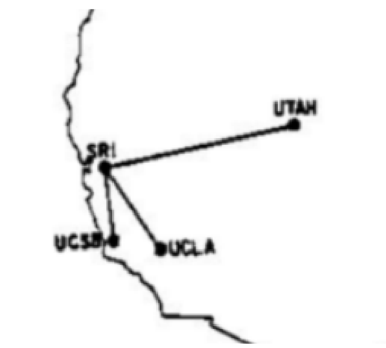
SDN-Lab2

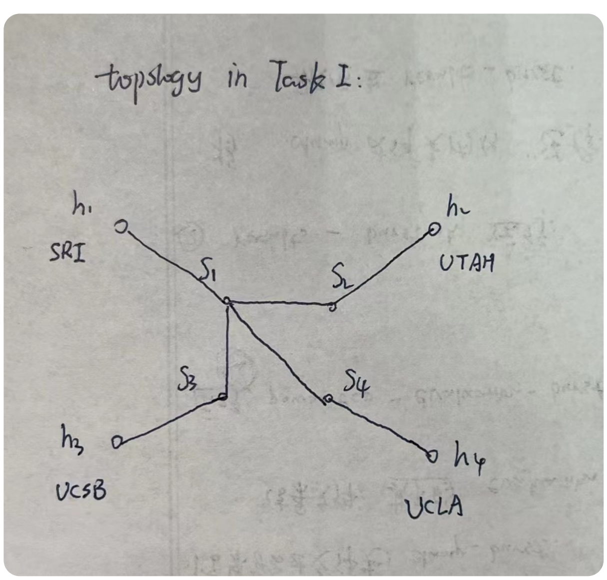
本实验工作目录：

```
❯ pwd
/home/root-hbx/CS

❯ ls
 __pycache__   Broadcast_Loop.py  gmon.out   Learning_Switch.py   ryu-xjt
```

1. 自学习交换机

1.1 问题说明





情景：

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。

前文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。

采用 `ryu` 作为远程控制器 (`remote controller`)，开启方式：

```
ryu-manager --observe-links Broadcast_Loop.py
```

网络拓扑为 `topo_1969_1.py` ，启动方式：

```
sudo mn --custom topo_1969_1.py --topo generated --controller remote
```

1.2 实验代码

`task1_selfLearningSwitch.py`:

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

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

```python
'''
an adaptation in Broadcast_Loop.py
'''

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

    def __init__(self, *args, **kwargs):
        super(Switch_Dict, self).__init__(*args, **kwargs)
        self.sw = {}
        # topo: src--in_port--|Sw = dpid|--aim_port--dst
        # mapping-1: portIn = mac_to_port[Sw][src]
        # mapping-2: portOut = mac_to_port[Sw][dst]

        self.mac_to_port = {}

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0, ha
        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_
        self.add_flow(dp, 0, match, actions)

    # what we actually done:
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        # method-def (Packet-in): self -- An event

        msg = ev.msg # message object
        dp = msg.datapath # data object
        ofp = dp.ofproto  # constants about OpenFlow
```

```python
        parser = dp.ofproto_parser # parser to construct and analysis OpenFlo

        # the identity of switch
        dpid = dp.id # Sw.id
        self.mac_to_port.setdefault(dpid, {}) # add {empty} into Sw.

        # 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 typ
        if dst == ETHERNET_MULTICAST and ARP in header_list:
            # need to code here to avoid broadcast loop to finish mission 2
            # this part can be passed here
            pass

        # self-learning
        # src---in_port---|Sw = dpid|---aim_port---dst

        self.mac_to_port[dpid][src] = in_port # mapping-1
        currentSw = self.mac_to_port[dpid]
        flag = 0 # judge if it's a new mapping

        # the logic process of OpenFlow Controller

        if dst in currentSw:
            aim_port = self.mac_to_port[dpid][dst] # mapping-2
        else:
            aim_port = ofp.OFPP_FLOOD # pre-flooding this packet to all nodes
            flag = 1

        # sending to which port (specific one / flooding) depends on "aim_por
        actions = [parser.OFPActionOutput(aim_port)]

        # a new mapping, add the flow table to switch
        if (flag):
```

```
        # create a matching condition, only the pkt (portIn: in_port, dst
        portMacPair = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        # add this matching into flow table
        self.add_flow(dp, 10, portMacPair, actions) # avoid flow table sh

    '''
    After receiving Pack-In,
    controller will send Packet-Out to Switch and give orders to the spec
    '''

    data = None
    # for flow tables, in-time send is necessary
    if msg.buffer_id == ofp.OFP_NO_BUFFER:
        # if there's no packetBuffer in Switch, it implies:
        #   this packet is sending to Controller directly!
        #   and we need to copy the "data in packet" into dataMessage
        #   for it will be loaded into "outPut" towards Sw.
        data = msg.data

    outPut = parser.OFPPacketOut(
        datapath=dp, # towards the Sw. who is sending Packet-In message
        buffer_id=msg.buffer_id, # the Buffer ID of the packetNum
        in_port=in_port, # the receiving portNum
        actions=actions,
        data=data,
    )

    dp.send_msg(outPut)
```

