

COMP550705 软件定义网络

实验报告

第 2 次



姓名

班级

学号

日期

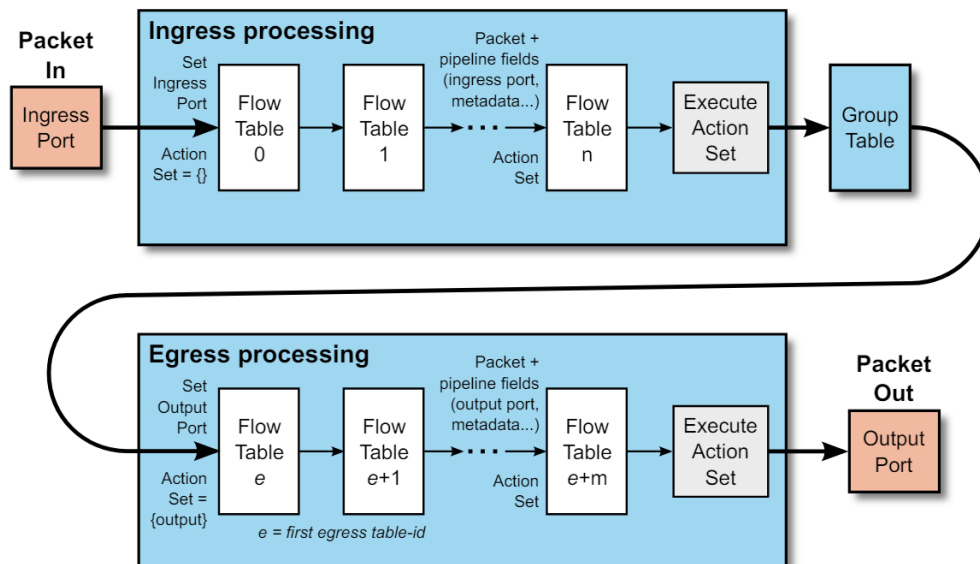
一、 OpenFlow 以及 Ryu 简介

1.1 OpenFlow

OpenFlow 是一种网络通信协议，应用于 SDN 架构中控制器和转发器之间的通信。软件定义网络 SDN 的一个核心思想就是“转发、控制分离”，要实现转、控分离，就需要在控制器与转发器之间建立一个通信接口标准，允许控制器直接访问和控制转发器的转发平面。OpenFlow 引入了“流表”的概念，转发器通过流表来指导数据包的转发。控制器正是通过 OpenFlow 提供的接口在转发器上部署相应的流表，从而实现对转发平面的控制。

Flow Table（流表）是 OpenFlow 交换机收到流后进行抓发的规则表，相当于二层的 Mac 地址表和三层的路由表，OpenFlow 1.1 之后支持多张流表流表由流表项组成。每个流表项包含：

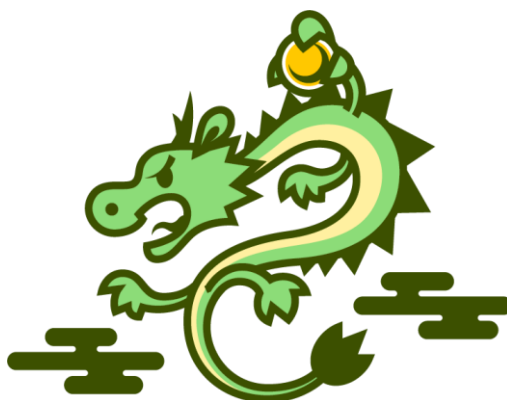
- **match fields:** to match against packets. These consist of the ingress port and packet headers, and optionally other pipeline fields such as metadata specified by a previous table.
- **priority:** matching precedence of the flow entry.
- **counters:** updated when packets are matched.
- **instructions:** to modify the action set or pipeline processing.
- **timeouts:** maximum amount of time or idle time before flow is expired by the switch.
- **cookie:** opaque data value chosen by the controller. May be used by the controller to filter flow entries affected by flow statistics, flow modification and flow deletion requests. Not used when processing packets.
- **flags:** flags alter the way flow entries are managed, for example the flag OFPPF_SEND_FLOW_REM triggers flow removed messages for that flow entry.



