SDN-Lab3

前言

实验目的:

1. 学习利用 ryu.topology.api 发现网络拓扑
2. 学习利用 LLDP 和 Echo 数据包测量链路时延
3. 学习计算基于跳数和基于时延的最短路由
4. 学习设计能够容忍链路故障的路由策略
5. 分析网络集中式控制与分布式控制的差异, 思考 SDN 的得与失

实验背景:

时间来到 1970 年,在你的建设下 ARPANET 飞速发展,在一年内从原来西部 4 个结点组成的简单网络逐渐发展为拥有 9 个结点,横跨东西海岸,初具规模的网络。

ARPANET 的拓展极大地便利了东西海岸之间的通信,但用户仍然十分关心网络服务的性能。一条时
延较小的转发路由将显著提升用户体验,尤其是在一些实时性要求很高的应用场景下。另外,路由策略对网络故障的容忍能力也是影响用户体验的重要因素,好的路由策略能够向用户隐藏一定程度的链路故障,使得个别链路断开后用户间的通信不至于中断。

SDN 是一种集中式控制的网络架构,控制器可以方便地获取网络拓扑、各链路和交换机的性能指标、网络故障和拓扑变化等全局信息,这也是 SDN 的优势之一。在掌握全局信息的基础上,SDN 就能实现更高效、更健壮的路由策略。

在正式任务之前,为帮助同学们理解,本指导书直接给出了一个示例。请运行示例程序,理解怎样利用 ryu.topology.api 获取网络拓扑,并计算 **跳数最少的路由**。

跳数最少的路由不一定是最快的路由,在实验任务一中,你将学习怎样利用 LLDP 和 Echo 数据包
测量链路时延,并计算 **时延最小的路由**。

1970 年的网络硬件发展尚不成熟,通信链路和交换机端口发生故障的概率较高。在实验任务二中,
你将学习在链路不可靠的情况下,设计 **对链路故障有一定容忍能力的路由策略**。

实验备注:

为了更加细致地展示代码分析，从这章教程开始，每一部分的实验代码将以"节选"形式展示，保留关键内容并进行分析，实验完整代码在最后的附录内 :)

> If you are interested, please come to my sdnRepo for whole scripts

实验0 最少跳数路径

预备知识

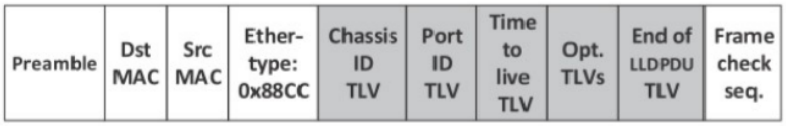
> 用 ryu.topology.api 获取网络拓扑,并计算 **跳数最少的路由**

拓扑感知:

控制器首先要获取网络的拓扑结构,才能够对网络进行各种测量分析,网络拓扑主要包括主机、链路和交换机的相关信息

链路发现原理:

LLDP(Link Layer Discover Protocol) 即链路层发现协议,Ryu 主要利用 LLDP 发现网络拓扑。LLDP 被封装在以太网帧中,结构如下图:



其中深灰色的即为 LLDP 负载,Chassis ID TLV , Port ID TLV 和 Time to live TLV 是三个强制字段,分别代表交换机标识符(在局域网中是独一无二的),端口号和 TTL

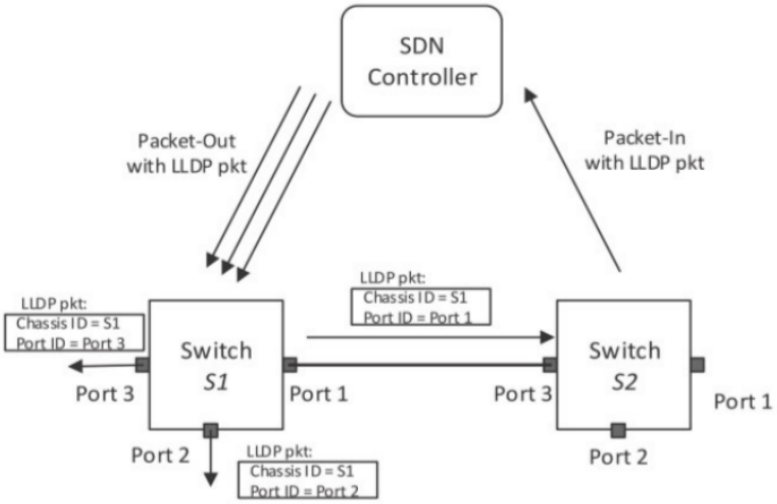
Ryu 如何利用 LLDP 发现链路: (假设有两个OpenFlow交换机连接在控制器上)

1. SDN 控制器构造 PacketOut 消息向 S1 的三个端口分别发送 LLDP 数据包,其中将 Chassis ID TLV 和 Port ID TLV 分别置为 S1 的 dpid 和端口号;
2. 控制器向交换机 S1 中下发流表,流表规则为: 将从 Controller 端口收到的 LLDP 数据包从他的对应端口发送出去;
3. 控制器向交换机 S2 中下发流表,流表规则为: 将从非 Controller 接收到的 LLDP 数据包发送给控制器;
4. 控制器通过解析 LLDP 数据包,得到链路的源交换机,源接口,通过收到的 PacketIn 消息知道目的交换机和目的接口。

