reverse))

#print self.graph

#store the path and the flow infomation in the path\_list

self.store\_path(path\_reverse,flow\_info\_reverse)

datapath= self.dpset.get(path\_reverse[0])

ofproto=datapath.ofproto

print "test the access\_table 22"

print self.access\_table

self.install\_flow(path\_reverse, flow\_info\_reverse, ofproto.OFP\_NO\_BUFFER, "111",2)

else:

# Reflesh the topology database.

self.network\_aware.get\_topology(None)

self.i=0

self.i=self.i+1

#caculate the recovery time 2015.10.24

recovery\_time=time.time()-detect\_time

print "++++++++++++++++++++++++++++++++++"

print "the recovery time is:"

print recovery\_time

def get\_link2port(self, src\_dpid, dst\_dpid):

if (src\_dpid, dst\_dpid) in self.link\_to\_port:

return self.link\_to\_port[(src\_dpid, dst\_dpid)]

else:

self.logger.debug("Link to port is not found.")

return None

def dijkstra(self, graph, src):

if graph is None:

self.logger.debug("Graph is empty.")

return None

length = len(graph)

type\_ = type(graph)

# Initiation

if type\_ == list:

nodes = [i for i in xrange(length)]

elif type\_ == dict:

nodes = graph.keys()

visited = [src]

path = {src: {src: []}}

if src not in nodes:

self.logger.debug("Src is not in nodes.")

return None

else:

nodes.remove(src)

distance\_graph = {src: 0}

pre = next = src

no\_link\_value = 100000

while nodes:

distance = no\_link\_value

for v in visited:

for d in nodes:

new\_dist = graph[src][v] + graph[v][d]

if new\_dist <= distance:

distance = new\_dist

next = d

pre = v

graph[src][d] = new\_dist

if distance < no\_link\_value:

path[src][next] = [i for i in path[src][pre]]

path[src][next].append(next)

distance\_graph[next] = distance

visited.append(next)

nodes.remove(next)

else:

self.logger.debug("Next node is not found.")

return None

return distance\_graph, path

@set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def \_packet\_in\_handler(self, ev):

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

in\_port = msg.match['in\_port']

pkt = packet.Packet(msg.data)

eth\_type = pkt.get\_protocols(ethernet.ethernet)[0].ethertype

arp\_pkt = pkt.get\_protocol(arp.arp)

ip\_pkt = pkt.get\_protocol(ipv4.ipv4)

if arp\_pkt:

arp\_src\_ip = arp\_pkt.src\_ip

arp\_dst\_ip = arp\_pkt.dst\_ip

# record the access info

if in\_port in self.access\_ports[datapath.id]:

self.access\_table[(datapath.id, in\_port)] = arp\_src\_ip

result = self.get\_host\_location(arp\_dst\_ip)

if result: # host record in access table.

datapath\_dst, out\_port = result[0], result[1]

actions = [parser.OFPActionOutput(out\_port)]

datapath = self.datapaths[datapath\_dst]

out = parser.OFPPacketOut(

datapath=datapath,

buffer\_id=ofproto.OFP\_NO\_BUFFER,

in\_port=ofproto.OFPP\_CONTROLLER,

actions=actions, data=msg.data)

datapath.send\_msg(out)

else: # access info is not existed. send to all host.

for dpid in self.access\_ports:

for port in self.access\_ports[dpid]:

if (dpid, port) not in self.access\_table.keys():

actions = [parser.OFPActionOutput(port)]

datapath = self.datapaths[dpid]

out = parser.OFPPacketOut(

datapath=datapath,

buffer\_id=ofproto.OFP\_NO\_BUFFER,

in\_port=ofproto.OFPP\_CONTROLLER,

actions=actions, data=msg.data)

datapath.send\_msg(out)

if isinstance(ip\_pkt, ipv4.ipv4):