1.2 Ryu

Ryu 是日本 NTT 公司推出的 SDN 控制器框架，它基于 Python 开发，模块清晰，可扩展性好，逐步取代了早期的 NOX 和 POX。其具有以下特征：

- Ryu 支持 OpenFlow 1.0 到 1.5 版本，也支持 Netconf, OF-CONFIG 等其他南向协议
- Ryu 可以作为 OpenStack 的插件
- Ryu 提供了丰富的组件，便于开发者构建 SDN 应用



二、实验任务

实验任务一：自学习交换机



The ARPANET in December 1969

1969 年的 ARPANET 非常简单，仅由四个结点组成。假设每个结点都对应一个交换机，每个交换机都具有一个直连主机，你的任务是实现不同主机之间的正常通信。实验指导书给出的简单交换机洪泛数据包，虽然能初步实现主机间的通信，但会带来不必要的带宽消耗，并且会使通信内容泄露给第三者。因此，请你在简单交换机的基础上实现二层自学习交换机，避免数据包的洪泛。

- SDN 自学习交换机的工作流程可以参考：

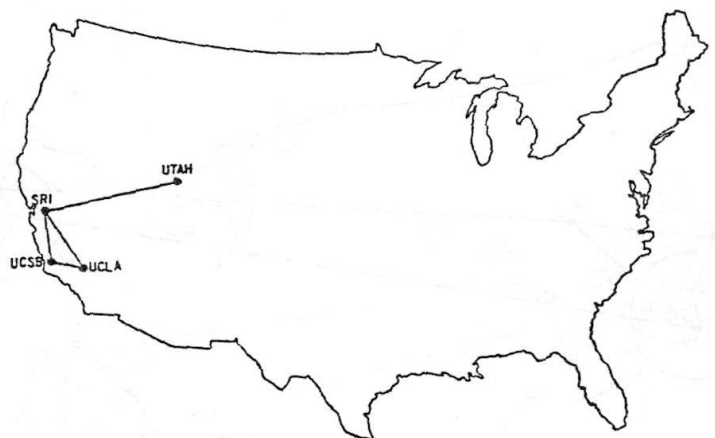
- (1) 控制器为每个交换机维护一个 mac-port 映射表。
- (2) 控制器收到 packet_in 消息后，解析其中携带的数据包。
- (3) 控制器学习 src_mac - in_port 映射。
- (4) 控制器查询 dst_mac，如果未学习，则洪泛数据包；如果已学习，则向指定端口转发数据包(packet_out)，并向交换机下发流表项(flow_mod)，指导交换机转发同类型的数据包。

- 网络拓扑为 topo_1969_1.py，启动方式：

```
sudo python topo_1969_1.py
```

- 可以不考虑交换机对数据包的缓存(no_buffer)

实验任务二：处理环路广播



UCLA 和 UCSB 通信频繁，两者间建立了一条直连链路。在新的拓扑 topo_1969_2.py 中运行自学习交换机，UCLA 和 UTAH 之间无法正常通信。分析流表发现，源主机虽然只发了很少的几个数据包，但流表项却匹配了上千次；WireShark 也截取到了数目异常大的相同报文。这实际上是 ARP 广播数据包在环状拓扑中洪泛导致的，传统网络利用生成树协议解决这一问题。在 SDN 中，不必局限于生成树协议，可以通过多种新的策略解决这一问题。以下给出一种解决思路，请在自学习交换机的基础上完善代码，解决问题：当序号为 `dpid` 的交换机从 `in_port` 第一次收到某个 `src_mac` 主机发出，询问 `dst_ip` 的广播 ARP Request 数据包时，控制器记录一个映射 `(dpid,src_mac,dst_ip)->in_port`。下一次该交换机收到同一 `(src_mac, dst_ip)` 但 `in_port` 不同的 ARP Request 数据包时直接丢弃，否则洪泛。