沉默主机现象:

主机如果没有主动发送过数据包,控制器就无法发现主机 。运行前面的 NetworkAwareness.py 时, 你可能会看到 host 输出为空,这就是沉默主机现象导致的。

你可以在 `mininet` 中运行 `pingall` 指令,令每个主机发出 `ICMP` 数据包,这样控制器就能够发现主机。当然命令的结果是 `ping` 不通, 因为程序中并没有下发路由的代码



实验代码

```
# NetworkAwareness.py (offered by TA as a reference)
from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller import ofp_event
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib import hub
from ryu.topology.api import get_all_host, get_all_link, get_all_switch


class NetworkAwareness(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(NetworkAwareness, self).__init__(*args, **kwargs)
        self.dpid_mac_port = {}
        self.topo_thread = hub.spawn(self._get_topology)

    def add_flow(self, datapath, priority, match, actions):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions
```

```
        mod = parser.OFPFlowMod(datapath=dp, priority=priority, match=match,
        dp.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_
        self.add_flow(dp, 0, match, actions)

    def _get_topology(self):
        while True:
            self.logger.info('\n\n\n')

            hosts = get_all_host(self)
            switches = get_all_switch(self)
            links = get_all_link(self)

            self.logger.info('hosts:')
            for hosts in hosts:
                self.logger.info(hosts.to_dict())

            self.logger.info('switches:')
            for switch in switches:
                self.logger.info(switch.to_dict())

            self.logger.info('links:')
            for link in links:
                self.logger.info(link.to_dict())

            hub.sleep(2)
```

实验1 链路时延测量

预备知识

链路时延计算:

控制器将带有时间戳的 LLDP 报文下发给 S1 , S1 转发给 S2 , S2 上传回控制器(即内圈红色箭头的路径),根据收到的时间和发送时间即可计算出控制器经 S1 到 S2 再返回控制器的

时延,记为
lldp_delay_s12

反之,控制器经 S2 到 S1 再返回控制器的时延,记为 lldp_delay_s21
交换机收到控制器发来的Echo报文后会立即回复控制器,我们可以利用 Echo Request/Reply 报文求出控制器到 S1 、 S2 的往返时延,记为 echo_delay_s1 , echo_delay_s2

则 S1 到 S2 的时延:

> $delay = (lldp\_delay\_s12 + lldp\_delay\_s21 - echo\_delay\_s1 - echo\_delay\_s2)/2$

修改Ryu源码并编译

1. ryu/topology/Switches.py 的 PortData/**init**():

PortData 记录交换机的端口信息, 我们需要增加 self.delay 属性记录上述的 lldp_delay
self.timestamp 为 LLDP 包在发送时被打上的时间戳

```
class PortData(object):
    def __init__(self, is_down, lldp_data):
        super(PortData, self).__init__()
        self.is_down = is_down
        self.lldp_data = lldp_data
        self.timestamp = None
        self.sent = 0
        self.delay = 0
```

2. ryu/topology/Switches/lldp_packet_in_handler():

lldp_packet_in_handler() 处理接收到的 LLDP 包, 在这里用收到 LLDP 报文的时间戳减去发送时的时间戳即为 lldp_delay, 由于 LLDP 报文被设计为经一跳后转给控制器, 我们可将 lldp_delay 存入发送 LLDP 包对应的交换机端口

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def lldp_packet_in_handler(self, ev):
    # add receive timestamp
    recv_timestamp = time.time()
    if not self.link_discovery:
        return
    msg = ev.msg
    try:
```

```
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
    except LLDPPacket.LLDPUnknownFormat:
        # This handler can receive all the packets which can be
        # not-LLDP packet. Ignore it silently
        return
    # calc the delay of lldp packet
    for port, port_data in self.ports.items():
        if src_dpid == port.dpid and src_port_no == port.port_no:
            send_timestamp = port_data.timestamp
            if send_timestamp:
                port_data.delay = recv_timestamp - send_timestamp
```

测量 LLDP Delay

1. add: `self.lldp_delay = {}` in `__init(...)__`

```
def __init__(self, *args, **kwargs):
        super(NetworkAwareness, self).__init__(*args, **kwargs)
        # basic config info of a topo
        self.switch_info = {}  # dpid: datapath
        self.link_info = {}  # (s1, s2): s1.port
        self.port_link={} # s1,port:s1,s2
        self.port_info = {}  # dpid: (ports linked hosts)
        self.topo_map = nx.Graph()
        self.topo_thread = hub.spawn(self._get_topology)
        self.lldp_delay = {}
        self.delay = {}
        self.controller_switch_delay = {}

        # define weight of path
        self.weight = 'hop' # use "hopNumber between 2 nodes" as weight in Sh
        self.switches = None
```

2. add `packet_in_handler(...)`

```
def packet_in_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser
    dpid = dp.id

    pkt = packet.Packet(msg.data)
```

```
    eth_pkt = pkt.get_protocol(ethernet.ethernet)
    arp_pkt = pkt.get_protocol(arp.arp)
    pkt_type = eth_pkt.ethertype

    if pkt_type == ether_types.ETH_TYPE_LLDP:
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
        if self.switches is None:
            self.switches = lookup_service_brick('switches')

        for port in self.switches.ports.keys():
            if src_dpid == port.dpid and src_port_no == port.port_no:
                self.lldp_delay[(src_dpid,dpid)] = self.switches.ports[port].
```

测量 ECHO Delay

让控制器给交换机发送 ECHO request 消息，然后再让交换机返回给控制器，通过两个时间戳来
得到 ECHO 时间，同时这里时间差也乘以 1000，保证与上述单位一致

