SDN Experiment 3

实验环境

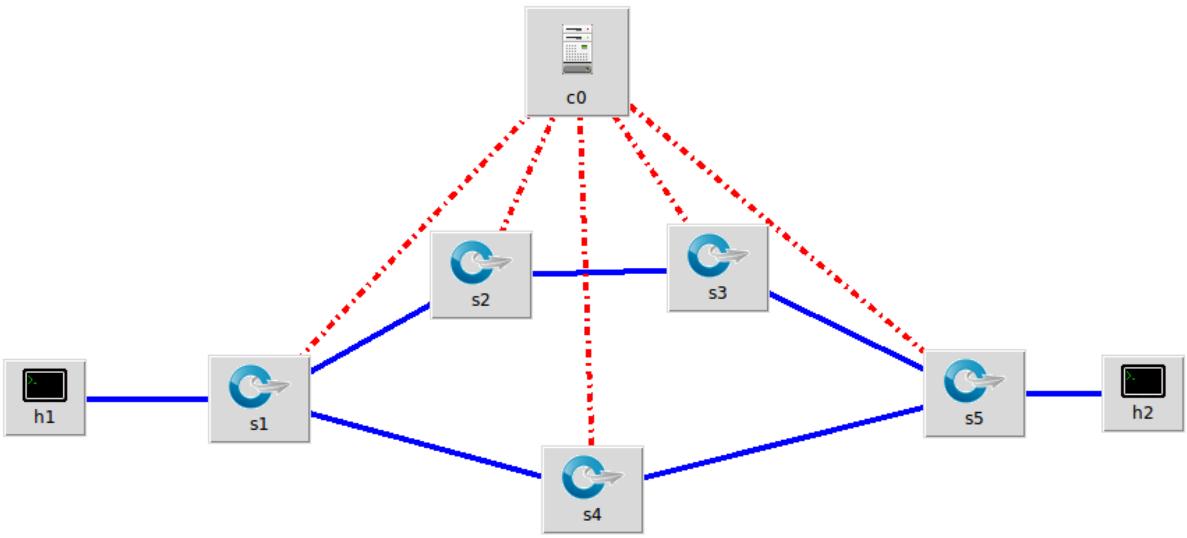
与第一次实验相同

实验内容

第三次实验主要为设计性实验，要求各位在熟悉SDN的基本原理和RYU API的基础上解决下面问题

题目

部署一个如下图所示的网络环境，在此拓扑基础上完成动态转发规则的改变和链路故障恢复功能



实验1 动态改变转发规则

在上图所示的拓扑结构中，h1到h2有两条通路，所谓动态地改变转发规则，就是要让h1到h2的包在两条路径上交替转发，具体描述如下：

假设h1 ping h2，初始的路由规则为s1-s4-s5，5秒后，路由转发规则变为s1-s2-s3-s5，再过5秒后，转发规则又回到最初的s1-s4-s5，通过这个循环调度的例子动态的改变交换机的转发规则

思路

- idle_timeout 与 hard_timeout

在`/sdn/ryu/ryu/ofproto/ofproto_v1_3_parser.py`中，描述了`OFPFlowMod`的定义以及使用方法的样例：

```
    """
```

```
    Example:

        def send_flow_mod(self, datapath):
            ofp = datapath.ofproto
            ofp_parser = datapath.ofproto_parser

            cookie = cookie_mask = 0
            table_id = 0
            idle_timeout = hard_timeout = 0
            priority = 32768
            buffer_id = ofp.OFP_NO_BUFFER
            match = ofp_parser.OFPMatch(in_port=1, eth_dst='ff:ff:ff:ff:ff:ff')
            actions = [ofp_parser.OFPActionOutput(ofp.OFPP_NORMAL, 0)]
            inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                                     actions)]
            req = ofp_parser.OFPFlowMod(datapath, cookie, cookie_mask,
                                        table_id, ofp.OFPFC_ADD,
                                        idle_timeout, hard_timeout,
                                        priority, buffer_id,
                                        ofp.OFPP_ANY, ofp.OFPG_ANY,
                                        ofp.OFPFF_SEND_FLOW_REM,
                                        match, inst)
            datapath.send_msg(req)
    """
```

