一、实验任务

（1）实验任务一：自学习交换机。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。
（2）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。
（3）附加题：实验任务二只给出了一种参考方案， SDN 中还有多种方案可供选择，请尝试设计实现一种新的策略解决环路广播问题。

二、实验原理

（1）OpenFlow 是一种网络通信协议，主要用于 SDN 架构中控制器和转发器之间的通信。SDN 的核心思想是“转发与控制分离”，也就是说，网络设备的转发决策和控制决策是分开的。为了实现这一目标，需要在控制器与转发器之间建立一个通信接口标准，使得控制器可以直接访问和控制转发器的转发平面。

OpenFlow 协议通过引入“流表”的概念，解决了这一问题。流表是 OpenFlow 交换机接收到数据流后进行转发决策的规则表，类似于二层的 MAC 地址表 和三层的路由表。OpenFlow 控制器可以通过 OpenFlow 协议向交换机下发流表条目，从而控制数据包的转发行为。

每个流表项包含多个字段，主要包括以下内容：

流匹配字段：描述了数据流的特征，如源 MAC 地址、目标 MAC 地址、源 IP 地址、目标 IP 地址、协议类型等。

动作字段：定义了交换机在匹配到某条流表项时需要执行的动作，比如转发到某个端口、修改数据包、丢弃数据包等。

计数器：记录与该流相关的数据包数量和字节数，便于流量监控和统计。

在 OpenFlow 1.1 及以后的版本中，支持多张流表，能够更加灵活地进行流的匹配和转发。每个流表项决定了如何处理数据包，当数据包与流表项匹配时，

交换机会根据流表项指定的动作来处理数据包。

（2）OS-Ken 基于 Python 语言进行开发，具备模块化的特点，对 OpenFlow 协议提供了良好的支持，借助这一特性，它能够有效地对 SDN 网络中的交换机和路由器等设备进行控制和管理。

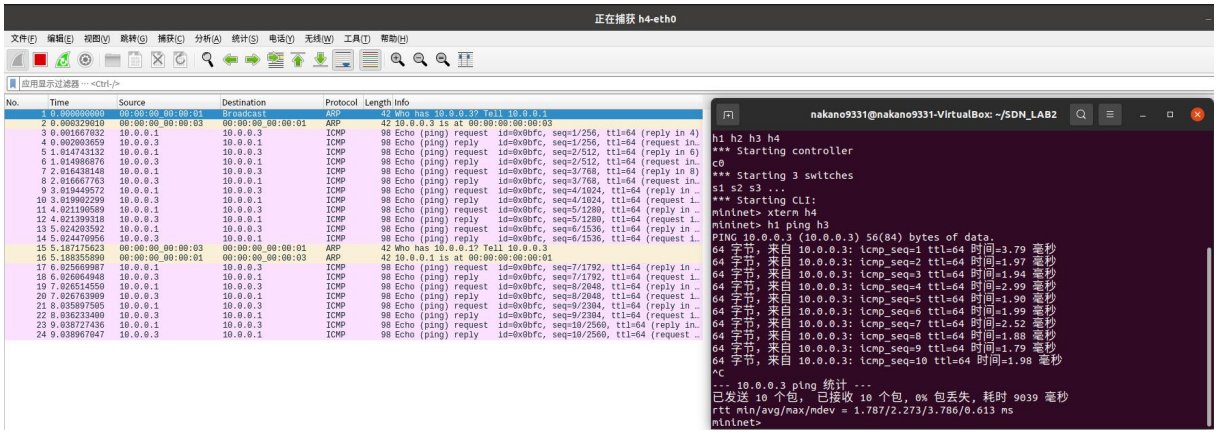
OS-Ken 控制器框架具有一系列显著的主要特征。它支持多个 OpenFlow 版本，增强了其兼容性和适用性。同时，它也具备 OpenStack 插件支持，能够与 OpenStack 等云平台进行良好的集成，为构建复杂的网络环境提供了便利。此外，OS-Ken 拥有丰富的组件，涵盖了从基础的网络功能到高级的策略管理等多个方面，满足了不同场景下的需求。其易于扩展的特性更是让开发者能够快速地对其进行定制和优化，以适应不断变化的网络需求。

三、实验过程

3.1 OS-Ken 编程示例：简单交换机

执行 sudo mn --controller remote --mac --topo=tree,2,2 使用 mininet 创建树状拓扑，并将实验指导书所给出的参考代码保存为 simple_switch.py ，启动控制器。

在 mininet 的 CLI 中打开 h4 的 xterm ，启动 wireshark 抓取端口 h4-eth0，同时在 mininet 中 h1 ping h3 ，查看 wireshark 中关于 h4 的抓包情况。可以观察到如下输出结果。



通过抓包结果可以验证交换机存在验证缺陷，代码中的 packet_in_handler 函数导致交换机对未知目的 MAC 地址的数据包进行泛洪。这意味着在初始状态下，任何主机，如 h1，发送的包，都会被交换机转发到所有其他端口，直到交换机学习到 MAC 地址对应的端口。

3.2 自学习交换机

3.2.1 SDN 自学习交换机工作流程

①控制器为每个交换机维护一张 mac-port 映射表；

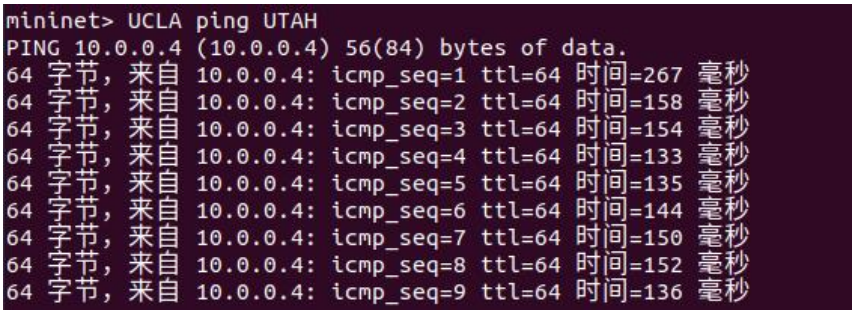
②控制器收到 packet_in 消息后，解析其中携带的数据包；

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。

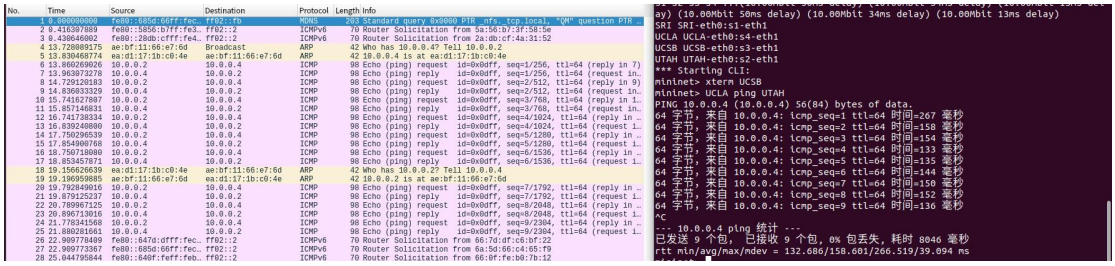
网络拓扑为 topo_1969_1.py，启动方式为执行 sudo python topo_1969_1.py 命令，可以不考虑交换机对数据包的缓存。