ip\_src = ip\_pkt.src

ip\_dst = ip\_pkt.dst

result = None

src\_sw = None

dst\_sw = None

src\_location = self.get\_host\_location(ip\_src)

dst\_location = self.get\_host\_location(ip\_dst)

if src\_location:

src\_sw = src\_location[0]

if dst\_location:

dst\_sw = dst\_location[0]

result = self.dijkstra(self.graph, src\_sw)

if result[1]:

path = result[1][src\_sw][dst\_sw]

path.insert(0, src\_sw)

self.logger.info(

" PATH[%s --> %s]:%s\n" % (ip\_src, ip\_dst, path))

flow\_info = (eth\_type, ip\_src, ip\_dst, in\_port)

#store the path and the flow infomation in the path\_list

self.store\_path(path,flow\_info)

self.install\_flow(path, flow\_info, msg.buffer\_id, "111",1)

else:

# Reflesh the topology database.

self.network\_aware.get\_topology(None)

### 5.4网络延迟测量模块

from ryu.controller import handler

from ryu.controller import dpset

from ryu.controller import ofp\_event

from ryu.ofproto import ofproto\_v1\_3

from ryu.ofproto import ofproto\_v1\_3\_parser

from ryu.ofproto import inet

from ryu.ofproto import ether

from ryu.controller.handler import set\_ev\_cls

from ryu.base import app\_manager

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet

from ryu.lib.packet import arp

from ryu.lib.packet import vlan

from ryu.lib.packet import ipv4

from ryu.lib.packet import udp

from ryu.lib.packet import tcp

from ryu.ofproto.ofproto\_parser import MsgBase, msg\_str\_attr

from ryu.lib import mac

import netaddr

import array

import time

from ryu.topology.api import get\_switch, get\_link

from ryu.app.wsgi import ControllerBase

from ryu.topology import event, switches

from ryu.topology import api

import networkx as nx

from ryu.lib import hub

path = [1,2,3]

class PacketTest(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

\_CONTEXTS = {

'dpset': dpset.DPSet,

}

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(PacketTest,self).\_\_init\_\_(\*args,\*\*kwargs)

self.sw = {}

self.topology\_api\_app = self

self.net=nx.DiGraph()

self.nodes = {}

self.links = {}

self.dpset=kwargs['dpset']

self.no\_of\_nodes = 0

self.no\_of\_links = 0

self.i=0

self.path\_List = []

self.flows\_List = []

self.add\_flow\_time=[]

self.receveTestTime=0.0

self.sendTestTime=0.0

self.add\_flow\_path=[1,2,3]

#self.test\_delay\_path=[1,2,3]

self.limit=1

self.flow\_limit=1

time.sleep(1)

self.monitor\_thread = hub.spawn(self.testdelay)

def testdelay(self):

#hub.sleep(3)

#self.get\_topology\_data()

'''if(self.flow\_limit==1):

self.add\_path\_flow(self.add\_flow\_path)

self.flow\_limit += 1'''

#hub.sleep(3)

while True:

test\_delay\_path=[1,2,3]

dp= self.dpset.get(test\_delay\_path[0])

p = self.build\_udp()

link = self.net.get\_edge\_data(test\_delay\_path[0],test\_delay\_path[1])

if link:

port\_no = link['port']

self.send\_openflow\_packet(dp, p.data,port\_no)

print 'success'

hub.sleep(3)

hub.sleep(3)

def build\_udp(self):

#dst = '1' \* 6

dst = '00:00:00:00:00:01'

src = '00:00:00:00:00:02'

ethertype = ether.ETH\_TYPE\_8021Q

e = ethernet.ethernet(dst, src, ethertype)

v = vlan.vlan(1, 1, 3, ether.ETH\_TYPE\_IP)

ip = ipv4.ipv4(4, 5, 0, 0, 0, 0, 0, 255, 17, 33, '192.168.1.1', '192.168.1.11')

u = udp.udp(12, 34, 0)

p = packet.Packet()

p.add\_protocol(e)

p.add\_protocol(v)

p.add\_protocol(ip)

p.add\_protocol(u)