idle_timeout称为空闲超时，指在指定时间之内若流表没有匹配任何报文则将此流表删除

hard_timeout称为硬超时，指自流表下发之后开始，经过指定时间之后无条件将流表删除

两个时间的单位都是秒，当其值设为0时则表示不对流表采取超时限制，除非控制器发出 `OFPFC_DELETE` 请求流表修改，否则交换机不会主动移除流表项

本实验可利用hard_timeout完成转发规则的改变

- 拓扑结构的存储以及路径的选择

本次实验提供`dynamic_rules.py`代码文件，在函数`get_topology()`中，记录了存储拓扑图所用的数据结构与记录流程：

```
    @set_ev_cls([event.EventSwitchEnter, event.EventSwitchLeave,
event.EventPortAdd, event.EventPortDelete,
        event.EventPortModify, event.EventLinkAdd, event.EventLinkDelete])
    def get_topology(self, ev):
        links_list = get_link(self.topology_api_app, None)
        self.src_links.clear()
        for link in links_list:
            sw_src = link.src.dpid
            sw_dst = link.dst.dpid
            src_port = link.src.port_no
            dst_port = link.dst.port_no
            src_port_name = link.src.name
```

```
            dst_port_name = link.dst.name
            self.port_name_to_num[src_port_name] = src_port
            self.port_name_to_num[dst_port_name] = dst_port
            self.src_links[sw_src][(sw_src, sw_dst)] = (src_port, dst_port)
            self.src_links[sw_dst][(sw_dst, sw_src)] = (dst_port, src_port)
            # self.logger.info("****src_port_name : %s", str(src_port_name))
            # self.logger.info("src_links : %s", str(self.src_links))
            # self.logger.info("port_name_to_num : %s",
str(self.port_name_to_num))
```

本次实验提供`dynamic_rules.py`代码文件，在函数`short_path()`中，记录了遍历拓扑图与筛选路径的方法：

```
    def short_path(self, src, dst, bw=0):
        if src == dst:
            return []
        result = defaultdict(lambda: defaultdict(lambda: None))
        distance = defaultdict(lambda: defaultdict(lambda: None))

        # the node is checked
        seen = [src]

        # the distance to src
        distance[src] = 0

        w = 1  # weight
        while len(seen) < len(self.src_links):
            node = seen[-1]
            if node == dst:
                break
            for (temp_src, temp_dst) in self.src_links[node]:
                if temp_dst not in seen:
                    temp_src_port = self.src_links[node][(temp_src, temp_dst)]
[0]
                    temp_dst_port = self.src_links[node][(temp_src, temp_dst)]
[1]
                    if (distance[temp_dst] is None) or (distance[temp_dst] >
distance[temp_src] + w):
                        distance[temp_dst] = distance[temp_src] + w
                        # result = {"dpid":(link_src, src_port, link_dst,
dst_port)}
                        result[temp_dst] = (temp_src, temp_src_port, temp_dst,
temp_dst_port)
            min_node = None
            min_path = 999
            # get the min_path node
            for temp_node in distance:
                if (temp_node not in seen) and (distance[temp_node] is not
None):
```

```
                    if distance[temp_node] < min_path:
                        min_node = temp_node
                        min_path = distance[temp_node]
            if min_node is None:
                break
            seen.append(min_node)

        path = []

        if dst not in result:
            return None

        while (dst in result) and (result[dst] is not None):
            path = [result[dst][2:4]] + path
            path = [result[dst][0:2]] + path
            dst = result[dst][0]
        #self.logger.info("path : %s", str(path))
        return path
```