3.2.2 UCLA ping UTAH

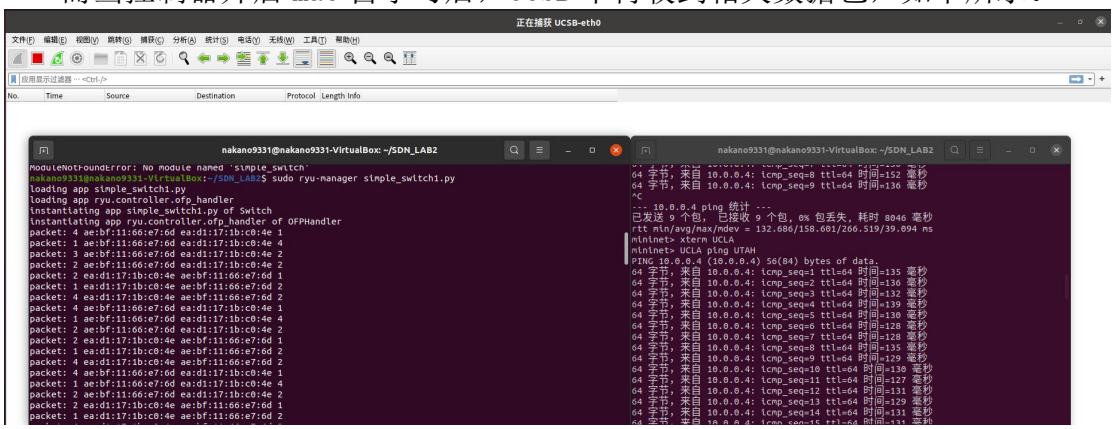


由于在构建拓扑的过程中，为交换机之间的链路指定了时延，因此这里的时延都在 100ms 以上。其中，第一次时延较大是控制器与交换机之间的转发数据包及流表项导致的，后续的数据包则通过匹配流表项后直接转发。

若控制器无进行 mac 自学习，只洪泛数据包时，UCSB 也能收到 UCLA ping UTAH 的 ICMP 报文，如下所示。



而当控制器开启 mac 自学习后，UCSB 不再收到相关数据包，如下所示。



为了实现自学习交换机，通过代码设置控制器维护 mac_to_port 映射表，记

录每个交换机的源 MAC 地址及其对应的入端口。收到 packet_in 消息后，控制器解析源 MAC、目的 MAC 和入端口信息，并更新映射表。

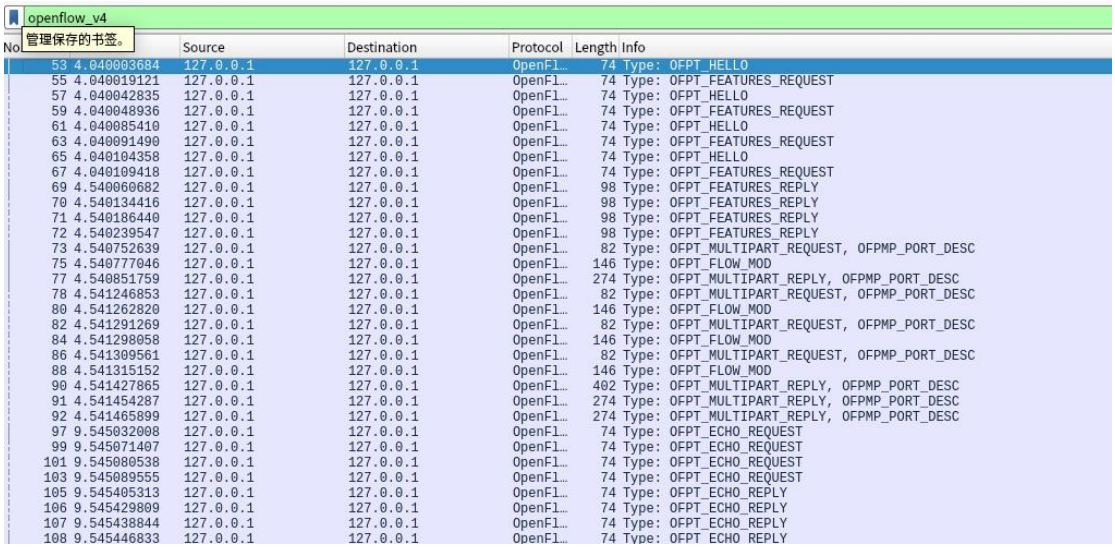
如果目的 MAC 已在映射表中，则数据包被定向转发至相应端口，并下发流表项，确保后续相同流量可直接由交换机处理，而无需再通知控制器。

如果目的 MAC 未知，则交换机对数据包进行洪泛，使其广播至所有端口，以确保目的主机能接收数据。

流表项的下发优化了后续转发过程，提高了网络效率。无论是否学习到目的 MAC，控制器最终都会通过 OFPPacketOut 发送数据包，以确保其正确转发。控制器需要处理高效的流表管理，确保交换机在不需要时清除不必要的流表项，避免浪费资源。

通过以上操作，控制器能够高效地管理 MAC 地址学习和数据包转发，大大提升交换机的性能。使得 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。

控制器与交换机建立连接的过程如下图所示。



其中 OFPT_HELLO 消息用于交换 OpenFlow 协议版本信息，以便交换机和控制器确定它们支持的 OpenFlow 协议版本。发送方是交换机和控制器都可以发送该消息。传输时机在 TCP/TLS 连接建立时发送，是 OpenFlow 通信渠道建立过程的一部分。通过交换版本信息，确保控制器和交换机之间的通信兼容。