#data content

p.add\_protocol("123456789")

p.serialize()

return p

def send\_openflow\_packet(self, dp, packet, port\_no,

inport=ofproto\_v1\_3.OFPP\_CONTROLLER):

actions = [dp.ofproto\_parser.OFPActionOutput(port\_no)]

out=dp.ofproto\_parser.OFPPacketOut(datapath=dp,buffer\_id=dp.ofproto.OFP\_NO\_BUFFER,in\_port=dp.ofproto.OFPP\_CONTROLLER,actions=actions,data=packet)

dp.send\_msg(out)

self.sendTestTime=time.time()

print "the send time is %f and the send datapath is %d"%(self.sendTestTime,dp.id)

#dp.send\_packet\_out(in\_port=inport, actions=actions, data=packet)

#add flow

def add\_flow(self, datapath, priority, match, actions, buffer\_id=None, flags=0):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS,

actions)]

if buffer\_id:

mod = parser.OFPFlowMod(datapath=datapath, buffer\_id=buffer\_id, idle\_timeout=0, hard\_timeout=0,

priority=priority, match=match,

instructions=inst, flags=flags)

else:

mod = parser.OFPFlowMod(datapath=datapath, priority=priority, idle\_timeout=0, hard\_timeout=0,

match=match, instructions=inst)

datapath.send\_msg(mod)

print '%%%%%%%%%%%'

#get topology

@set\_ev\_cls(event.EventSwitchEnter)

def get\_topology\_data(self,ev):

time.sleep(0.01)

switch\_list = get\_switch(self.topology\_api\_app, None)

switches=[switch.dp.id for switch in switch\_list]

self.net.add\_nodes\_from(switches)

links\_list = get\_link(self.topology\_api\_app, None)

links=[(link.src.dpid,link.dst.dpid,{'port':link.src.port\_no}) for link in links\_list]

self.net.add\_edges\_from(links)

links=[(link.dst.dpid,link.src.dpid,{'port':link.dst.port\_no}) for link in links\_list]

self.net.add\_edges\_from(links)

print "\*\*\*\*\*\*\*\*\*\*List of links"

print self.net.edges()

'''while(self.limit>0):

self.testdelay(self.test\_delay\_path)

self.limit-=1'''

#hub.sleep(3)

def add\_path\_flow(self,path):

i=0

pathlen=len(path)

while i<pathlen-1:

datapath = self.dpset.get(path[i])

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

dpid = path[i]

link = self.net.get\_edge\_data(path[i],path[i+1])

if link:

out\_port = link['port']

match = parser.OFPMatch(ipv4\_src='192.168.1.1')

actions = [datapath.ofproto\_parser.OFPActionOutput(out\_port)]

self.add\_flow(datapath, 1, match, actions)

#print "add a flow"

#print "match is"

#print match

else:

print 'no link'

i=i+1

@handler.set\_ev\_cls(ofp\_event.EventOFPPacketIn, handler.MAIN\_DISPATCHER)

def packet\_in\_handler(self, ev):

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

pkt = packet.Packet(msg.data)

nw=pkt.get\_protocol(ipv4.ipv4)

pkt\_arp = pkt.get\_protocol(arp.arp)

#if pkt\_arp:

#print 'paket is arp'

if(nw):

ip\_src = nw.src

print datapath.id

print '///////////'

print ip\_src

if(ip\_src=='192.168.1.1'):

print'+++++++receive the test packet from %d '% datapath.id

#print msg.data

self.receveTestTime=time.time()

print 'the receive time is %f'%self.receveTestTime

print'\*\*\*\*\*\*\*\*\*\*caculate the delay time is \*\*\*\*\*\*'

delaytime=self.receveTestTime-self.sendTestTime

f=open('/home/john/Data/delay.txt','a')

f.write(str(delaytime)+'\n')

f.close()

print delaytime