1.3 实验分析

原理

当前方案中给出的"自学习"，本质上是根据"映射关系"进行查询：

- 当交换机上报一个Packet In消息给控制器后，控制器检查该消息携带的是否为Ethernet类型报文
  - 如果是：提取出 eth_src 和 portIn，建立映射关系
  - 反之：不建立

- 后续Ethernet类型报文进入时，控制器检测是否已学习过该报文中 `dst_mac` 对应的 `portIn`
  - 如果是：建立对应关系,并下发FlowTable
  - 反之：说明当前是ARP，立即洪泛

安装控制器并进行对接

```
ryu-manager --observe-links task1_selfLearningSwitch.py
```

`pingall` 结果

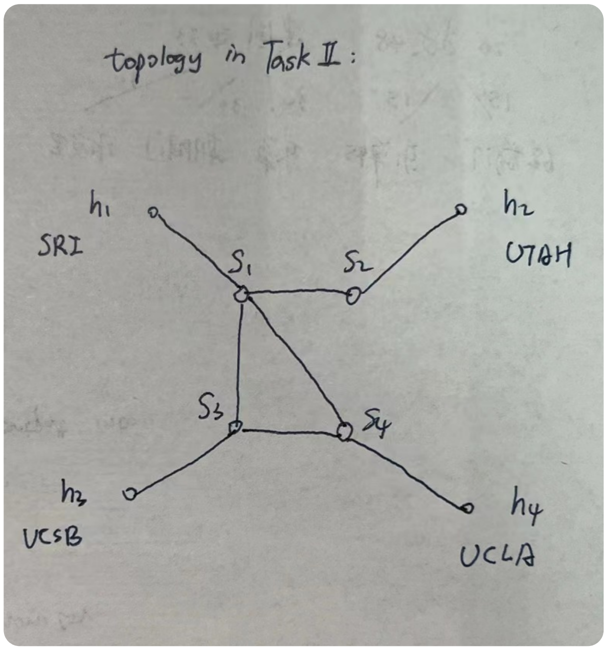
```
sudo mn --custom topo_1969_1.py --topo generated --controller remote
......

mininet> pingall
...
```

查看各Switch流表

```
sudo ovs-ofctl dump-flows s1 # SRI
sudo ovs-ofctl dump-flows s2 # UCLA
sudo ovs-ofctl dump-flows s3 # UCSB
sudo ovs-ofctl dump-flows s4 # UTAH
```

2. 避免环路广播

2.1 问题说明



情景:

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：

1. 当序号为 dpid 的交换机从 portIn 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 ;
2. 下一次该交换机收到同一 (src_mac, dst_ip) 但 in_port 不同的 ARP Request 数据包时直接丢弃即可

2.2 实验代码

task2_BreakLoop.py:

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import arp
from ryu.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.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Switch_Dict, self).__init__(*args, **kwargs)
        self.sw = {}

        self.arp_in_port = {}
        self.mac_to_port = {}

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0, ha
        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_
```

```
        self.add_flow(dp, 0, match, actions)

