

# CS339 计算机网络

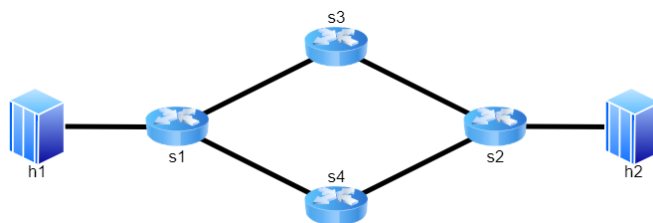
## Lab 6: RYU Open Flow Controller

### 目录

- 1 Problem1
- 2 Problem2
  - 2.1 实现
  - 2.2 运行结果
- 3 Problem3
  - 3.1 实现
  - 3.2 运行结果
- 4 Problem4
  - 4.1 实现
  - 4.2 运行结果

## 1 Problem1

使用mininet构造网络，网络结构结构如下：



network.py 中网络结构如下：

```
1 switch1 = net.addSwitch('s1')
2 switch2 = net.addSwitch('s2')
3 switch3 = net.addSwitch('s3')
4 switch4 = net.addSwitch('s4')
5 host1 = net.addHost('h1', cpu=.25, mac='00:00:00:00:00:01')
6 host2 = net.addHost('h2', cpu=.25, mac='00:00:00:00:00:02')
7 net.addLink(host1, switch1, bw=10, delay='5ms', loss=0, use_htb=True)
8 net.addLink(host2, switch2, bw=10, delay='5ms', loss=0, use_htb=True)
9 net.addLink(switch1, switch3, bw=10, delay='5ms', loss=0, use_htb=True)
10 net.addLink(switch1, switch4, bw=10, delay='5ms', loss=0, use_htb=True)
11 net.addLink(switch2, switch3, bw=10, delay='5ms', loss=0, use_htb=True)
12 net.addLink(switch2, switch4, bw=10, delay='5ms', loss=0, use_htb=True)
13 c1 = net.addController('c1', controller=RemoteController, ip="127.0.0.1",
    port=6653)
14 net.build()
15 c1.start()
```

```

16 s1, s2, s3, s4 = net.getNodeByName('s1', 's2', 's3', 's4')
17 s1.start([c1])
18 s2.start([c1])
19 s3.start([c1])
20 s4.start([c1])
21 net.start()

```

## 2 Problem2

### 2.1 实现

为了实现每5秒切换一次路由表，我们需要在控制器创建后台线程，该线程每隔5秒钟向指定的路由器发送路由表更新命令。参考示例代码 `simple_monitor_13.py`，可以使用 `ryu.lib.hub` 实现。

在本题中，首先监听事件 `ofp_event.EventOFPStateChange`，用以保存当前在线的所有节点信息。

```

1 @set_ev_cls(ofp_event.EventOFPStateChange,
2             [MAIN_DISPATCHER, DEAD_DISPATCHER])
3 def _state_change_handler(self, ev):
4     datapath = ev.datapath
5     if ev.state == MAIN_DISPATCHER:
6         if datapath.id not in self.datapaths:
7             self.logger.debug('register datapath: %016x', datapath.id)
8             self.datapaths[datapath.id] = datapath
9     elif ev.state == DEAD_DISPATCHER:
10        if datapath.id in self.datapaths:
11            self.logger.debug('unregister datapath: %016x', datapath.id)
12            del self.datapaths[datapath.id]

```

同时，在控制器启动时创建 `hub`，后台运行下面的函数以定时切换路由表。

```

1 def _monitor(self):
2     out = 0
3     while True:
4         try:
5             datapath1, datapath2 = self.datapaths[1], self.datapaths[2]
6         except KeyError:
7             pass
8         else:
9             self.change_route(datapath1, out%2+2)
10            self.change_route(datapath2, out%2+2)
11            out += 1
12            hub.sleep(5)