```
@set_ev_cls(ofp_event.EventOFPEchoReply, [MAIN_DISPATCHER,CONFIG_DISPATCHER,H
    def echo_reply_handler(self, ev):
        now_timestamp = time.time()
        try:
            echo_delay = now_timestamp - eval(ev.msg.data)
            self.controller_switch_delay[ev.msg.datapath.id] = echo_delay * 1
        except:
            print ("Overtime! Error!")
            return

    def send_echo_request(self, switch):
        datapath = switch.dp
        parser = datapath.ofproto_parser
        echo_req = parser.OFPEchoRequest(datapath,data=bytes("%.12f"%time.tim
        datapath.send_msg(echo_req)
```

计算总时延

特殊情况分析：

以下是计算总时延的代码，_get_topology() 函数本身是一个不断运行的线程,所以我们在这里获取时延,来得到不断的时延更新。

值得注意的是，这里计算 echo_request 的过程中需要取"冷却时间"来暂停一下，如果不暂停的话，echo delay 非常大，不符合运转过程(这里我取的是 hub.sleep(0.5))。

猜测原因是：如果快速给每个交换机发送 `echo_request`，每个交换机会同时快速发送 `echo_reply` 到控制器,而控制器没办法一下子同时处理，则产生排队时延

因此：会降低`ofp_event.EventOFPEchoReply` 的处理速度,导致测量误差非常大

测量并计算总时延代码：

```
def _get_topology(self):
    _hosts, _switches, _links = None, None, None
    while True:
        hosts = get_host(self)
        switches = get_switch(self)
        links = get_link(self)

        # If topo is still, continue
        if [str(x) for x in hosts] == _hosts and [str(x) for x in switches] =
            continue

        # update topo_map when topology change
        _hosts, _switches, _links = [str(x) for x in hosts], [str(x) for x ir

        for switch in switches: # ports & Sw.
            self.port_info.setdefault(switch.dp.id, set()) # origin, a Sw. wi
            # record all ports in this Sw.
            for port in switch.ports:
                self.port_info[switch.dp.id].add(port.port_no) # attach ports
            self.send_echo_request(switch)
            hub.sleep(0.5)

        for host in hosts: # Sw. & hosts
            # take one ipv4 address as host id
            if host.ipv4:
                self.link_info[(host.port.dpid, host.ipv4[0])] = host.port.pc
                self.topo_map.add_edge(host.ipv4[0], host.port.dpid, hop=1, c

        for link in links:
            # delete link (host-port -- link -- host-port)
            self.port_info[link.src.dpid].discard(link.src.port_no)
            self.port_info[link.dst.dpid].discard(link.dst.port_no)

            # s1 -> s2: s1.port, s2 -> s1: s2.port
            self.port_link[(link.src.dpid,link.src.port_no)]=(link.src.dpid,
            self.port_link[(link.dst.dpid,link.dst.port_no)] = (link.dst.dpic

            self.link_info[(link.src.dpid, link.dst.dpid)] = link.src.port_nc
            self.link_info[(link.dst.dpid, link.src.dpid)] = link.dst.port_nc
```

```
            # self.topo_map.add_edge(link.src.dpid, link.dst.dpid, hop=1, is_

            delay_src_to_dst = 0.0
            delay_dst_to_src = 0.0
            delay_ctl_to_src = 0.0
            delay_ctl_to_dst = 0.0

            if (link.src.dpid, link.dst.dpid) in self.lldp_delay:
                delay_src_to_dst = self.lldp_delay[(link.src.dpid, link.dst.c

            if (link.dst.dpid, link.src.dpid) in self.lldp_delay:
                delay_dst_to_src = self.lldp_delay[(link.dst.dpid, link.src.c

            if link.src.dpid in self.controller_switch_delay:
                delay_ctl_to_src = self.controller_switch_delay[link.src.dpic

            if link.dst.dpid in self.controller_switch_delay:
                delay_ctl_to_dst = self.controller_switch_delay[link.dst.dpic

            delay = (delay_src_to_dst + delay_dst_to_src +
                     delay_ctl_to_src + delay_ctl_to_dst) / 2

            if (delay < 0):
                delay = 0

            self.delay[(link.src.dpid, link.dst.dpid)] = delay
            self.topo_map.add_edge(link.src.dpid, link.dst.dpid, hop = 1, del

        if self.weight == 'hop':
            self.show_topo_map_1()
        if self.weight == 'delay':
            self.show_topo_map_2()

        hub.sleep(GET_TOPOLOGY_INTERVAL)
```

这里如果计算出 `delay<0` 时,让 `delay` 为 `0`。
BUT! 经过多次实验,发现不存在 `delay<0` 的情况, 有理由怀疑是上述情况导致的?

实验结果与验证

实际延迟测量

```
topo map:
   node     ->     node         delay
    1               9           10.6ms
    2               3           11.5ms
    2               4           13.5ms
    3               4           14.5ms
    4               5           15.6ms
    5               9           29.5ms
    5               6           18.0ms
    6               7           10.6ms
    7               8           62.6ms
    8               9           17.6ms
```

理论设置