由于本次实验拓扑简单，可以修改最短路计算函数为最长路计算函数，以获取另一条路径；也可以自己编写函数，求出任意点对之间的所有路径

- PacketIn消息与install_path

算法的基本思想应为利用hard_timeout使交换机自动删除流表，在下一次`packet_in`消息来临之后选择另外一条路径进行安装

本次实验提供`dynamic_rules.py`代码文件，关于路径的选择及流表的下发请在阅读`_packet_in_handler()`和`install_path()`函数后自行设计和实现

特别提示： 在`__init__()`函数中定义了两个全局变量path_mod和path以方便算法设计与实现

实验2 链路故障恢复功能

在上图所示的拓扑结构中，h1到h2有两条通路，若其中正在进行传输的路径因为发生故障而断开连接，系统应当及时作出反应，改变转发规则到另外一条路径上；若故障修复，系统也应当即时作出反应，改变转发规则到优先级较高的路径上

假设h1 ping h2，首选的路由规则为s1-s4-s5，由于故障，s1-s4之间的链路被断开，系统应当将转发规则改变为备选路径s1-s2-s3-s5，若此时s1-s4之间的故障被修复，链路恢复连接，系统应该将路径重新确定为首选路径s1-s4-s5

本实验中，路径的优先级由链路上的跳数决定，跳数越少的优先级越高

思路

在实验1的基础上，改变`install_path()`使其始终获取`short_path()`的结果，以达到始终选取当前拓扑中最短路径的效果

修改下发流表时的hard_timeout值，使其为0

- 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

```
    def add_flow(self, datapath, priority, match, actions, buffer_id=None,
idle_timeout=0, hard_timeout=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,
                                    priority=priority, match=match,
                                    idle_timeout=idle_timeout,
                                    hard_timeout=hard_timeout,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    idle_timeout=idle_timeout,
                                    hard_timeout=hard_timeout,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)
```

- get_OFPPortStatus_msg函数与EventOFPPortStatus消息

在链路发生改变时，端口信息会有所变化，此时会抛出端口状态改变的事件，即`EventOFPPortStatus`，通过将此事件与处理函数`get_OFPPortStatus_msg`绑定在一起，就可以获取状态改变的信息以及编写相应的处理函数

ryu自带的`EventOFPPortStatus`事件处理函数位于`/sdn/ryu/ryu/controller/ofp_handler.py`中，部分代码截取如下：