```

完整的实现代码见 `hw2.py`。

## 2.2 运行结果

运行结果如下图，可以看出 s1 中对应的路由规则每5秒进行一次转变，且转变过程中不会丢失数据分组。

```
64 bytes from 10.0.0.2: icmp_seq=27 ttl=64 time=47.3 ms
64 bytes from 10.0.0.2: icmp_seq=28 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=29 ttl=64 time=45.6 ms
64 bytes from 10.0.0.2: icmp_seq=30 ttl=64 time=46.9 ms
64 bytes from 10.0.0.2: icmp_seq=31 ttl=64 time=47.0 ms
64 bytes from 10.0.0.2: icmp_seq=32 ttl=64 time=47.0 ms
64 bytes from 10.0.0.2: icmp_seq=33 ttl=64 time=44.8 ms
64 bytes from 10.0.0.2: icmp_seq=34 ttl=64 time=45.4 ms
64 bytes from 10.0.0.2: icmp_seq=35 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=36 ttl=64 time=46.7 ms
64 bytes from 10.0.0.2: icmp_seq=37 ttl=64 time=46.9 ms
64 bytes from 10.0.0.2: icmp_seq=38 ttl=64 time=45.0 ms
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=45.9 ms
64 bytes from 10.0.0.2: icmp_seq=41 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=42 ttl=64 time=47.0 ms
64 bytes from 10.0.0.2: icmp_seq=43 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=44 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=45 ttl=64 time=46.9 ms
64 bytes from 10.0.0.2: icmp_seq=46 ttl=64 time=46.9 ms

mcx@mcx-virtual-machine:~/code/CN$ sudo ovs-ofctl -O openflow13 d
ump-flows s1
cookie=0x0, duration=118.630s, table=0, n_packets=72, n_bytes=80
98, priority=1,in_port="s1-eth2" actions=output:"s1-eth1"
cookie=0x0, duration=118.630s, table=0, n_packets=71, n_bytes=80
68, priority=1 in port="s1-eth3" actions=output:"s1-eth1"
cookie=0x0, duration=4.874s, table=0, n_packets=50, n_bytes=4564
, priority=2,in_port="s1-eth1" actions=output:"s1-eth2"
cookie=0x0, duration=118.630s, table=0, n_packets=4, n_bytes=330
, priority=0 actions=CONTROLLER:65535
mcx@mcx-virtual-machine:~/code/CN$ sudo ovs-ofctl -O openflow13 d
ump-flows s1
cookie=0x0, duration=120.667s, table=0, n_packets=72, n_bytes=80
98, priority=1,in_port="s1-eth2" actions=output:"s1-eth1"
cookie=0x0, duration=120.667s, table=0, n_packets=74, n_bytes=83
76, priority=1 in port="s1-eth3" actions=output:"s1-eth1"
cookie=0x0, duration=1.910s, table=0, n_packets=52, n_bytes=4760
, priority=2,in_port="s1-eth1" actions=output:"s1-eth3"
cookie=0x0, duration=120.667s, table=0, n_packets=4, n_bytes=330
, priority=0 actions=CONTROLLER:65535
mcx@mcx-virtual-machine:~/code/CN$
```

## 3 Problem3

### 3.1 实现

通过在交换机的流表中添加 OFPGT\_SELECT 类型的 group 实现分流，配置代码如下。值得注意的是，对于 OFPGT\_SELECT 类型的 group，并不需要指定 watch\_port、watch\_group 参数。

```
1 actions1 = [parser.OFPActionOutput(2)]
2 actions2 = [parser.OFPActionOutput(3)]
3 match = parser.OFPMatch(in_port=1)
4 buckets = [parser.OFPBucket(weight=50,actions=actions1),
5             parser.OFPBucket(weight=50,actions=actions2)]
6 group_id = datapath.id
7 req = parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD,ofproto.OFPGT_SELECT, group_id,
8                             buckets)
9 datapath.send_msg(req)
```

完整的实现代码见 hw3.py。

### 3.2 运行结果

使用 ovs-ofctl -O OpenFlow13 dump-group-stats s1 查看 s1 上 group 的状态，如下图所示。可以看出，从 h1 进入 s1 的分组被转发到两条链路。