三、实验内容与分析

3.1 自学习交换机

为了实现自学习交换机，控制器维护一个 `mac-port` 映射表 `self.mac_to_port = {}`。这个表记录每个 `dpid`（交换机序号）的源地址—端口字典。

当控制器收到 `packet_in` 消息后，会解析其中携带的数据包（具体地说，会解析得到入端口、源 MAC 地址以及目的 MAC 地址），接着通过解析得到的内容填充 `mac-port` 映射表。

如果发现数据包的目的 MAC 地址已被记录在 `mac-port` 映射表中了，则应该向指定端口转发数据包并向交换机下发流表项（`flow_mod`，在具体实现中即为调用 `add_flow()`），以指导交换机转发同类型的数据包。如果没有，则只有泛洪数据包才可能将数据包送达目的地址。

值得注意的是，无论是向指定端口转发数据包还是洪泛数据包，都是使用 `parser.OFPPacketOut()` 函数，不涉及流表，只指示交换机将数据包转发到指定端口（泛洪即将数据包广播到所有除接收端口外的其他端口）。

下面给出填充的代码，完整代码可见附件 Learning_Switch.py。

```

# Record the mapping relationship between src_mac to in_port
self.mac_to_port[dpid][src] = in_port
if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = ofp.OFPP_FLOOD
actions = [parser.OFPAActionOutput(out_port)]
if out_port != ofp.OFPP_FLOOD: # Already learned, therefore sending flow
table entries
    match = parser.OFPMatch(in_port = in_port, eth_src = src, eth_dst = dst)
    self.add_flow(dp, 1, match, actions)
# Anyway, instruct the switch to send packets
# Function parser.OFPPacketOut() does not involve the flow table, it just
instructs the switch
# to forward the packet to the specified port
out = parser.OFPPacketOut(datapath = dp, buffer_id = msg.buffer_id, in_port =
in_port,
    actions = actions, data = msg.data)
dp.send_msg(out)

```

启动网络拓扑，启动控制器。启动控制器的命令如下

```
sudo ryu-manager Learning_Switch.py
```

UCLA ping UTAH 并在 UCSB 客户机上抓包，发现 ping 通且 UCSB 没有收到相关数据包。

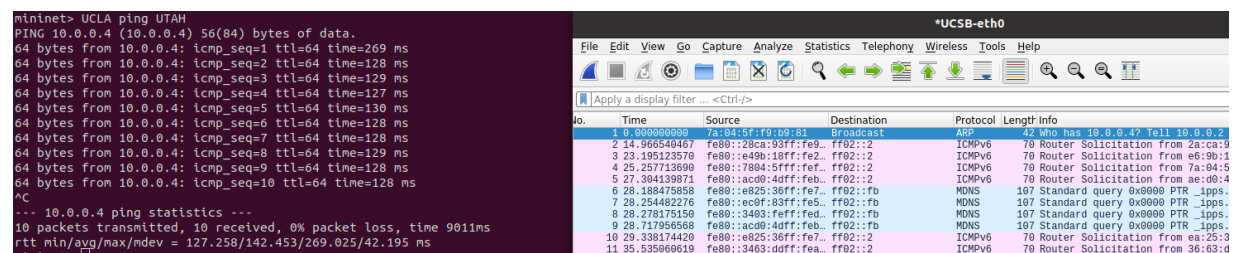


图 3.1.1

3.2 处理环路广播

为了处理环路广播，控制器记录一个映射 $(dpid, src_mac, dst_ip) \rightarrow in_port$ ，在代码中通过表 `self.sw = {}` 实现。当交换机从某端口第一次收到源主机发出的 ARP Request 广播数据包时，控制器填充表项。当下一次该交换机收到同 (src_mac, dst_ip) 但 in_port 不同的 ARP Request 数据包时直接丢弃该包（因为这个包是由于环路产生的无用的请求包）以避免广播风暴。如果 in_port 也相同，则包可能是之后的 ARP 请求，因此照常洪泛即可。