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

        if msg.reason in [ofproto.OFPPR_ADD, ofproto.OFPPR_MODIFY]:
            datapath.ports[msg.desc.port_no] = msg.desc
        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.reason, msg.desc.port_no),
            datapath.state)
```

- PacketIn消息的合理利用

本实验算法的基本思路应当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，以便下一次交换机向控制器发送`packet_in`消息，从而获取全新的路径

在实验1的基础上，应当完成`get_OFPPortStatus_msg()`与`delete_flow()`函数的编写，实现在链路改变时的流表删除，以便接受`packet_in`消息

特别提示： 在`__init__()`函数中定义了全局变量path以方便算法设计与实现

关于细节

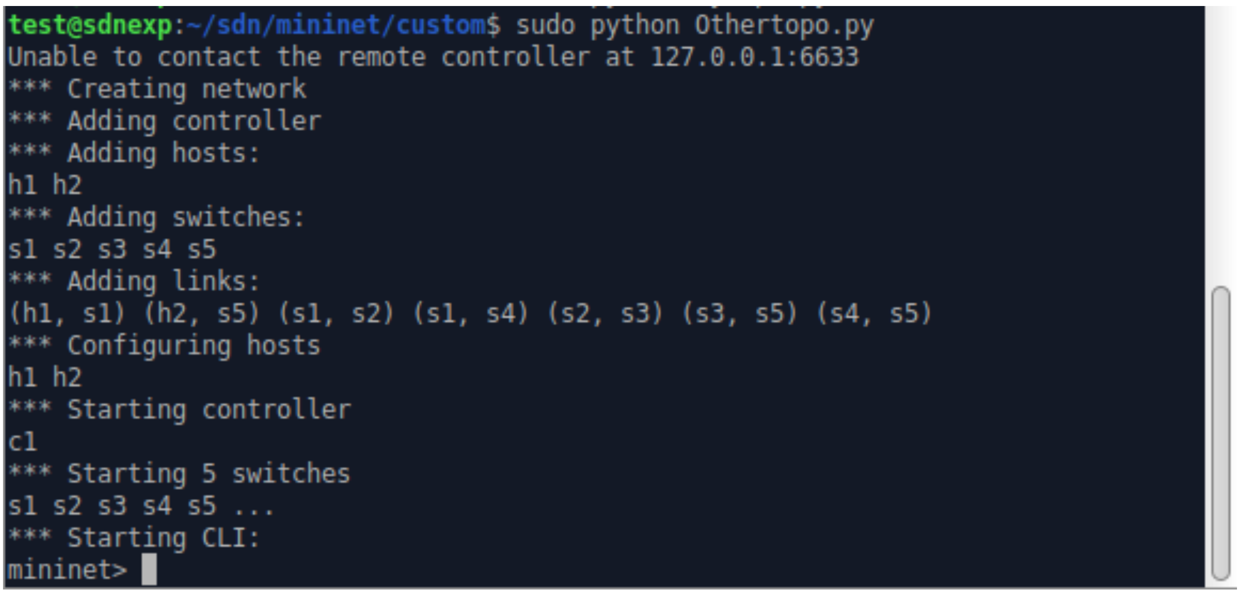
- 本实验提供了两个文件`dynamic_rules.py`和`network_monitor.py`，后者是前者在运行时所需要用到的文件，两者须放在同一文件夹下，建议放于`/sdn/ryu/ryu/app/`目录下
- `dynamic_rules.py`用于实现本次两个小实验，初始代码只实现了最短路的计算，需要修改和补充以完成实验要求，具体实现细节请根据本实验参考书前文**思路**部分自行设计
- 在实验1时可以将`get_OFPPortStatus_msg()`与`delete_flow()`两个函数的定义部分注释掉，否则运行时会出错
- 在实验2时务必在添加流表时将hard_timeout修改为0，否则达不到链路恢复的效果
- mininet中链路的控制可以采用命令link down和link up来控制

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

- 实验时，利用python启动加载自定义拓扑文件的mininet

```
$ sudo python Mytopo.py
```

启动mininet：



- 实验时，利用如下命令启动`dynamic_rules.py`

```
$ ryu-manager dynamic_rules.py --ofp-tcp-listen-port 6633 --observe-links
```

启动ryu：

```
Terminal - test@sdnexp: ~/sdn/ryu/ryu/app
test@sdnexp:~/sdn/ryu/ryu/app$ ryu-manager dynamic_rules.py --ofp-tcp-listen-port 6633 --observe-links
loading app dynamic_rules.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app None of Network_Monitor
creating context Network_Monitor
instantiating app dynamic_rules.py of dynamic_rules
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
(3897) wsgi starting up on http://0.0.0.0:8080
```

- 运行实验时，用`xterm h1`打开h1终端，使其不停地ping h2，在实验1中，应当能看到路径随时间有规律地变化：