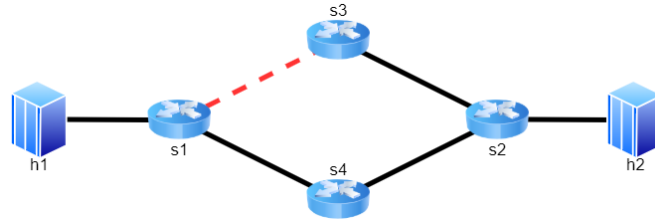
```
mcx@mcx-virtual-machine:~/code/CN$ sudo ovs-ofctl -O OpenFlow13 d
ump-group-stats s1
OFPST_GROUP reply (OF1.3) (xid=0x6):
  group_id=1,duration=615.177s,ref_count=1,packet_count=24,byte_co
unt=2000,bucket0:packet_count=10,byte_count=644,bucket1:packet_co
unt=14,byte_count=1356
```

## 4 Problem4

### 4.1 实现

与上一题类似，在交换机的流表中添加 OFPGT\_FF 类型的 group 实现快速恢复。OFPGT\_FF 类型需要传入 watch\_port、watch\_group 参数以指定监控的端口或组，在本例中，我们只需要监控端口状况，因此只传入 watch\_port 参数，watch\_group 将采用缺省值 OFPG\_ANY。

但是这样的配置是无法正常工作的。考虑下图的情况，当 s1-s3 链路中断时，s1 检测到错误，h1 发出的数据通过 s4 转发给 h2，但是 s2 无法检测到错误，因此 s2 会将 h2 返回的数据转发到故障线路上，h1 与 h2 依然无法正常通信。



解决问题的办法是为 s3 也添加一个添加 OFPGT\_FF 类型的 group，当 s1-s3 链路故障时，将 s2 发送的数据重新发送回 s2，s2 收到后会向 s4 转发回流的分组。在实际实现中，我们通过匹配 in\_port 和 eth\_src 字段，让 s2 识别回流数据包。需要指出的是，这种策略会导致数据分组会在 s2、s3 之间绕行，降低网络性能。

这个问题的解决方案是使用控制器介入，交换机会向控制器汇报连接状态改变的事件，控制器再向对方交换机流表中添加一条路由规则，使得分组向正常工作的链路转发，待故障恢复后再删除额外的路由规则。在上面的情况中，s1、s3 检测到连接故障，首先会触发 OFPGT\_FF 路由规则，同时也会向控制器汇报连接状态改变；控制器了解到 s1-s3 连接故障之后，会在 s2 中添加一条向 s4 转发分组的高优先级路由规则，使得 h2 发送的数据通过 s4 传输。

这一部分控制器代码如下所示。

```

1  @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
2  def _port_state_change_handler(self, ev):
3      datapath = ev.msg.datapath
4      parser = datapath.ofproto_parser
5      ofproto = datapath.ofproto
6      desc = ev.msg.desc
7      if desc.state & ofproto.OFPPS_LIVE:
8          if datapath.id in (1,2) and desc.port_no in (2,3) and self.failover:
9              self.logger.warning(f"datapath {datapath.id} port {desc.port_no}
UP!")
10             if datapath.id == 1:
11                 datapath_ = self.datapaths[2]
12             else:
13                 datapath_ = self.datapaths[1]
14             out_port = 3 if desc.port_no == 2 else 2
15             match = parser.OFPMatch(in_port=1)
16             self.del_flow(datapath_, match, out_port)
17             self.logger.info(f"del flow: datapath {datapath.id}
in_port:1,out_port:{out_port}")
18             self.failover = False
19         else:
20             if datapath.id in (1,2) and desc.port_no in (2,3) and not self.failover:
21                 self.logger.warning(f"datapath {datapath.id} port {desc.port_no}
DOWN!")
22             if datapath.id == 1:
23                 datapath_ = self.datapaths[2]