在代码中，同 (src_mac, dst_ip) 但 in_port 不同的 ARP Request 数据包直接丢弃的交换机行为可以通过控制器直接返回控制（对应 3.3 节表中的方法 1）。

下面给出了处理环路广播部分填充的代码。

```
if dst == ETHERNET_MULTICAST and ARP in header_list:
    # you need to code here to avoid broadcast loop to finish mission 2
    arp_packet = header_list[ARP]
    dst_ip = arp_packet.dst_ip
    if (dpid, src, dst_ip) in self.sw:
        if self.sw[(dpid, src, dst_ip)] != in_port: # Received duplicate
            packets on different ports
            return
    else:
        self.sw[(dpid, src, dst_ip)] = in_port
```

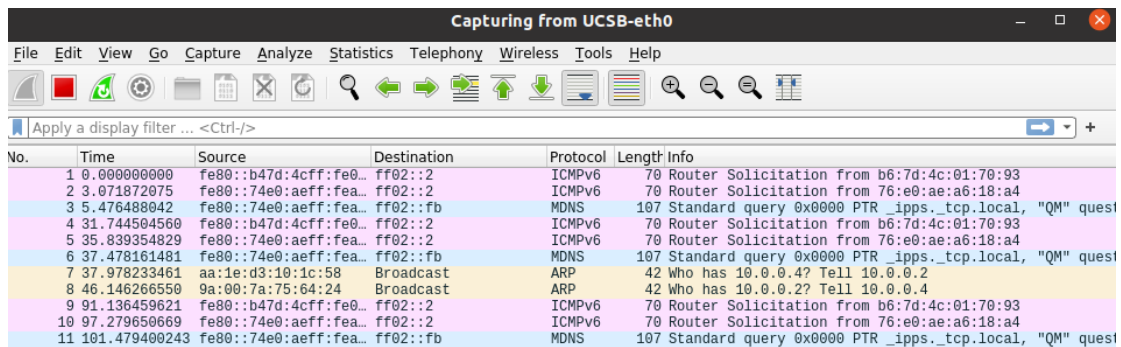
启动网络拓扑，启动控制器。启动控制器的命令如下

```
sudo ryu-manager Broadcast_Loop.py
```

UCLA ping UTAH 并在 UCSB 客户机上抓包，发现 ping 通且 ARP 报文数量正常。

```
mininet> UCLA ping UTAH
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=15.7 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.399 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.073 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.064 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=64 time=0.072 ms
^C
--- 10.0.0.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4064ms
rtt min/avg/max/mdev = 0.064/3.264/15.714/6.226 ms
```

图 3.2.1



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe80::b47d:4cff:fe0...	ff02::2	ICMPv6	70	Router Solicitation from b6:7d:4c:01:70:93
2	3.071872075	fe80::74e0:aeff:fea...	ff02::2	ICMPv6	70	Router Solicitation from 76:e0:ae:a6:18:a4
3	5.476488042	fe80::74e0:aeff:fea...	ff02::fb	MDNS	107	Standard query 0x0000 PTR _ipps._tcp.local, "QM" quest
4	31.744504560	fe80::b47d:4cff:fe0...	ff02::2	ICMPv6	70	Router Solicitation from b6:7d:4c:01:70:93
5	35.839354829	fe80::74e0:aeff:fea...	ff02::2	ICMPv6	70	Router Solicitation from 76:e0:ae:a6:18:a4
6	37.478161481	fe80::74e0:aeff:fea...	ff02::fb	MDNS	107	Standard query 0x0000 PTR _ipps._tcp.local, "QM" quest
7	37.978233461	aa:1e:d3:10:1c:58	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
8	46.146266550	9a:00:7a:75:64:24	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.4
9	91.136459621	fe80::b47d:4cff:fe0...	ff02::2	ICMPv6	70	Router Solicitation from b6:7d:4c:01:70:93
10	97.279650669	fe80::74e0:aeff:fea...	ff02::2	ICMPv6	70	Router Solicitation from 76:e0:ae:a6:18:a4
11	101.479400243	fe80::74e0:aeff:fea...	ff02::fb	MDNS	107	Standard query 0x0000 PTR _ipps._tcp.local, "QM" quest