```
^Ctest@sdnexp:~/sdn/ryu/ryu/app$ ryu-manager dynamic_rules.py --ofp-tcp-listen-port 6633 --observe-links
loading app dynamic_rules.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app None of Network_Monitor
creating context Network_Monitor
instantiating app dynamic_rules.py of dynamic_rules
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
(3931) wsgi starting up on http://0.0.0.0:8080
path : [(1, 1), (1, 3), (4, 1), (4, 2), (5, 3), (5, 1)]
path : [(5, 1), (5, 3), (4, 2), (4, 1), (1, 3), (1, 1)]
path : [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2), (5, 2), (5, 1)]
path : [(5, 1), (5, 2), (3, 2), (3, 1), (2, 2), (2, 1), (1, 2), (1, 1)]
path : [(1, 1), (1, 3), (4, 1), (4, 2), (5, 3), (5, 1)]
path : [(5, 1), (5, 3), (4, 2), (4, 1), (1, 3), (1, 1)]
path : [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2), (5, 2), (5, 1)]
path : [(5, 1), (5, 2), (3, 2), (3, 1), (2, 2), (2, 1), (1, 2), (1, 1)]
path : [(1, 1), (1, 3), (4, 1), (4, 2), (5, 3), (5, 1)]
path : [(5, 1), (5, 3), (4, 2), (4, 1), (1, 3), (1, 1)]
```

- 运行实验时，用`xterm h1`打开h1终端，使其不停地ping h2，在实验2中，link down 之前应当能看到路径是稳定的：

Terminal - test@sdnexp: ~/sdn/ryu/ryu/app

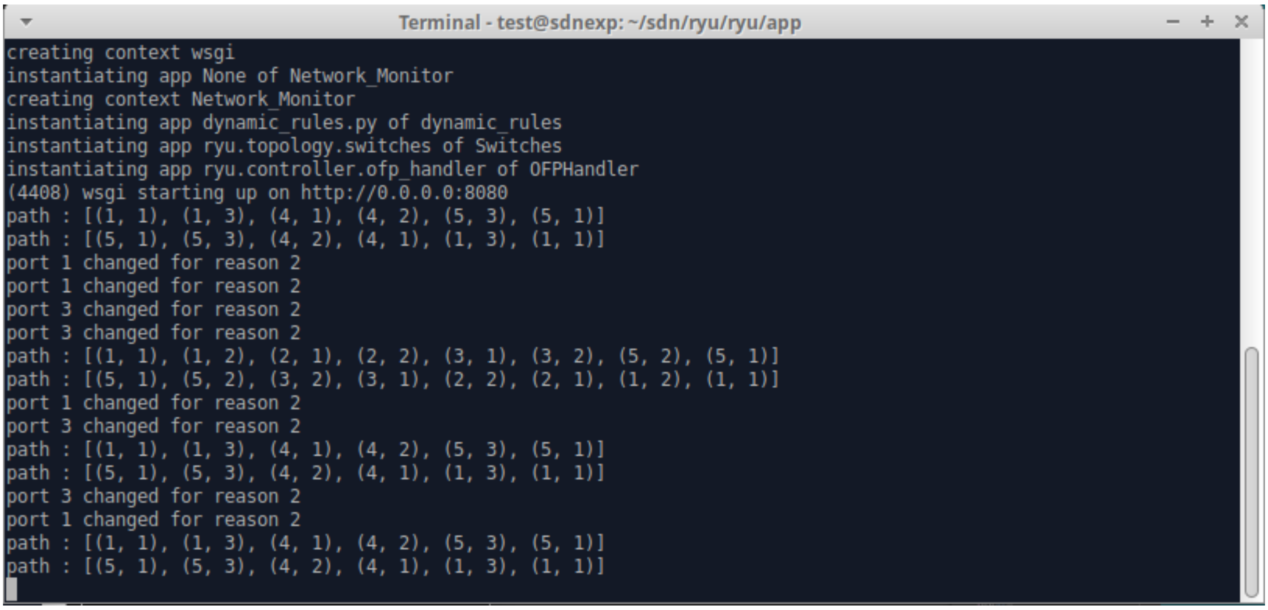
```
test@sdnexp:~/sdn/ryu/ryu/app$ ryu-manager dynamic_rules.py --ofp-tcp-listen-port 6633 --observe-links
loading app dynamic_rules.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app None of Network_Monitor
creating context Network_Monitor
instantiating app dynamic_rules.py of dynamic_rules
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
(4408) wsgi starting up on http://0.0.0.0:8080
path : [(1, 1), (1, 3), (4, 1), (4, 2), (5, 3), (5, 1)]
path : [(5, 1), (5, 3), (4, 2), (4, 1), (1, 3), (1, 1)]
```

- 运行实验时，用`xterm h1`打开h1终端，使其不停地ping h2，在实验2中，link down 之后应当能看到路径迅速发生变化：

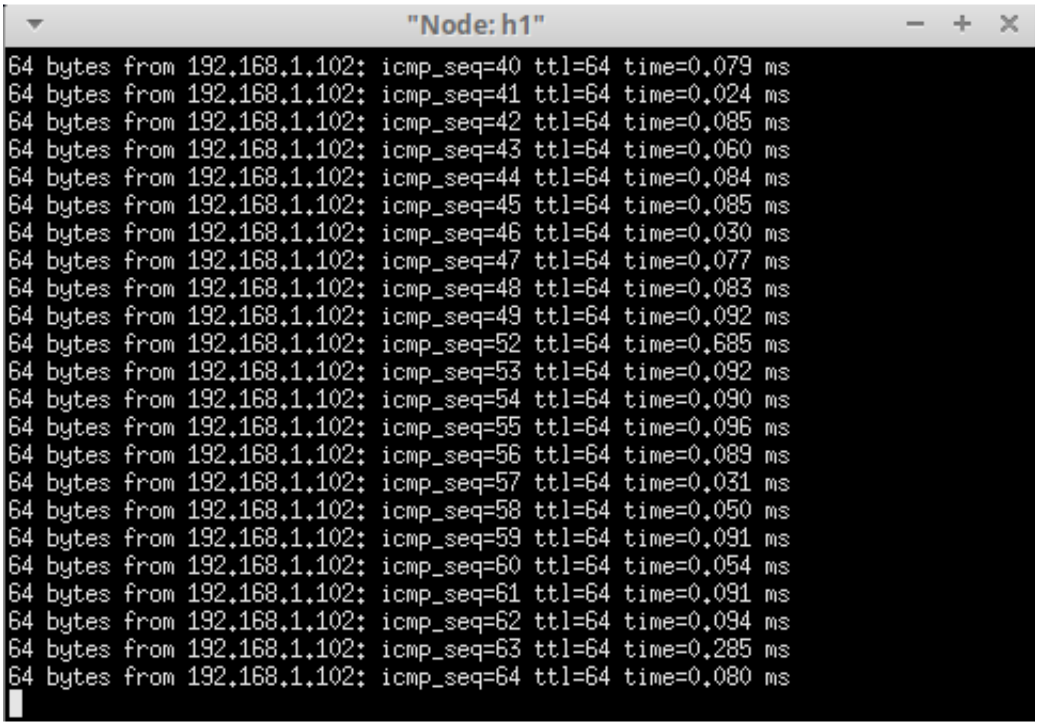
Terminal - test@sdnexp: ~/sdn/ryu/ryu/app