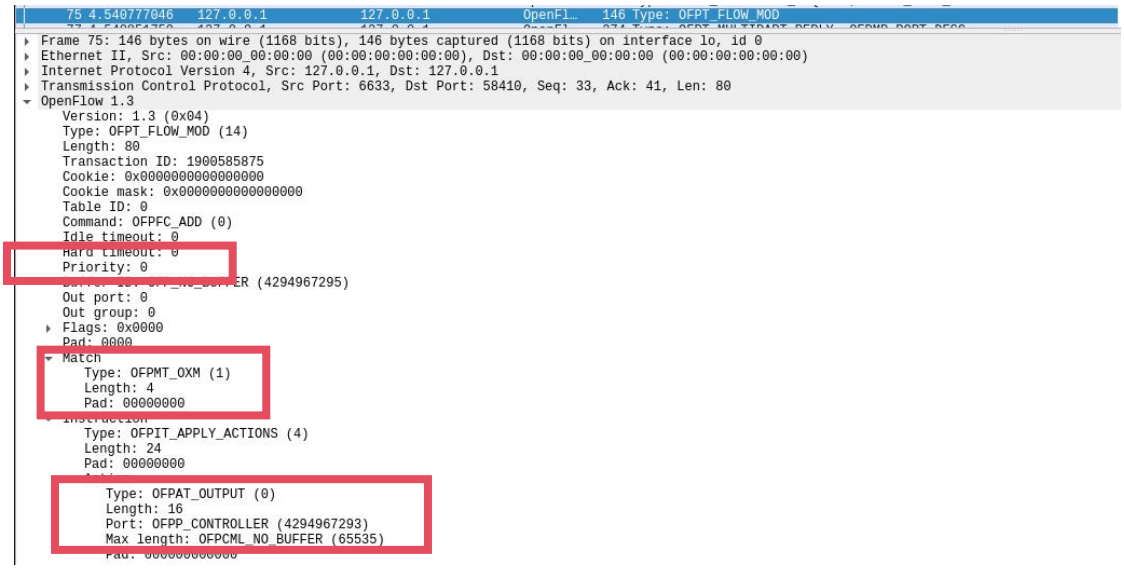
OFPT_FEATURES_REQUEST 功能是控制器向交换机请求相关信息，例如交换机的功能、可用端口、流表等。发送方为控制器向交换机发送该消息。交换机在接收到请求后，会返回 OFPT_FEATURES_REPLY 消息，其中包含交换机的详细信息。控制器通过该消息获取交换机的功能和配置，进而做出相应的网络配置。

而 OFPT_MULTIPART_REQUEST 消息则向交换机发送一组相关联的请求，用于查询交换机上的状态和属性信息。其中每个请求都会被分配一个唯一的 ID，便于控制器和交换机对相应的请求和响应进行匹配。通过多部分请求，控制器可以一次性查询交换机的多个状态信息。

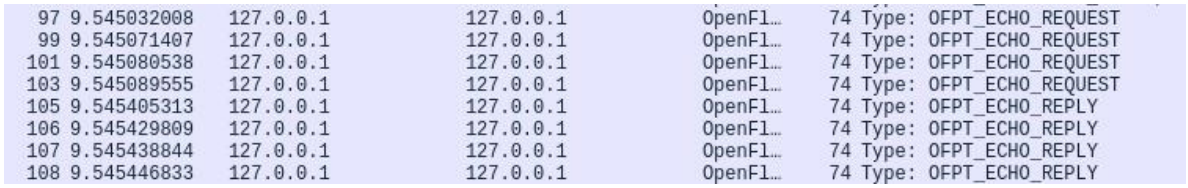
OFPT_MULTIPART_REPLY 消息用于向控制器返回 OFPT_MULTIPART_REQUEST 消息的响应结果，包含了请求中描述的一系列交换机状态或配置信息。请求 ID 是响应消息中包含与请求消息相对应的请求 ID，帮助控制器识别并解析响应消息。当 type 字段取值为 OFPMP_PORT_DESC 时，表示消息携带的是与端口描述相关的

信息。通过该消息，控制器可以获取关于交换机的详细状态信息，并根据这些信息来配置流表、调整网络拓扑结构等。

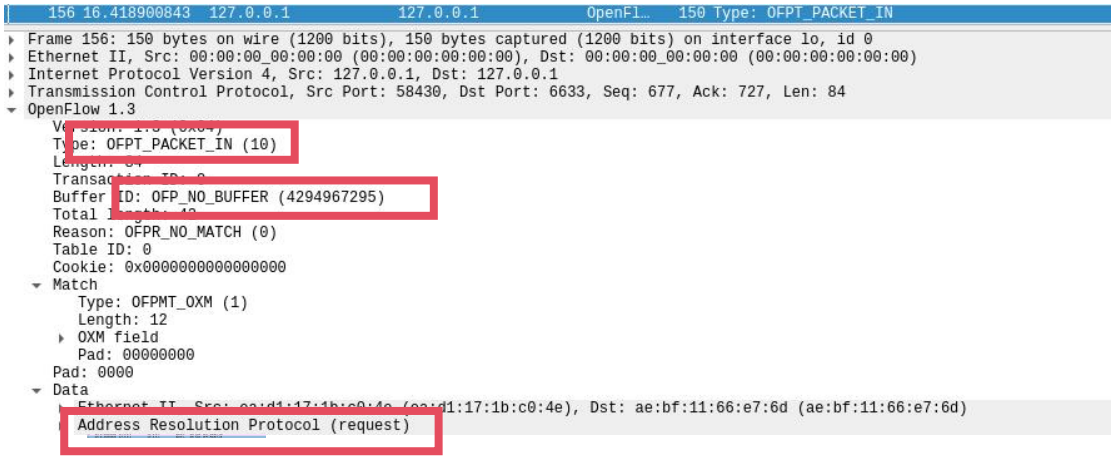
当交换机与控制器建立连接后，控制器会下发一条默认流表项，优先级设为最低（0），匹配条件为空，即匹配所有数据包，执行动作是将数据包发送至控制器处理。具体如下所示。