```
59          # add edges between switches
60          self.addLink( s1 , s9, bw=10, delay='10ms')
61          self.addLink( s2 , s3, bw=10, delay='11ms')
62          self.addLink( s2 , s4, bw=10, delay='13ms')
63          self.addLink( s3 , s4, bw=10, delay='14ms')
64          self.addLink( s4 , s5, bw=10, delay='15ms')
65          self.addLink( s5 , s9, bw=10, delay='29ms')
66          self.addLink( s5 , s6, bw=10, delay='17ms')
67          self.addLink( s6 , s7, bw=10, delay='10ms')
68          self.addLink( s7 , s8, bw=10, delay='62ms')
69          self.addLink( s8 , s9, bw=10, delay='17ms')
```

该权重选择下的链路选择

```
*** Starting CLI:
mininet> SDC ping -i 0.5 MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=10 ttl=64 time=69.9 ms
64 bytes from 10.0.0.3: icmp_seq=11 ttl=64 time=127 ms
64 bytes from 10.0.0.3: icmp_seq=12 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=13 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=14 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=15 ttl=64 time=126 ms
^C
--- 10.0.0.3 ping statistics ---
15 packets transmitted, 6 received, 60% packet loss, time 7106ms
rtt min/avg/max/mdev = 69.951/117.051/127.727/21.075 ms
mininet>
```

```
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:2 -> 4:s5:3 -> 3:s9:4 -> 3:s8:1 -> 10.0.0.3    delay=64.7ms
topo map:
   node     ->     node         delay
    1               9           10.5ms
    2               3           11.6ms
    2               4           13.6ms
    3               4           14.6ms
    4               5           16.0ms
    5               9           29.7ms
    5               6           17.6ms
    6           10.0.0.5        0.0ms
    6               7           10.5ms
    7               8           62.8ms
    8               9           17.4ms
    8           10.0.0.3        0.0ms
```

实验2： 容忍链路故障

实验背景

1970年的网络硬件发展尚不成熟,通信链路和交换机端口发生故障的概率较高。请设计 Ryu app ,在任务一的基础上实现容忍链路故障的路由选择：每当链路出现故障时,重新选择当前可用路径中时延最低的路径;当链路故障恢复后,也重新选择新的时延最低的路径。
请在实验报告里附上你计算的:
(1)最小时延路径 (2)最小时延路径的 RTT (3)链路故障/恢复后发生的路由转移

模拟链路故障:

```
mininet>link s1 s4 down
mininet>link s1 s4 up
```

控制器捕捉链路故障:

链路状态改变时,链路关联的端口状态也会变化,从而产生端口状态改变的事件,即 EventOFPPortStatus, 通过将此事件与你设计的处理函数绑定在一起,就可以获取状态改变的信息, 执行相应的处理。可以在你的代码中实现你需要的 EventOFPPortStatus 事件处理函数。

OFPFC_DELETE 消息:

与向交换机中增加流表的 OFPFC_ADD 命令不同, OFPFC_DELETE 消息用于删除交换机中符合匹配项的所有流表。

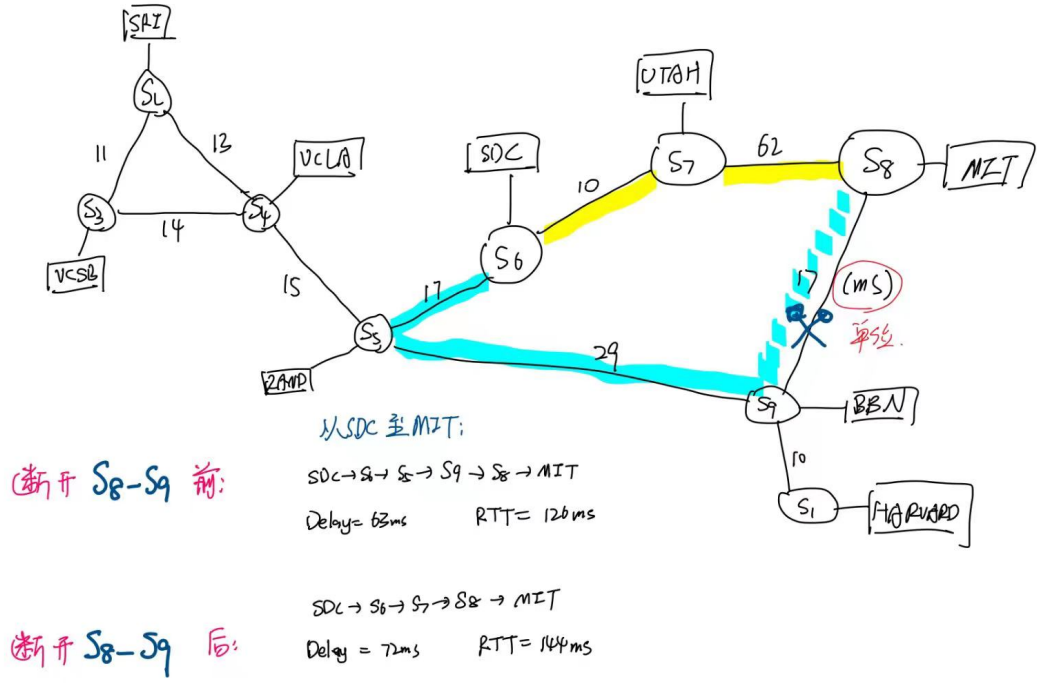
由于添加和删除都属于 OFPFlowMod 消息, 因此只需稍微修改 add_flow( ) 函数, 即可生成
delete_flow( ) 函数。

Packet_In 消息的合理利用:

基本思路是：在链路发生改变时, 删除受影响的链路上所有交换机上的相关流表的信息,下一次交换机将匹配默认流表项, 向控制器发送 packet_in 消息, 控制器重新计算并下发最小时延路径。

理论分析

对于原拓扑：考察断开S8-S9后的情景



代码设计

OFPFC_DELETE: 几乎跟 add_flow() 一模一样, 略之!