```
test@sdnexp:~/sdn/ryu/ryu/app$ ryu-manager dynamic_rules.py --ofp-tcp-listen-port 6633 --observe-links
loading app dynamic_rules.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app None of Network_Monitor
creating context Network_Monitor
instantiating app dynamic_rules.py of dynamic_rules
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
(4408) wsgi starting up on http://0.0.0.0:8080
path : [(1, 1), (1, 3), (4, 1), (4, 2), (5, 3), (5, 1)]
path : [(5, 1), (5, 3), (4, 2), (4, 1), (1, 3), (1, 1)]
port 1 changed for reason 2
port 1 changed for reason 2
port 3 changed for reason 2
port 3 changed for reason 2
path : [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2), (5, 2), (5, 1)]
path : [(5, 1), (5, 2), (3, 2), (3, 1), (2, 2), (2, 1), (1, 2), (1, 1)]
```

- 运行实验时，用`xterm h1`打开h1终端，使其不停地ping h2，在实验2中，link up 之后应当能看到路径迅速恢复成初始状态：



- 运行实验时，用`xterm h1`打开h1终端，使其不停地ping h2，在实验2中，无论执行link down或者是link up之后，ping的界面应当是不停顿地执行（但会有延迟，为什么？）：



- 实验过程中如需查看流表，可另起一个终端，执行下面的命令即可查看

```
$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```