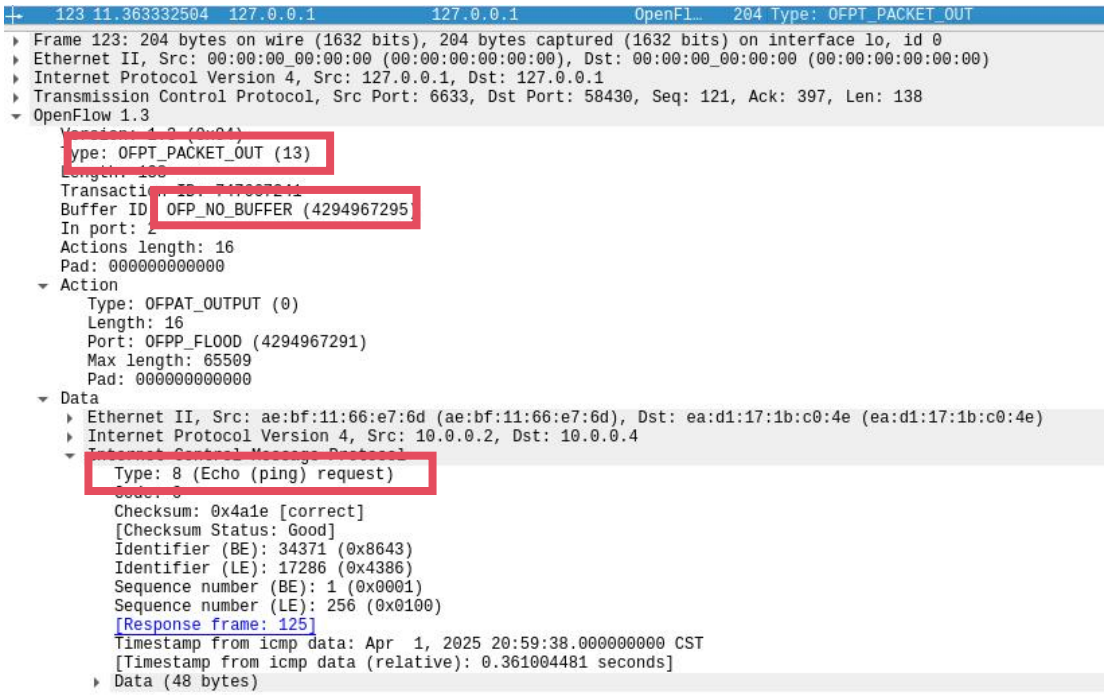
而 OFPT_ECHO_REQUEST 是 OpenFlow 协议中的一种消息类型，用于检测控制器与交换机之间的连接状态。当交换机发送 OFPT_ECHO_REQUEST 消息时，控制器应立即返回包含相同数据的 OFPT_ECHO_REPLY 消息。这一过程类似于网络工具 ping 命令，可用于检测连接的活性和延迟，同时可携带额外信息，如时间戳或带宽测量数据。



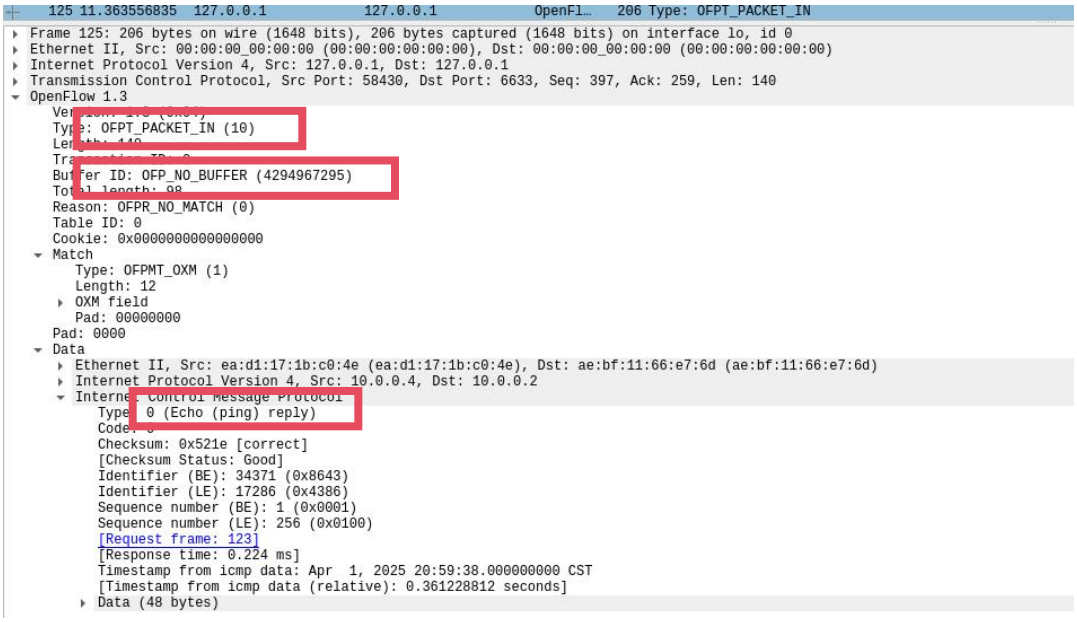
当 UCLA 设备尝试 ping UTAH 时，它首先发送 ARP 请求来获取 UTAH 的 MAC 地址。交换机收到 ARP 请求后，因未匹配到已有流表项，会依据 table-miss 规则将数据包转发至控制器。由于交换机不缓存数据包，控制器接收到的 PACKET_IN 消息中会包含完整的 ARP 请求数据。ARP 请求报文如下所示。