```

```

24         else:
25             datapath_ = self.datapaths[1]
26             out_port = 3 if desc.port_no == 2 else 2
27             match = parser.OFPMatch(in_port=1)
28             actions = [parser.OFPACTIONOutput(out_port)]
29             self.add_flow(datapath_, 10, match, actions)
30             self.logger.info(f"add flow: datapath {datapath.id}
in_port:1,out_port:{out_port}")
31             self.failover = True

```

其中，del\_flow 函数实现如下：

```

1 def del_flow(self, datapath, match, out_port):
2     ofproto = datapath.ofproto
3     parser = datapath.ofproto_parser
4
5     mod = parser.OFPFlowMod(datapath=datapath, match=match,
6                             command=ofproto.OFPFC_DELETE, out_port=out_port,
7                             out_group=ofproto.OFPG_ANY, flags=ofproto.OFPFF_SEND_FLOW_REM)
8     datapath.send_msg(mod)

```

值得一提的是，对路由规则的删除相较于添加有更高的匹配要求，即使目标规则设置时没有指定 out\_group，在删除时也必须将 out\_group 指定为 OFPG\_ANY，否则无法匹配到目标规则。

完整的实现代码见 hw4.py。

## 4.2 运行结果

启动控制器与mininet，使用在mininet的终端中使用 h1 ping h2 测试连接，过程中在另一个终端使用 ifconfig 命令改变个连接的状态，观察 h1、h2 的连接状态。

```

终端 端口 问题 1 输出 调试控制台
64 bytes from 10.0.0.2: icmp_seq=39 ttl=64 time=45.9 ms
64 bytes from 10.0.0.2: icmp_seq=40 ttl=64 time=43.9 ms
64 bytes from 10.0.0.2: icmp_seq=41 ttl=64 time=44.8 ms
64 bytes from 10.0.0.2: icmp_seq=42 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=43 ttl=64 time=44.9 ms
64 bytes from 10.0.0.2: icmp_seq=44 ttl=64 time=45.8 ms
64 bytes from 10.0.0.2: icmp_seq=45 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=46 ttl=64 time=45.0 ms
64 bytes from 10.0.0.2: icmp_seq=47 ttl=64 time=44.5 ms
64 bytes from 10.0.0.2: icmp_seq=48 ttl=64 time=43.9 ms
64 bytes from 10.0.0.2: icmp_seq=49 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=50 ttl=64 time=47.9 ms
64 bytes from 10.0.0.2: icmp_seq=51 ttl=64 time=45.9 ms
64 bytes from 10.0.0.2: icmp_seq=52 ttl=64 time=44.5 ms
64 bytes from 10.0.0.2: icmp_seq=53 ttl=64 time=46.8 ms
64 bytes from 10.0.0.2: icmp_seq=54 ttl=64 time=44.7 ms
64 bytes from 10.0.0.2: icmp_seq=55 ttl=64 time=45.9 ms

mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s1-eth3 down
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s1-eth3 up
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s1-eth2 down
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s1-eth2 up
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s2-eth2 down
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s2-eth2 up
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s2-eth3 down
mcx@mcx-virtual-machine:~/code/CN$ sudo ifconfig s2-eth3 up
mcx@mcx-virtual-machine:~/code/CN$

64 bytes from 10.0.0.2: icmp_seq=59 ttl=64 time=45.8 ms
64 bytes from 10.0.0.2: icmp_seq=60 ttl=64 time=46.1 ms
64 bytes from 10.0.0.2: icmp_seq=61 ttl=64 time=45.1 ms
^C
--- 10.0.0.2 ping statistics ---
61 packets transmitted, 61 received, 0% packet loss, time 60109ms
rtt min/avg/max/mdev = 43.529/45.322/47.906/1.088 ms
mininet>

```

从运行结果可以看出，对于 s1、s2 之间任意一条连接出现故障，我们的控制器均可以正确调节网络使得 h1、h2 之间的通信不出现丢包。