```
def delete_flow(self, datapath, match):
    # get this on line
    dp = datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser
    inst = []

    req = parser.OFPFlowMod (
          dp,
          0, 0, 0, ofp.OFPFC_DELETE,
          0, 0, 0, ofp.OFP_NO_BUFFER,
          ofp.OFPP_ANY,
          ofp.OFPG_ANY,
          ofp.OFPFF_SEND_FLOW_REM,
          match,
          inst
    )
    dp.send_msg(req)
```

链路断裂带来的主机/交换机行为:

```
@set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def port_status_handler(self, ev):
        msg = ev.msg
        datapath = ev.datapath
        ofproto = ev.ofproto
        parser = datapath.ofproto_parser

        if msg.reason in [ofproto.OFPPR_ADD, ofproto.OFPPR_MODIFY]:
            datapath.ports[msg.desc.port_no] = msg.desc
            self.topo_map.clear()

            for dpid in self.port_info.keys():
                for port in self.port_info[dpid]:
                    match = parser.OFPMatch(in_port = port)
                    self.delete_flow(self.switch_info[dpid],match)
        elif msg.reason == ofproto.OFPPR_DELETE:
            datapath.ports.pop(msg.desc.port_no, None)
        else:
            return
self.send_event_to_observers(ofp_event.EventOFPPortStateChange(datapath,msg.r
```

解释：这是删除所有交换机连向的主机端口，让主机下次 `ping` 必定触发 `Packetin` 消息。下一次交换机将匹配默认流表项(`prio=0`)，向控制器发送 `packet_in` 消息，控制器重新计算并下发最小时延路径

实验结果与验证

在上述理论分析的拓扑中，考察`SDC`到`MIT`，故障情景同上

初始正常链路

```
ms delay)
BBN BBN-eth0:s9-eth1
HARVARD HARVARD-eth0:s1-eth1
MIT MIT-eth0:s8-eth1
RAND RAND-eth0:s5-eth1
SDC SDC-eth0:s6-eth1
SRI SRI-eth0:s2-eth1
UCLA UCLA-eth0:s4-eth1
UCSB UCSB-eth0:s3-eth1
UTAH UTAH-eth0:s7-eth1
*** Starting CLI:
mininet> SDC ping MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=70.7 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=9 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=10 ttl=64 time=126 ms
^C
--- 10.0.0.3 ping statistics ---
10 packets transmitted, 6 received, 40% packet loss, time 9077ms
rtt min/avg/max/mdev = 70.712/117.104/126.642/20.749 ms
mininet>
```

```
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:2 -> 4:s5:3 -> 3:s9:4 -> 3:s8:1 -> 10.0.0.3    delay=64.7ms
```

RTT = 126ms, 路径跟理论分析的也一样

故障发生后调整链路

```
mininet> link s8 s9 down
mininet> SDC ping MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=76.6 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=144 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=144 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=144 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=144 ms
^C
--- 10.0.0.3 ping statistics ---
6 packets transmitted, 5 received, 16% packet loss, time 5026ms
rtt min/avg/max/mdev = 76.623/130.791/144.471/27.085 ms
mininet>
```

```
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:3 -> 2:s7:3 -> 2:s8:1 -> 10.0.0.3    delay=73.3ms
```

RTT = 144ms, 路径跟理论分析的也一样

故障恢复后的链路

```
mininet> link s8 s9 up
mininet> SDC ping MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=70.7 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=126 ms
64 bytes from 10.0.0.3: icmp_seq=9 ttl=64 time=126 ms
^C
--- 10.0.0.3 ping statistics ---
```

```
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:2 -> 4:s5:3 -> 3:s9:4 -> 3:s8:1 -> 10.0.0.3    delay=64.4ms
```

RTT = 126ms，路径跟故障发生前一样，说明恢复成功

附录

network_awareness.py

```python
from ryu.base import app_manager
from ryu.base.app_manager import lookup_service_brick
from ryu.ofproto import ofproto_v1_3
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER, DEAD_D
from ryu.controller import ofp_event
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet, arp
from ryu.lib.packet import ether_types
from ryu.lib import hub
from ryu.topology import event
from ryu.topology.api import get_host, get_link, get_switch
from ryu.topology.switches import LLDPPacket

import networkx as nx
import copy
import time

# this script is to get network topology by controller
# with the help of Ryu
# method:
'''
    sudo mn --topo=tree,2,2 --controller remote
    ryu-manager NetworkAwareness.py --observe-links
'''