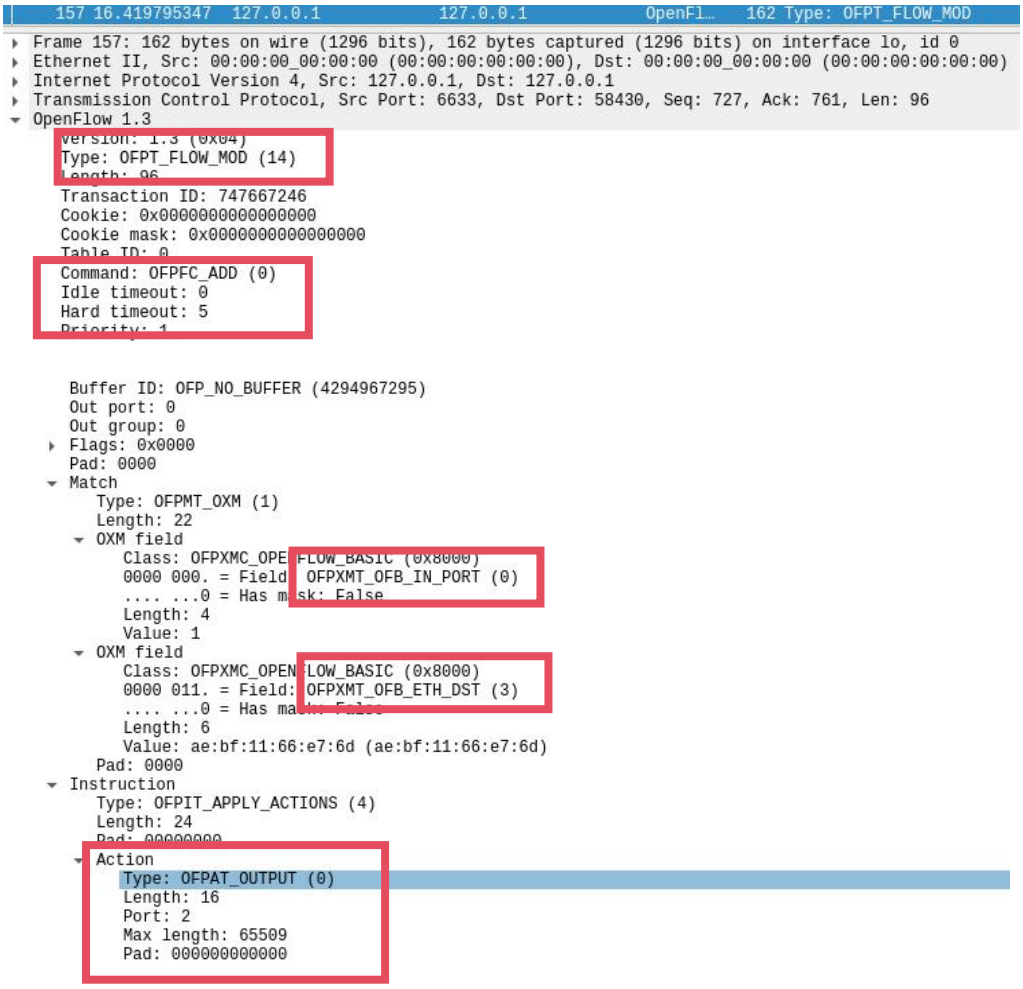
ARP 请求的目的 MAC 地址是广播地址，而控制器的 MAC-Port 映射表中尚未记录该地址。因此，控制器指示交换机对 ARP 请求进行泛洪，并在过程中学习 UCLA 的 MAC 地址及其对应端口。泛洪 ARP 请求报文如下所示。



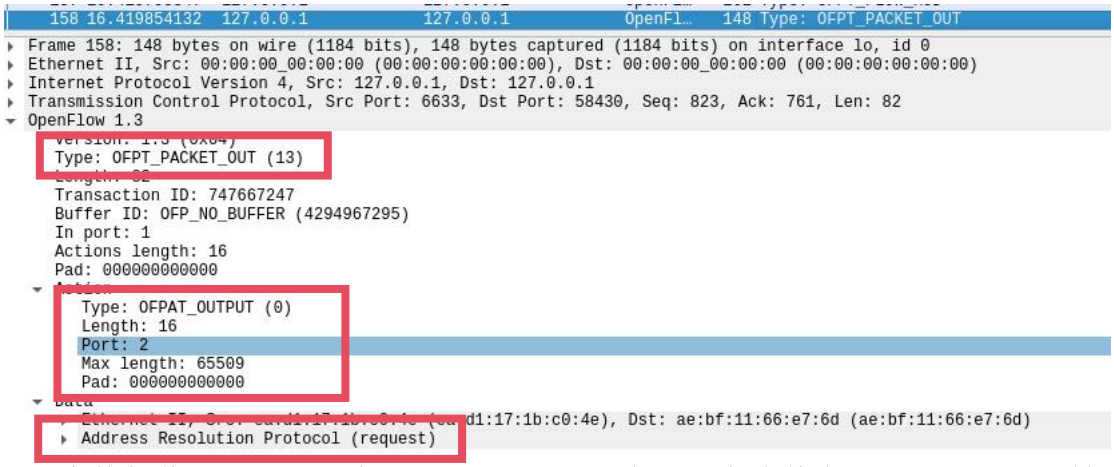
当 UTAH 设备回应 ARP 请求时，其 ARP 应答报文到达交换机，因未匹配到现有流表项，再次触发 table-miss 规则，导致 OFPT_PACKET_IN 消息发送至控制器，其中包含完整的 ARP 应答数据。ARP 应答报文如下所示。



控制器下发流表项具体情况如下。



若控制器在先前收到 ARP 请求时已记录了目标 MAC 地址，因此可以下发新的流表项。该流表项匹配入端口与目标 MAC 地址，并指定转发至相应端口，且设定硬超时为 5 秒。优先级为 1，高于 table-miss 规则，以确保后续数据流不再触发 table-miss。随后，控制器还会将 ARP 应答报文转发至交换机，并执行相应的转发动作。向指定端口转发 ARP 应答报文如下。



交换机收到控制器下发的 PACKET_OUT 消息后，会将其中的 ARP 应答报文转发至指定端口，使 UCLA 获取到 UTAH 的 MAC 地址。

3.3 处理环路广播

3.3.1 工作流程

为了有效地处理环路广播问题，控制器通过记录一个映射(dpid, src_mac, dst_ip) -> in_port 来追踪每个交换机端口收到的特定数据包，如 ARP 请求。该映射可以通过 self.sw = {}字典来实现。

