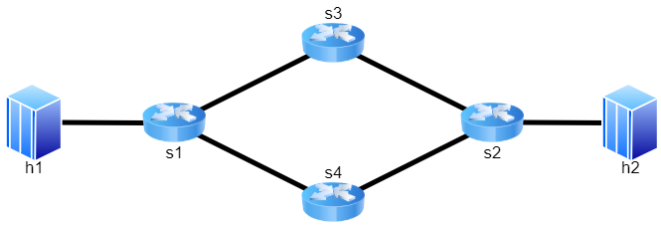
CS339 计算机网络
Lab 6: RYU Open Flow Controller

目录

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：

```
switch1 = net.addSwitch('s1')
switch2 = net.addSwitch('s2')
switch3 = net.addSwitch('s3')
switch4 = net.addSwitch('s4')
host1 = net.addHost('h1', cpu=.25, mac='00:00:00:00:00:01')
host2 = net.addHost('h2', cpu=.25, mac='00:00:00:00:00:02')
net.addLink(host1, switch1, bw=10, delay='5ms', loss=0, use_htb=True)
net.addLink(host2, switch2, bw=10, delay='5ms', loss=0, use_htb=True)
net.addLink(switch1, switch3, bw=10, delay='5ms', loss=0, use_htb=True)
net.addLink(switch1, switch4, bw=10, delay='5ms', loss=0, use_htb=True)
net.addLink(switch2, switch3, bw=10, delay='5ms', loss=0, use_htb=True)
net.addLink(switch2, switch4, bw=10, delay='5ms', loss=0, use_htb=True)
c1 = net.addController('c1', controller=RemoteController, ip="127.0.0.1",
port=6653)
net.build()
c1.start()
```

```
s1, s2, s3, s4 = net.getNodeByName('s1', 's2', 's3', 's4')
s1.start([c1])
s2.start([c1])
s3.start([c1])
s4.start([c1])
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。

```
@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 datapath.id not 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]
```

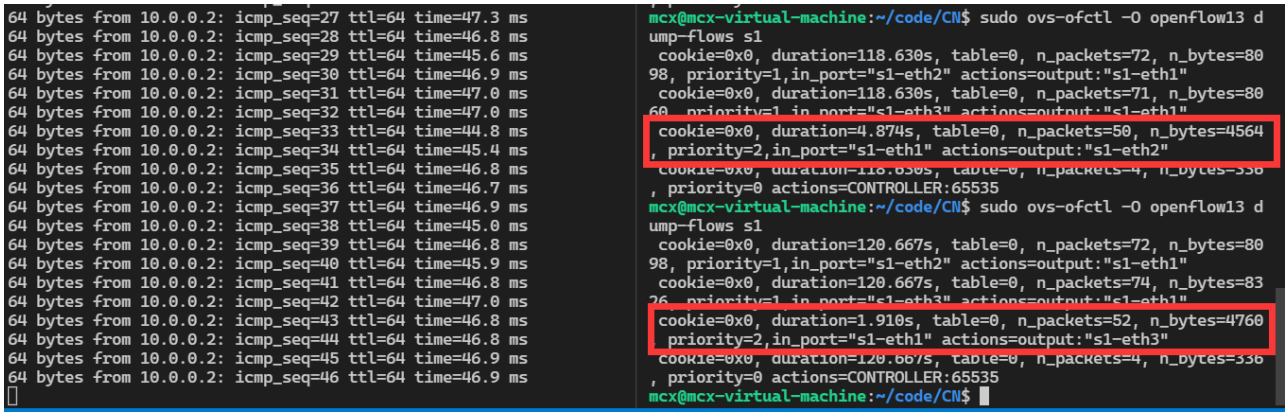
同时，在控制器启动时创建 `hub` ，后台运行下面的函数以定时切换路由表。

```
def _monitor(self):
    out = 0
    while True:
        try:
            datapath1,datapath2 = self.datapaths[1],self.datapaths[2]
        except KeyError:
            pass
        else:
            self.change_route(datapath1,out%2+2)
            self.change_route(datapath2,out%2+2)
            out += 1
        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。



3 Problem3

3.1 实现

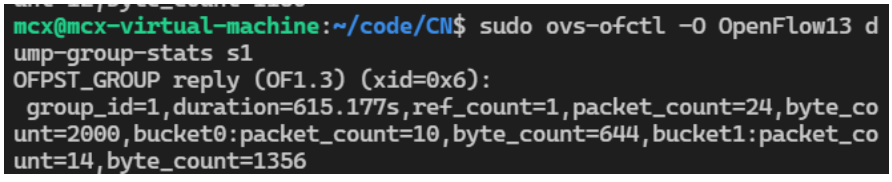
通过在交换机的流表中添加 OFPGT_SELECT 类型的 group 实现分流，配置代码如下。值得注意的是，对于 OFPGT_SELECT 类型的 group ，并不需要指定 watch_port 、 watch_group 参数。

```
actions1 = [parser.OFPActionOutput(2)]
actions2 = [parser.OFPActionOutput(3)]
match = parser.OFPMatch(in_port=1)
buckets = [parser.OFPBucket(weight=50,actions=actions1),
    parser.OFPBucket(weight=50,actions=actions2)]
group_id = datapath.id
req = parser.OFPGroupMod(datapath, ofproto.OFPGC_ADD,ofproto.OFPGT_SELECT, group_id, buckets)
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。

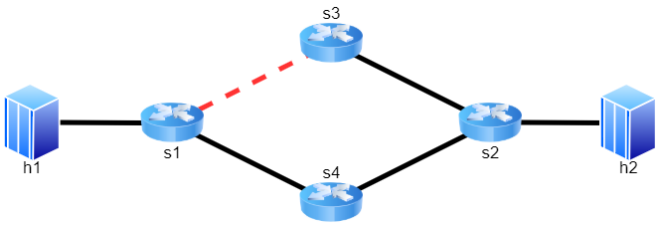


4 Problem4

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。