图 3.2.2

使用以下命令查看流表。

```
dpctl dump-flows
```

查到的流表见图 3.2.3，可见表项的匹配次数明显减少。

```
mininet> dpctl dump-flows
*** s1 ***
cookie=0x0, duration=45.783s, table=0, n_packets=3, n_bytes=126, priority=1,arp,d_src=b6:7c:25:64:33:ce,arp_tpa=10.0.0.4 actions=drop
cookie=0x0, duration=45.773s, table=0, n_packets=8, n_bytes=616, priority=1,in_port="s1-eth2",d_src=ba:26:c8:21:45:9b,d_dst=b6:7c:25:64:33:ce actions=output:"s1-eth4"
cookie=0x0, duration=45.771s, table=0, n_packets=4, n_bytes=392, priority=1,in_port="s1-eth4",d_src=b6:7c:25:64:33:ce,d_dst=ba:26:c8:21:45:9b actions=output:"s1-eth2"
cookie=0x0, duration=52.324s, table=0, n_packets=50, n_bytes=6141, priority=0 actions=CONTROLLER:65535
*** s2 ***
cookie=0x0, duration=45.781s, table=0, n_packets=8, n_bytes=616, priority=1,in_port="s2-eth1",d_src=ba:26:c8:21:45:9b,d_dst=b6:7c:25:64:33:ce actions=output:"s2-eth2"
cookie=0x0, duration=45.774s, table=0, n_packets=4, n_bytes=392, priority=1,in_port="s2-eth2",d_src=b6:7c:25:64:33:ce,d_dst=ba:26:c8:21:45:9b actions=output:"s2-eth1"
cookie=0x0, duration=52.293s, table=0, n_packets=23, n_bytes=2399, priority=0 actions=CONTROLLER:65535
*** s3 ***
cookie=0x0, duration=45.791s, table=0, n_packets=0, n_bytes=0, priority=1,arp,d_src=b6:7c:25:64:33:ce,arp_tpa=10.0.0.4 actions=drop
cookie=0x0, duration=52.323s, table=0, n_packets=42, n_bytes=4961, priority=0 actions=CONTROLLER:65535
*** s4 ***
cookie=0x0, duration=45.786s, table=0, n_packets=8, n_bytes=616, priority=1,in_port="s4-eth2",d_src=ba:26:c8:21:45:9b,d_dst=b6:7c:25:64:33:ce actions=output:"s4-eth1"
cookie=0x0, duration=45.785s, table=0, n_packets=7, n_bytes=518, priority=1,in_port="s4-eth1",d_src=b6:7c:25:64:33:ce,d_dst=ba:26:c8:21:45:9b actions=output:"s4-eth2"
cookie=0x0, duration=52.342s, table=0, n_packets=50, n_bytes=5080, priority=0 actions=CONTROLLER:65535
```