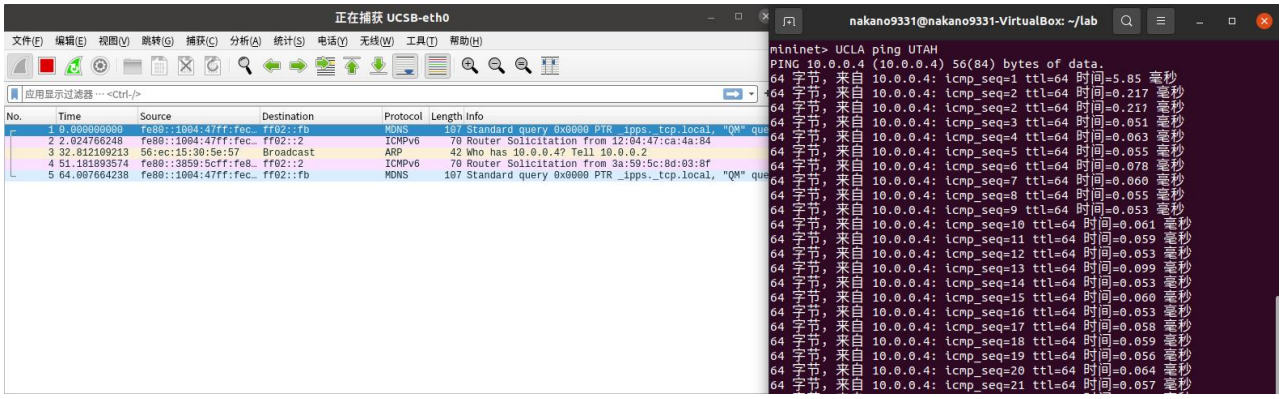
初次接收 ARP 请求时，当交换机从某个端口第一次收到源主机发出的 ARP Request 广播数据包，控制器将填充映射表项(dpid, src_mac, dst_ip) -> in_port。当交换机接收到具有相同 src_mac 和 dst_ip 但 in_port 不同的 ARP Request 数据包时，控制器判断这是由于网络环路引起的无用请求包。因此，控制器将丢弃该数据包，避免发生广播风暴。如果 in_port 相同，说明这是后续的 ARP 请求，数据包仍然有效，因此照常洪泛。

3.3.2 UCLA ping UTAH

控制器维护一个 self.sw 映射表，记录(dpid, src_mac, dst_ip)与 in_port 之间的映射关系，以跟踪 ARP 请求的流向并避免环路导致的广播风暴。

控制器首先判断收到的数据包是否为 ARP 广播请求。如果是，则提取目标 IP 地址 dst_ip。控制器在 self.sw 中查找(dpid, src_mac, dst_ip)是否已有记录。若该组合已存在，则进一步检查当前 in_port 是否与先前记录的端口不同，若不同，说明数据包因环路被重复接收，直接丢弃，避免广播风暴。若相同，说明是正常的 ARP 请求，继续处理并进行洪泛。若 (dpid, src_mac, dst_ip) 之前未记录，则说明该 ARP 请求是首次到达，控制器将其记录到 self.sw 中，并允许其正常传播。

在网络拓扑启动后，使用 uv run osken-manager Loop.py 命令启动控制器。UCLA 设备 ping UTAH，并在 UCSB 客户机上抓包，结果显示 ping 正常，且 ARP 报文数量符合预期，说明控制器成功抑制了不必要的广播风暴。



由于拓扑构建时未指定交换机间的链路时延，因此本实验的时延相较于 3.2 实验内容中的实验更小。首次通信时，较高的时延主要源于控制器处理数据包并下发流表项的过程，而后续数据包则可直接匹配流表项进行转发，从而显著降低时延。

dpctl dump-flows 命令用于查看交换机的流表内容，显示流表中的各个表项及其匹配情况。通过该命令分析控制器下发的流表项和它们的匹配次数，从而推断出交换机转发的流量模式。

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=2.368s, table=0, n_packets=119153, n_bytes=11517835, priority=0 actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=2.369s, table=0, n_packets=118555, n_bytes=11472573, priority=0 actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=2.372s, table=0, n_packets=119129, n_bytes=11514797, priority=0 actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=2.374s, table=0, n_packets=119159, n_bytes=11521041, priority=0 actions=CONTROLLER:65535
mininet> UCLA ping UTAH
```

观察到流表项的匹配次数显著减少。经过多次实验，同一交换机内某条流表项的匹配次数始终比另一条多 1。额外的匹配过程应对应 ARP 应答报文，而其余匹配次数则来自 ICMP 报文。因此，可以推测匹配次数较多的流表项，其 output 端口对应 UCLA 在 ping 过程中接收数据包的端口，而匹配次数较少的流表项则对应发送端口。

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=165.728s, table=0, n_packets=49, n_bytes=4578, priority=1,in_port="s1-eth2",dl_src=0a:7e:5e:67:6f:56,dl_dst=56:ec:
15:30:5e:57 actions=output:"s1-eth4"
 cookie=0x0, duration=165.727s, table=0, n_packets=48, n_bytes=4480, priority=1,in_port="s1-eth4",dl_src=56:ec:15:30:5e:57,dl_dst=0a:7e:
5e:67:6f:56 actions=output:"s1-eth2"
 cookie=0x0, duration=173.602s, table=0, n_packets=15, n_bytes=1216, priority=0 actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=165.731s, table=0, n_packets=49, n_bytes=4578, priority=1,in_port="s2-eth1",dl_src=0a:7e:5e:67:6f:56,dl_dst=56:ec:
15:30:5e:57 actions=output:"s2-eth2"
 cookie=0x0, duration=165.729s, table=0, n_packets=48, n_bytes=4480, priority=1,in_port="s2-eth2",dl_src=56:ec:15:30:5e:57,dl_dst=0a:7e:
5e:67:6f:56 actions=output:"s2-eth1"
 cookie=0x0, duration=173.604s, table=0, n_packets=8, n_bytes=606, priority=0 actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=173.606s, table=0, n_packets=11, n_bytes=862, priority=0 actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=165.734s, table=0, n_packets=49, n_bytes=4578, priority=1,in_port="s4-eth2",dl_src=0a:7e:5e:67:6f:56,dl_dst=56:ec:
15:30:5e:57 actions=output:"s4-eth1"
 cookie=0x0, duration=165.734s, table=0, n_packets=48, n_bytes=4480, priority=1,in_port="s4-eth1",dl_src=56:ec:15:30:5e:57,dl_dst=0a:7e:
5e:67:6f:56 actions=output:"s4-eth2"
 cookie=0x0, duration=173.608s, table=0, n_packets=12, n_bytes=960, priority=0 actions=CONTROLLER:65535
mininet>
```

3.3.3 OpenFlow 协议分析

当控制器接收到相同(dpid, src_mac, dst_ip)但 in_port 不同时的 ARP 请求数据包时，意味着可能存在环路。为防止广播风暴，控制器在发送 PACKET_OUT 消息时未指定 Actions，且未附带 ARP 请求报文。如下所示。