```
@set_ev_cls(ofp_event.EventOFPPortStatus,MAIN_DISPATCHER)
def _port_state_change_handler(self, ev):
      datapath = ev.msg.datapath
      parser = datapath.ofproto_parser
      ofproto = datapath.ofproto
      desc = ev.msg.desc
      if desc.state & ofproto.OFPPS_LIVE:
            if datapath.id in (1,2) and desc.port_no in (2,3) and self.failover:
                  self.logger.warning(f"datapath {datapath.id} port {desc.port_no} UP!")
                  if datapath.id == 1:
                        datapath_ = self.datapaths[2]
                  else:
                        datapath_ = self.datapaths[1]
                  out_port = 3 if desc.port_no == 2 else 2
                  match = parser.OFPMatch(in_port=1)
                  self.del_flow(datapath_,match,out_port)
                  self.logger.info(f"del flow: datapath {datapath.id} in_port:1,out_port:{out_port}")
                  self.failover = False
      else:
            if datapath.id in (1,2) and desc.port_no in (2,3) and not self.failover:
                  self.logger.warning(f"datapath {datapath.id} port {desc.port_no} DOWN!")
                  if datapath.id == 1:
                        datapath_ = self.datapaths[2]
```

```
                else:
                    datapath_ = self.datapaths[1]
                out_port = 3 if desc.port_no == 2 else 2
                match = parser.OFPMatch(in_port=1)
                actions = [parser.OFPActionOutput(out_port)]
                self.add_flow(datapath_,10,match,actions)
                self.logger.info(f"add flow: datapath {datapath.id} in_port:1,out_port:{out_port}")
                self.failover = True
```

其中，del_flow 函数实现如下：

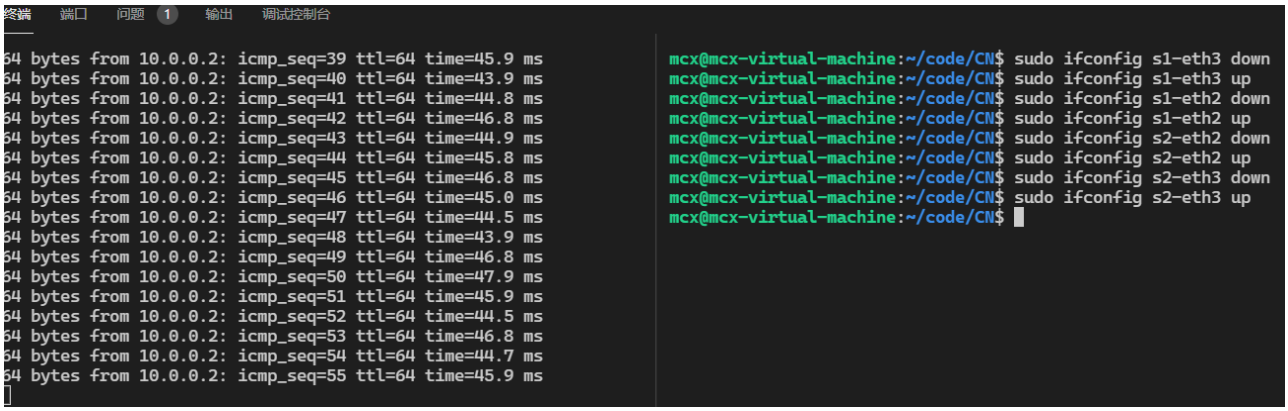
```
def del_flow(self, datapath, match, out_port):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    mod = parser.OFPFlowMod(datapath=datapath,match=match,
                command=ofproto.OFPFC_DELETE,out_port=out_port,
                out_group=ofproto.OFPG_ANY,flags=ofproto.OFPFF_SEND_FLOW_REM)
    datapath.send_msg(mod)
```

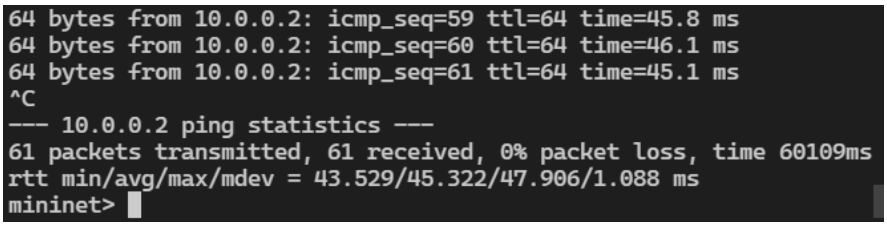
值得一提的是，对路由规则的删除相较于添加有更高的匹配要求，即使目标规则设置时没有指定 out_group ，在删除时也必须将 out_group 指定为 OFPG_ANY ，否则无法匹配到目标规则。

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。





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。