图 3.2.3

3.3 环路处理的其他方法

直接丢弃同 MAC 同目的 IP 的 ARP 包（下发流表实现）处理环路时，值得注意的是，流表若没有进行入端口匹配，控制器下发流表应注意填充 `idle_timeout` 和 `hard_timeout` 项，否则会出现图 3.2.4 的情形，即最开始可以 ping 通但一段时间后无法 ping 通。这种情况的出现是因为 ARP 缓存，主机需要再次进行 ARP 查询，但根据之前设定的流表项这次 ARP 查询包会被直接丢弃。由于 ARP 查询包被丢弃，主机无法收到应答，因此无法 ping 通。填充 `idle_timeout` 和 `hard_timeout` 项后，流表就会经常更新，以适应网络拓扑的变化。填充之后再次执行 UCLA ping UTAH，发现发出的 ICMP 包均能收到回复，见图 3.2.5。

```
mininet> UCLA ping UTAH
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=14.7 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.074 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.077 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=64 time=0.078 ms
64 bytes from 10.0.0.4: icmp_seq=6 ttl=64 time=0.086 ms
64 bytes from 10.0.0.4: icmp_seq=7 ttl=64 time=0.096 ms
64 bytes from 10.0.0.4: icmp_seq=8 ttl=64 time=0.068 ms
From 10.0.0.2 icmp_seq=45 Destination Host Unreachable
From 10.0.0.2 icmp_seq=46 Destination Host Unreachable
From 10.0.0.2 icmp_seq=47 Destination Host Unreachable
From 10.0.0.2 icmp_seq=48 Destination Host Unreachable
From 10.0.0.2 icmp_seq=49 Destination Host Unreachable
From 10.0.0.2 icmp_seq=50 Destination Host Unreachable
```

图 3.2.4

```
64 bytes from 10.0.0.4: icmp_seq=163 ttl=64 time=0.074 ms
64 bytes from 10.0.0.4: icmp_seq=164 ttl=64 time=0.119 ms
64 bytes from 10.0.0.4: icmp_seq=165 ttl=64 time=0.062 ms
64 bytes from 10.0.0.4: icmp_seq=166 ttl=64 time=0.059 ms
^C
--- 10.0.0.4 ping statistics ---
166 packets transmitted, 166 received, 0% packet loss, time 168928ms
rtt min/avg/max/mdev = 0.040/0.140/9.420/0.723 ms
```

图 3.2.5

如果不填充 `idle_timeout` 和 `hard_timeout` 项的话，则 `match` 匹配必须加入 `in_port`。或者更直接地，在发现入端口不匹配后直接返回。这就是 3.2 中使用的方法，概括来说，就是丢弃同 MAC 同目的 IP 但入端口与最开始记录不同的 ARP 包（单纯丢弃相应包）。

以上两种处理都能应对环路。然而这两种方式虽可以支持之后的 ARP 查询，却不能够处理网络故障时的情形（即先前入端口对应的路径已经故障，新 ARP 请求经另一路径从新端口进入）。因此将它们综合起来，丢弃同 MAC 同目的 IP 但入端口与最开始记录不同的 ARP 包且加入可更新的流表。

在下页给出这种方法的实现代码

```

if dst == ETHERNET_MULTICAST and ARP in header_list:
    # you need to code here to avoid broadcast loop to finish mission 2
    arp_packet = header_list[ARP]
    dst_ip = arp_packet.dst_ip
    if (dpid, src, dst_ip) in self.sw:
        if self.sw[(dpid, src, dst_ip)] != in_port: # Received duplicate
packets on different ports
            # Add flow entry to drop ARP packet
            match = parser.OFPMatch(eth_type = ether_types.ETH_TYPE_ARP,
                eth_src = src,
                arp_tpa = arp_packet.dst_ip,
                in_port = in_port)
            actions = []
            self.add_flow(dp, 1, match, actions, idle_timeout = 5,
hard_timeout = 10) # Flow table has to update.
            return
    else:
        self.sw[(dpid, src, dst_ip)] = in_port

```

将上文提到的方法总结为下表：

方法 序号	解决环路的方法	是否解决环路
1	丢弃同 MAC 同目的 IP 但入端口与最开始记录不同的 ARP 包（单纯丢弃相应包）	是
2	丢弃同 MAC 同目的 IP 的 ARP 包（下发流表实现）	不完全
3	丢弃同 MAC 同目的 IP 的 ARP 包（下发可更新流表实现）	是
4	丢弃同 MAC 同目的 IP 但入端口与最开始记录不同的 ARP 包（下发可更新流表实现）	是