GET_TOPOLOGY_INTERVAL = 2
SEND_ECHO_REQUEST_INTERVAL = 0.05
```

```
GET_DELAY_INTERVAL = 2


class NetworkAwareness(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(NetworkAwareness, self).__init__(*args, **kwargs)
        # basic config info of a topo
        self.switch_info = {}  # dpid: datapath
        self.link_info = {}  # (s1, s2): s1.port
        self.port_link={} # s1,port:s1,s2
        self.port_info = {}  # dpid: (ports linked hosts)
        self.topo_map = nx.Graph()
        self.topo_thread = hub.spawn(self._get_topology)
        self.lldp_delay = {}
        self.delay = {}
        self.controller_switch_delay = {}

        # define weight of path
        self.weight = 'hop' # use "hopNumber between 2 nodes" as weight in Sh
        self.switches = None

    def add_flow(self, datapath, priority, match, actions):
        # add an object to flow_table
        dp = datapath # Sw.
        ofp = dp.ofproto # objects and consts about OpenFlow itself
        parser = dp.ofproto_parser # parser of OpenFlow

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions
        # message downing
        mod = parser.OFPFlowMod(datapath=dp, priority=priority, match=match,
        '''
        mod = message downing to Sw.
        dp = Sw.
        prio = the processing Prio.
        match = matching condition
        inst = corresponding actions
        '''
        dp.send_msg(mod)

    def delete_flow(self, datapath, match):
        # get this on line
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
```

```
        inst = []

        req = parser.OFPFlowMod (
              dp,
              0, 0, 0, ofp.OFPFC_DELETE,
              0, 0, 0, ofp.OFP_NO_BUFFER,
              ofp.OFPP_ANY,
              ofp.OFPG_ANY,
              ofp.OFPFF_SEND_FLOW_REM,
              match,
              inst
        )
        dp.send_msg(req)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        # 在交换机连接时向其添加一个默认的流表项，使所有未匹配到的流量都发送到控制器
        msg = ev.msg # message of packet
        dp = msg.datapath # Sw.
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        match = parser.OFPMatch() # MatchObject = None, matching all flows
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_
        '''
        ofp.OFPP_CONTROLLER: Controller Receiving Port
        ofp.OFPCML_NO_BUFFER: no Buffer for this msg
        '''
        self.add_flow(dp, 0, match, actions) # matching all, so prio = 0

    @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def port_status_handler(self, ev):
        msg = ev.msg
        datapath = ev.datapath
        ofproto = ev.ofproto
        parser = datapath.ofproto_parser

        if msg.reason in [ofproto.OFPPR_ADD, ofproto.OFPPR_MODIFY]:
            datapath.ports[msg.desc.port_no] = msg.desc
            self.topo_map.clear()

            for dpid in self.port_info.keys():
                for port in self.port_info[dpid]:
                    match = parser.OFPMatch(in_port = port)
                    self.delete_flow(self.switch_info[dpid],match)
        elif msg.reason == ofproto.OFPPR_DELETE:
```

```
            datapath.ports.pop(msg.desc.port_no, None)
        else:
            return

        self.send_event_to_observers(ofp_event.EventOFPPortStateChange(datapa

    @set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATC
    def state_change_handler(self, ev):
        # 当交换机状态改变时（连接或断开），会触发该方法：处理交换机连接和断开事件，更新内
        dp = ev.datapath # Sw.
        dpid = dp.id # Sw.ID

        if ev.state == MAIN_DISPATCHER: # Sw. is in connection (ON)
            self.switch_info[dpid] = dp # add map<ID, Sw.>

        if ev.state == DEAD_DISPATCHER: # Sw. is out of connection (OFF)
            del self.switch_info[dpid] # del ...

    @set_ev_cls(ofp_event.EventOFPEchoReply, [MAIN_DISPATCHER,CONFIG_DISPATCH
    def echo_reply_handler(self, ev):
        now_timestamp = time.time()
        try:
            echo_delay = now_timestamp - eval(ev.msg.data)
            self.controller_switch_delay[ev.msg.datapath.id] = echo_delay * 1
        except:
            print ("Overtime! Error!")
            return

    def send_echo_request(self, switch):
        datapath = switch.dp
        parser = datapath.ofproto_parser
        echo_req = parser.OFPEchoRequest(datapath,data=bytes("%.12f"%time.tim
        datapath.send_msg(echo_req)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        dpid = dp.id

        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        arp_pkt = pkt.get_protocol(arp.arp)
        pkt_type = eth_pkt.ethertype
```

```
        if pkt_type == ether_types.ETH_TYPE_LLDP:
            src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
            if self.switches is None:
                self.switches = lookup_service_brick('switches')

            for port in self.switches.ports.keys():
                if src_dpid == port.dpid and src_port_no == port.port_no:
                    self.lldp_delay[(src_dpid,dpid)] = self.switches.ports[pc

    def _get_topology(self):
        _hosts, _switches, _links = None, None, None
        while True:
            hosts = get_host(self)
            switches = get_switch(self)
            links = get_link(self)

            # If topo is still, continue
            if [str(x) for x in hosts] == _hosts and [str(x) for x in switche
                continue