    # what we actually done:
    @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, {})
        self.arp_in_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 (head part of packet)
        header_list = dict((p.protocol_name, p) for p in pkt.protocols if typ

        # ARP Loop Processing

        if dst == ETHERNET_MULTICAST and ARP in header_list:
            # a ARP packet
            arp_pkt = pkt.get_protocol(arp.arp)

            if arp_pkt and arp_pkt.opcode == arp.ARP_REQUEST:
                # arp_pkt.opcode represents the option-code of ARP_Packet
                # option-code: the action message need to transfer back to Sw
                # arp.ARP_REQUEST: Sw. is waiting for {"the MAC Addr." which

                req_dst_ip = arp_pkt.dst_ip # the IP Addr. of the dstNode
                arp_src_mac = arp_pkt.src_mac # the MAC Addr. of the srcNode

                '''
```

```
        ARP: need the dstMAC according to dstIP

        (srcMAC, dstIP, Sw.pid)
        - if srcMAC is same && dstIP is same
            - if portIn is same: pass
            - else: "what we have used" => discard
        '''

        # get mac in mapping
        if arp_src_mac in self.arp_in_port[dpid]:
            # if the srcMAC is recorded

            # get IP in mapping
            if req_dst_ip in self.arp_in_port[dpid][arp_src_mac]:
                # if the dstIP is also recorded

                if in_port != self.arp_in_port[dpid][arp_src_mac][req
                    match = parser.OFPMatch (
                        in_port = in_port,
                        arp_op = arp.ARP_REQUEST,
                        arp_tpa = req_dst_ip,
                        arp_sha = arp_src_mac,
                    )

                    actions = []

                    # prio than self-learning
                    self.add_flow(dp, 20, match, actions)

                    outPut = parser.OFPPacketOut(
                        datapath = dp,
                        buffer_id = msg.buffer_id,
                        in_port = in_port,
                        actions = [],
                        data = None
                    )

                    dp.send_msg (outPut)

            # no req_dst_ip in mapping
            else:
                # record the dstIP and mapping(srcMAC, dstIP, portIn)
                self.arp_in_port[dpid][arp_src_mac].setdefault(req_ds
                self.arp_in_port[dpid][arp_src_mac][req_dst_ip] = in_
        # no arp_src_mac in mapping
        else:
```

```
                # record the srcMAC and mapping(srcMAC, dstIP, portIn)
                self.arp_in_port[dpid].setdefault(arp_src_mac, {})
                self.arp_in_port[dpid][arp_src_mac][req_dst_ip] = in_port

        # self-learning

        self.mac_to_port[dpid][src] = in_port
        currentSw = self.mac_to_port[dpid]
        flag = 0 # judge if it's a new mapping

        if dst in currentSw:
            aim_port = self.mac_to_port[dpid][dst]
        else:
            aim_port = ofp.OFPP_FLOOD
            flag = 1

        actions = [parser.OFPActionOutput(aim_port)]

        if (flag):
            # a new mapping, add the flow table to switch
            portMacPair = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(dp, 10, portMacPair, actions) # avoid flow table sh

        data = None
        # for flow tables, in-time send is necessary
        if msg.buffer_id == ofp.OFP_NO_BUFFER:
            data = msg.data
        outPut = parser.OFPPacketOut(
            datapath=dp,
            buffer_id=msg.buffer_id,
            in_port=in_port,
            actions=actions,
            data=data,
        )
        dp.send_msg(outPut)
```

2.3 实验分析

原理

当前方案给出的本质上还是 "映射记录,重复舍弃"

当交换机上报一个Packet In消息给控制器后，控制器检查最基本条件：

1. 该消息携带的是否为 `Ethernet` 类型报文
2. 该数据包是否携带 `ARP` 报头

如上述条件满足,则进入映射设计：

- 我们采取针对某一个交换机考察，建立映射 `(srcMAC, dstIP, portIn)`
- 具体来说，设计：`(srcMAC, dstIP, currentSw) = portIn_Sw`

针对ARP的专门处理：

- 如果 $srcMAC\ is\ same\ \&\&\ dstIP\ is\ same$
  - 如果 `portIn` 不相同，说明是经环路运作后的"重复包"，直接丢弃即可
  - 反之，`pass`即可

安装控制器并进行对接