相反，当控制器接收到相同 (dpid, src_mac, dst_ip)且 in_port 相同的 ARP 请求，或是首次接收到该(dpid, src_mac, dst_ip)组合的 ARP 请求时，控制器会将数据包转发至交换机，并指示交换机执行泛洪。

```
    851 34.362210876  127.0.0.1           127.0.0.1           OpenFl…   148 Type: OFPT_PACKET_OUT
▸ Frame 851: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 6633, Dst Port: 40546, Seq: 129, Ack: 1660, Len: 82
▾ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_PACKET_OUT (13)
    Length: 82
    Transaction ID: 2110217946
    Buffer ID: OFP_NO_BUFFER (4294967295)
    In port: 4
    Actions length: 16
    Pad: 000000000000
  ▾ Action
      Type: OFPAT_OUTPUT (0)
      Length: 16
      Port: OFPP_FLOOD (4294967291)
      Max length: 65509
      Pad: 000000000000
  ▾ Data
    ▸ Ethernet II, Src: d6:dc:12:65:4a:7b (d6:dc:12:65:4a:7b), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
    ▸ Address Resolution Protocol (request)
```

3.4 环路处理的其他方法

为了有效地处理环路广播问题，控制器可以使用几种方法。以下是基于流表的处理方法以及它们的优缺点。

方法一通过在控制器上记录每个(src_mac, dst_ip)对应的 in_port，并丢弃入端口不匹配的 ARP 包，避免环路产生的广播包影响网络。这种方法只能解决简单的环路问题，但无法处理网络拓扑变化或路径故障时的情况。

方法二具体为当交换机接收到 ARP 请求时，控制器会下发流表项，直接丢弃同 MAC 和目的 IP 的 ARP 包。即使入端口不同，仍会丢弃这些包。但是这种方法无法应对 ARP 查询被丢弃后，主机无法收到应答的情况，导致 ping 不通。主机可能会因为没有收到ARP应答而陷入长时间的ARP缓存等待,导致网络通信中断。

方法三通过在流表中加入 idle_timeout 和 hard_timeout 项，确保流表条目可以根据时间自动过期，从而支持网络拓扑变化和故障恢复。这样，网络中的 ARP 查询包可以被适时丢弃，同时在一定时间后会自动清除流表项，避免 ARP 查询被错误丢弃。而缺点是在网络拓扑频繁变化时，可能会导致流表更新较频繁，从而增加网络开销。

方法四结合了第一种方法和第三种方法的优点,通过丢弃ARP包来防止环路，同时在流表中加入 idle_timeout 和 hard_timeout，以确保网络拓扑变化时可以及时更新流表，防止误丢弃 ARP 请求包。既能有效避免环路广播，又能处理网络拓扑变化和路径故障。但与方法三类似，可能会有流表更新开销，但相比方法二，能够处理更多的情况。

实现思路是通过对 ARP 广播包的处理来避免广播环路。首先检查是否是 ARP 包。如果目标 MAC 地址为广播地址且数据包中存在 ARP 协议，则解析出 ARP 包并获取相关信息。

使用一个字典 self.sw 来记录已处理过的ARP包。字典的键是三元组 (dpid, src, dst_ip)，即交换机 ID、源 MAC 地址和目标 IP 地址。如果这个三元组已经存在，则说明之前已经处理过该 ARP 请求。

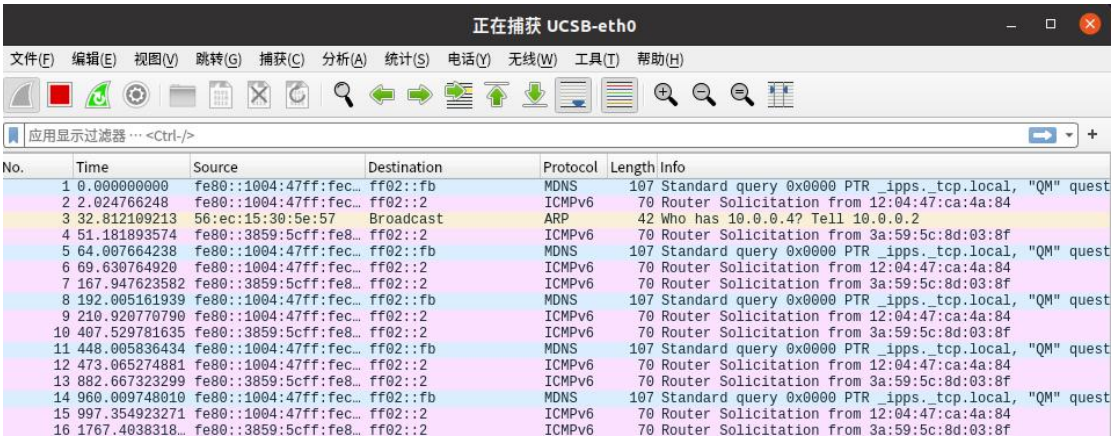
如果该 ARP 请求来自不同的端口，则需要丢弃该 ARP 包。目的是防止 ARP 包在交换机之间产生环路。因为 ARP 请求本质上是广播包，如果交换机收到了来自不同端口的相同 ARP 请求，就可能发生环路。

创建一个流表匹配条件，匹配到目标 IP 是当前 ARP 请求的目标 IP，源 MAC 地址是当前 ARP 包的源 MAC 地址，入端口是当前收到包的端口。设置为空的动作列表，意味着不对该包执行任何操作，直接丢弃。将此流表项添加到交换机中，设置流表项的超时时间。这样，在流表中添加了一个丢弃该 ARP 包的规则，避免了广播环路。

如果是第一次看到该 ARP 请求，则将 (dpid, src, dst_ip) 和对应的 in_port 记录下来。这样，下次如果交换机接收到相同的 ARP 请求，就可以判断是否为环路，并采取相应的措施。在 self.sw 字典中记录下该 ARP 请求的端口，用于后续判断是否为广播环路。

以下输出结果验证了思路的可行性，成功 ping 通。

```
mininet> UCLA ping UTAH
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 字节，来自 10.0.0.4: icmp_seq=1 ttl=64 时间=0.209 毫秒
64 字节，来自 10.0.0.4: icmp_seq=2 ttl=64 时间=0.060 毫秒
64 字节，来自 10.0.0.4: icmp_seq=3 ttl=64 时间=0.061 毫秒
64 字节，来自 10.0.0.4: icmp_seq=4 ttl=64 时间=0.062 毫秒
64 字节，来自 10.0.0.4: icmp_seq=5 ttl=64 时间=0.061 毫秒
64 字节，来自 10.0.0.4: icmp_seq=6 ttl=64 时间=0.059 毫秒
64 字节，来自 10.0.0.4: icmp_seq=7 ttl=64 时间=0.059 毫秒
^C
--- 10.0.0.4 ping 统计 ---
已发送 7 个包， 已接收 7 个包，0% 包丢失，耗时 6164 毫秒
```