            # update topo_map when topology change
            _hosts, _switches, _links = [str(x) for x in hosts], [str(x) for

            for switch in switches: # ports & Sw.
                self.port_info.setdefault(switch.dp.id, set()) # origin, a Sw
                # record all ports in this Sw.
                for port in switch.ports:
                    self.port_info[switch.dp.id].add(port.port_no) # attach p
                self.send_echo_request(switch)
                hub.sleep(0.5)

            for host in hosts: # Sw. & hosts
                # take one ipv4 address as host id
                if host.ipv4:
                    self.link_info[(host.port.dpid, host.ipv4[0])] = host.por
                    self.topo_map.add_edge(host.ipv4[0], host.port.dpid, hop=

            for link in links:
                # delete link (host-port -- link -- host-port)
                self.port_info[link.src.dpid].discard(link.src.port_no)
                self.port_info[link.dst.dpid].discard(link.dst.port_no)

                # s1 -> s2: s1.port, s2 -> s1: s2.port
                self.port_link[(link.src.dpid,link.src.port_no)]=(link.src.dp
                self.port_link[(link.dst.dpid,link.dst.port_no)] = (link.dst.
```

```
                self.link_info[(link.src.dpid, link.dst.dpid)] = link.src.por
                self.link_info[(link.dst.dpid, link.src.dpid)] = link.dst.por
                # self.topo_map.add_edge(link.src.dpid, link.dst.dpid, hop=1,

                delay_src_to_dst = 0.0
                delay_dst_to_src = 0.0
                delay_ctl_to_src = 0.0
                delay_ctl_to_dst = 0.0

                if (link.src.dpid, link.dst.dpid) in self.lldp_delay:
                    delay_src_to_dst = self.lldp_delay[(link.src.dpid, link.c

                if (link.dst.dpid, link.src.dpid) in self.lldp_delay:
                    delay_dst_to_src = self.lldp_delay[(link.dst.dpid, link.s

                if link.src.dpid in self.controller_switch_delay:
                    delay_ctl_to_src = self.controller_switch_delay[link.src.

                if link.dst.dpid in self.controller_switch_delay:
                    delay_ctl_to_dst = self.controller_switch_delay[link.dst.

                delay = (delay_src_to_dst + delay_dst_to_src +
                         delay_ctl_to_src + delay_ctl_to_dst) / 2

                if (delay < 0):
                    delay = 0

                self.delay[(link.src.dpid, link.dst.dpid)] = delay
                self.topo_map.add_edge(link.src.dpid, link.dst.dpid, hop = 1,

            if self.weight == 'hop':
                self.show_topo_map_1()
            if self.weight == 'delay':
                self.show_topo_map_2()

            hub.sleep(GET_TOPOLOGY_INTERVAL)

    def shortest_path(self, src, dst, weight='hop'):
        try:
            # import nx to solve "shortest path" automatically
            paths = list(nx.shortest_simple_paths(self.topo_map, src, dst, we
            return paths[0]
        except:
            self.logger.info('host not find/no path')
```

```
    def show_topo_map_1(self):
        self.logger.info('topo map:')
        self.logger.info('{:^10s} -> {:^10s} {}'.format('node', 'node','hop')
        for src, dst in self.topo_map.edges:
            self.logger.info('{:^10s} {:^10s} {}'.format(str(src),
            str(dst),self.topo_map.edges[src,dst]['hop']))

        self.logger.info('\n')

    def show_topo_map_2(self):
        self.logger.info('topo map:')
        self.logger.info('{:^10s} -> {:^10s} {}'.format('node', 'node','delay
        for src, dst in self.topo_map.edges:
            self.logger.info('{:^10s} {:^10s} '.format(str(src),
            str(dst))+'%.1f'%self.topo_map.edges[src,dst]['delay']+'ms')
        self.logger.info('\n')
```

shortest_forward.py

```
# ryu-manager shortest_forward.py --observe-links
from ryu.base import app_manager
from ryu.base.app_manager import lookup_service_brick
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_D
from ryu.controller.handler import set_ev_cls
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, arp, ipv4
from ryu.lib.packet import ether_types
from ryu.controller import ofp_event
from ryu.topology import event
from ryu.topology.api import get_switch
import sys
from network_awareness1 import NetworkAwareness
import networkx as nx
ETHERNET = ethernet.ethernet.__name__

ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__
class ShortestForward(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'network_awareness1': NetworkAwareness}
```

```
    def __init__(self, *args, **kwargs):
        super(ShortestForward, self).__init__(*args, **kwargs)
        self.network_awareness1 = kwargs['network_awareness1']
        self.weight = 'delay'
        self.mac_to_port = {}
        self.sw = {}
        self.path=None
        self.switches = None
        self.ip_to_mac = {}
        self.mac_to_dpid = {}
        self.dpid_to_dp = {}
        self.ip_to_port = {}

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0, ha
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions
        mod = parser.OFPFlowMod(
        datapath=dp, priority=priority,
        idle_timeout=idle_timeout,
        hard_timeout=hard_timeout,
        match=match, instructions=inst)
        dp.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        dpid = dp.id
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        arp_pkt = pkt.get_protocol(arp.arp)
        ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
        pkt_type = eth_pkt.ethertype