```
ryu-manager --observe-links task2_BreakLoop.py
```

pingall结果

```
sudo mn --custom topo_1969_2.py --topo generated --controller remote
......

mininet> pingall
...
```

查看各Switch流表

```
sudo ovs-ofctl dump-flows s1 # SRI
sudo ovs-ofctl dump-flows s2 # UCLA
sudo ovs-ofctl dump-flows s3 # UCSB
sudo ovs-ofctl dump-flows s4 # UTAH
```

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上述环路广播问题

3.1 设计原理

我们给任一个 ARP Record 赋予一定的生命周期，在这个时间范围内：

- 如果出现了多个 ARP Request报文，则只记录第一个报文，忽略后面的报文

查询资料可知：一个ARP Record的生存期大约为 1min (60s)

3.2 代码实现

task3_arp_sdn.py:

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

ETHERNET = ethernet.ethernet.__name__
ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__
ARP_TIMEOUT = 60


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

    def __init__ (self, *args, **kwargs):
        super(Switch_Dict, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
```

```
        # recording the latest TimeNode (a ARP Request Packet coming in)
        self.latest_stamp = {} # dpid, src_ip, dst_ip -> timestamp

    def add_flow (self, datapath, priority, match, actions, idle_timeout=0, h
        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

        '''
        创建一个动作列表，表示将数据包发送到控制器:
        - ofp.OFPP_CONTROLLER表示数据包将发送到控制器, ofp.OFPCML_NO_BUFFER表示不
        '''
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_
        '''
        调用add_flow方法向交换机添加流表规则:
        - 这条规则的优先级为0, 表示最低优先级, 匹配所有数据包, 并将它们发送到控制器
        '''
        self.add_flow(dp, 0, match, actions)

        # let switch send arp to controller
        '''
        创建一个匹配条件, 匹配目的MAC地址为广播地址 (ETHERNET_MULTICAST) 的数据包。
        '''
        match = parser.OFPMatch(eth_dst=ETHERNET_MULTICAST)
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_

        # a little higher priority to make switch must send arp to controller
```

```
        '''
        确保交换机会将ARP请求数据包发送到控制器
        '''
        self.add_flow(dp, 1, match, actions)

    # what we actually done
    @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, {})
        self.latest_stamp.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
        eth_dst = eth_pkt.dst
        eth_src = eth_pkt.src

        self.logger.info(
            "Pkt-in: SW(%s) Src(%s) Dst(%s) InPort(%s)", dpid, eth_src, eth_d
        )

        if eth_src not in self.mac_to_port[dpid]:
            # recording srcMAC and mapping(arcMAC, portIn)
            self.mac_to_port[dpid].setdefault(eth_src, {})
            self.mac_to_port[dpid][eth_src] = in_port

        # ARP request
        arp_pkt = pkt.get_protocol(arp.arp)
        currentTime = ev.timestamp

        if arp_pkt and arp_pkt.opcode == arp.ARP_REQUEST:
            # get the ip
```

```
            arp_src_ip = arp_pkt.src_ip
            arp_dst_ip = arp_pkt.dst_ip

            # arp for the first time
            if arp_src_ip not in self.latest_stamp[dpid]:
                self.latest_stamp[dpid].setdefault(arp_src_ip, {})
                self.latest_stamp[dpid][arp_src_ip][arp_dst_ip] = currentTime
            # another dst ip, update, too
            elif arp_dst_ip not in self.latest_stamp[dpid][arp_src_ip]:
                self.latest_stamp[dpid][arp_src_ip].setdefault(arp_dst_ip, {}
                self.latest_stamp[dpid][arp_src_ip][arp_dst_ip] = currentTime
            # arp for the second time
            elif currentTime - self.latest_stamp[dpid][arp_src_ip][arp_dst_ip
                print("duration is not long enough!")
                return

        # self-learning
        if eth_dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][eth_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=eth_dst)
            self.add_flow(dp, 10, match, actions)

        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=ac
        )
        dp.send_msg(out)
```