源代码

simple_switch1.py

```
from os_ken.base import app_manager
from os_ken.controller import ofp_event
from os_ken.controller.handler import MAIN_DISPATCHER,
CONFIG_DISPATCHER
from os_ken.controller.handler import set_ev_cls
from os_ken.ofproto import ofproto_v1_3
from os_ken.lib.packet import packet
from os_ken.lib.packet import ethernet
class Switch(app_manager.OSKenApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(Switch, self).__init__(*args, **kwargs)
        # maybe you need a global data structure to save the mapping
        self.mac_to_port = {}
    def add_flow(self, datapath, priority, match,
                 actions, idle_timeout=0, hard_timeout=0):
        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.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_BUFFER)]
        self.add_flow(dp, 0, match, actions)
    @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
        # the identity of switch
        dpid = dp.id
        # the port that receive the packet
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        # get the mac
        dst = eth_pkt.dst
        src = eth_pkt.src

        # we can use the logger to print some useful information
        self.logger.info('packet: %s %s %s %s', dpid, src, dst, in_port)
        # You need to code here to avoid the direct flooding
        # Have fun!
        # :)
        # 每台交换机学习 mac-port 表
        self.mac_to_port.setdefault(dpid, {})
        self.mac_to_port[dpid][src] = in_port
        if dst in self.mac_to_port[dpid]:
            # dst_mac 已学习则按指定端口转发
```

```
            out_port = self.mac_to_port[dpid][dst]
        else:
            # dst_mac 未学习则泛洪
            out_port = ofp.OFPP_FLOOD
        # 执行转发动作
        actions = [parser.OFPActionOutput(out_port)]

        if out_port != ofp.OFPP_FLOOD:
            # dst_mac 已学习则下发流表，指导同类型报文转发
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(dp, 1, match, actions, hard_timeout=5)
        # 判断交换机是否有缓存
        data = None
        if msg.buffer_id == ofp.OFP_NO_BUFFER:
            data = msg.data
        # 控制器向交换机发送 PACKET_OUT
        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,
                                  in_port=in_port,         actions=actions,
data=data)
        dp.send_msg(out)
```

Broadcast_Loop1.py

```
from os_ken.base import app_manager
from os_ken.controller import ofp_event
from      os_ken.controller.handler      import      MAIN_DISPATCHER,
CONFIG_DISPATCHER
from os_ken.controller.handler import set_ev_cls
from os_ken.ofproto import ofproto_v1_3
from os_ken.lib.packet import packet
from os_ken.lib.packet import ethernet
from os_ken.lib.packet import arp
from os_ken.lib.packet import ether_types

ETHERNET = ethernet.ethernet.__name__
ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__

class Switch_Dict(app_manager.OSKenApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Switch_Dict, self).__init__(*args, **kwargs)
        self.sw = {}  # (dpid, src_mac, dst_ip)=>in_port, you may use it
```

```
in mission 2
        # maybe you need a global data structure to save the mapping
        # just data structure in mission 1
        self.mac_to_port = {}

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0,
hard_timeout=0):
        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.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_BUFFER)]
        self.add_flow(dp, 0, match, actions)

    @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

        # the identity of switch
        dpid = dp.id
        self.mac_to_port.setdefault(dpid, {})
        # the port that receive the packet
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        if eth_pkt.ethertype == ether_types.ETH_TYPE_LLDP:
```

```
            return
        if eth_pkt.ethertype == ether_types.ETH_TYPE_IPV6:
            return

        # get the mac
        dst = eth_pkt.dst
        src = eth_pkt.src

        # get protocols
        header_list = dict((p.protocol_name, p) for p in pkt.protocols if
type(p) != str)

        if dst == ETHERNET_MULTICAST and ARP in header_list:
            # you need to code here to avoid broadcast loop to finish mission
2
            # 获取目的 ip 地址
            arp_dst_ip = header_list[ARP].dst_ip
            if (dpid, src, arp_dst_ip) in self.sw:
                # in_port 与第一次收到时记录的不同，表示出现了环路
                if self.sw[(dpid, src, arp_dst_ip)] != in_port:
                    # 丢弃数据包
                    out        =        parser.OFPPacketOut(datapath=dp,
buffer_id=msg.buffer_id, in_port=in_port, actions=[], data=None)
                    dp.send_msg(out)
                    return
            else:
                # 第一次收到报文
                self.sw[(dpid, src, arp_dst_ip)] = in_port
                # 泛洪在后面的 self-learning 中执行

        # self-learning
        # you need to code here to avoid the direct flooding
        # having fun
        # :)
        # just code in mission 1
        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.OFPActionOutput(out_port)]
```

```
        if out_port != ofp.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(dp, 1, match, actions, hard_timeout=5)

        data = None
        if msg.buffer_id == ofp.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,
                                  in_port=in_port,         actions=actions,
data=data)
        dp.send_msg(out)
```

Broadcast_Loop2.py

```
from os_ken.base import app_manager
from os_ken.controller import ofp_event
from      os_ken.controller.handler      import      MAIN_DISPATCHER,
CONFIG_DISPATCHER
from os_ken.controller.handler import set_ev_cls
from os_ken.ofproto import ofproto_v1_3
from os_ken.lib.packet import packet
from os_ken.lib.packet import ethernet
from os_ken.lib.packet import arp
from os_ken.lib.packet import ether_types

ETHERNET = ethernet.ethernet.__name__
ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__


class Switch_Dict(app_manager.OSKenApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Switch_Dict, self).__init__(*args, **kwargs)
        self.sw = {}  # (dpid, src_mac, dst_ip)=>in_port, you may use it
in mission 2
        self.mac_to_port = {}  # A global data structure to save the mapping

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0,
hard_timeout=0):
        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.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_BUFFER)]
        self.add_flow(dp, 0, match, actions)

    @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

        # the identity of switch
        dpid = dp.id
        self.mac_to_port.setdefault(dpid, {})
        # the port that receive the packet
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        if eth_pkt.ethertype == ether_types.ETH_TYPE_LLDP:
            return
        if eth_pkt.ethertype == ether_types.ETH_TYPE_IPV6:
            return

        # get the mac
        dst = eth_pkt.dst
        src = eth_pkt.src

        # get protocols
```

```
        header_list = dict((p.protocol_name, p) for p in pkt.protocols if
type(p) != str)

        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

        # self-learning
        # you need to code here to avoid the direct flooding
        # having fun
        # :)
        # just code in mission 1

        # 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.OFPActionOutput(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)
```