        # layer 2 self-learning
        dst_mac = eth_pkt.dst
        src_mac = eth_pkt.src
```

```
        if isinstance(arp_pkt, arp.arp):
            self.handle_arp(msg, in_port, dst_mac,src_mac, pkt, pkt_type)

        if isinstance(ipv4_pkt, ipv4.ipv4):
            self.handle_ipv4(msg, ipv4_pkt.src, ipv4_pkt.dst, pkt_type)

    def handle_arp(self, msg, in_port, dst,src, pkt,pkt_type):
        datapath = msg.datapath
        dpid = datapath.id
        ofp = datapath.ofproto
        parser = datapath.ofproto_parser
        arp_pkg=pkt.get_protocol(arp.arp)
        if arp_pkg.opcode == arp.ARP_REQUEST:
            if arp_pkg.src_ip not in self.ip_to_mac:
                self.ip_to_mac[arp_pkg.src_ip]=src
                self.mac_to_dpid[src]=(dpid,in_port)
                self.ip_to_port[arp_pkg.src_ip]=(dpid,in_port)
            if arp_pkg.dst_ip in self.ip_to_mac:
                self.arpReply(datapath=datapath,port=in_port,src_mac=self.ip_
                dst_mac=src,src_ip=arp_pkg.dst_ip,dst_ip=arp_pkg.src_ip)
            else:
                out_port=ofp.OFPP_FLOOD
                actions=[parser.OFPActionOutput(out_port)]
                out = parser.OFPPacketOut(datapath = datapath,
                                          buffer_id = msg.buffer_id,
                                          in_port = in_port,
                                          actions = actions,
                                          data = msg.data)
                datapath.send_msg(out)
            return
        elif arp_pkg.opcode == arp.ARP_REPLY:
            if arp_pkg.src_ip not in self.ip_to_mac:
                self.ip_to_mac[arp_pkg.src_ip]=src
                self.mac_to_dpid[src]=(dpid,in_port)
                self.ip_to_port[arp_pkg.src_ip]=(dpid,in_port)
            dst_mac=self.ip_to_mac[arp_pkg.dst_ip]
            dst_dpid,dst_port=self.mac_to_dpid[dst_mac]
            switches = get_switch(self)

            for switch in switches:
                if dst_dpid == switch.dp.id:
                    self.arpReply(datapath=switch.dp,port=dst_port,
                    src_mac=src,dst_mac=dst_mac,
                    src_ip=arp_pkg.src_ip,dst_ip=arp_pkg.dst_ip)
                return
```

```
    def send_pkt(self,datapath,port,pkt):
        ofp=datapath.ofproto
        parser=datapath.ofproto_parser
        pkt.serialize()
        data=pkt.data
        actions=[parser.OFPActionOutput(port=port)]
        out=parser.OFPPacketOut(datapath=datapath,buffer_id=ofp.OFP_NO_BUFFEF
        in_port=ofp.OFPP_CONTROLLER,actions=actions,data=data)
        datapath.send_msg(out)

    def arpReply(self,datapath,port,src_mac,dst_mac,src_ip,dst_ip):
        pkt=packet.Packet()
        pkt.add_protocol(ethernet.ethernet(ethertype=0x0806,dst=dst_mac,src=s
        pkt.add_protocol(arp.arp(opcode=arp.ARP_REPLY,src_mac=src_mac,
        src_ip=src_ip,dst_mac=dst_mac,dst_ip=dst_ip))
        self.send_pkt(datapath,port,pkt)

    def handle_ipv4(self, msg, src_ip, dst_ip, pkt_type):
        parser = msg.datapath.ofproto_parser

        dpid_path = self.network_awareness1.shortest_path(src_ip, dst_ip,weig
        if not dpid_path:
            return

        self.path=dpid_path
        # get port path: h1 -> in_port, s1, out_port -> h2
        port_path = []
        for i in range(1, len(dpid_path) - 1):
            in_port = self.network_awareness1.link_info[(dpid_path[i], dpid_p
            out_port = self.network_awareness1.link_info[(dpid_path[i], dpid_
            port_path.append((in_port, dpid_path[i], out_port))
        self.show_path(src_ip, dst_ip, port_path)
        # calc path delay

        # send flow mod
        for node in port_path:
            in_port, dpid, out_port = node
            self.send_flow_mod(parser, dpid, pkt_type, src_ip, dst_ip, in_por
            self.send_flow_mod(parser, dpid, pkt_type, dst_ip, src_ip, out_pc

        # send packet_out
        _, dpid, out_port = port_path[-1]
        dp = self.network_awareness1.switch_info[dpid]
        actions = [parser.OFPActionOutput(out_port)]
        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id, in_pc
```

```
        dp.send_msg(out)

    def send_flow_mod(self, parser, dpid, pkt_type, src_ip, dst_ip, in_port,
        dp = self.network_awareness1.switch_info[dpid]
        match = parser.OFPMatch(
        in_port=in_port, eth_type=pkt_type, ipv4_src=src_ip, ipv4_dst=dst_ip)
        actions = [parser.OFPActionOutput(out_port)]
        self.add_flow(dp, 1, match, actions, 10, 30)

    def show_path(self, src, dst, port_path):
        self.logger.info('path: {} -> {}'.format(src, dst))
        path_delay = 0
        path = src + ' -> '
        for i in range(len(port_path)):
            path += '{}:s{}:{}'.format(port_path[i][0],port_path[i][1],port_p

            if i == len(port_path) - 1:
                break
            path_delay += self.network_awareness1.delay[(port_path[i][1],port
        '''
        for node in port_path:
        path += '{}:s{}:{}'.format(*node) + ' -> '
        '''
        path += dst
        path += " delay=" +'%.1f'%path_delay+ 'ms'
        self.logger.info(path)
```