计算机网络与 SDN 技术：Mininet 大作业实验报告

姓名：吴博闻　学号：10245102410

2025 年 12 月 20 日

目录

1 Part 1: Mininet 入门与拓扑构建

1.1 实验目的

1. 掌握 Mininet 网络仿真器的基本操作。
2. 使用 Python 脚本构建自定义网络拓扑（星型拓扑）。
3. 使用 Mininet CLI 工具测试网络连通性与带宽。

1.2 拓扑设计与代码实现

本部分构建了一个包含 1 个交换机（s1）和 4 个主机（h1-h4）的星型拓扑。所有主机均连接至 s1。

修改文件：`project-1/topos/part1.py`

```
class part1_topo(Topo):
    def build(self):
        # 1. 添加交换机 s1
        s1 = self.addSwitch('s1')

        # 2. 添加主机 h1, h2, h3, h4
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        h4 = self.addHost('h4')

        # 3. 添加链路：将所有主机连接到交换机 s1
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(h3, s1)
        self.addLink(h4, s1)
```

Listing 1: Part 1 拓扑构建代码

1.3 实验结果与分析

如图 1 所示：

1. **连通性测试 (Pingall)**：所有主机之间连通性良好，丢包率为 0% (0% dropped)，验证了拓扑链路建立正确。
2. **带宽测试 (Iperf)**：h1 与 h4 之间的 TCP 带宽测试显示约为 27.1 Gbits/sec，符合 Mininet 虚拟环境的预期性能。

3. **节点信息 (Dump)**：正确列出了 h1-h4 的 IP 地址及 s1 的接口信息。

```
vagrant@mnvm:~$ sudo python3 project-1/topos/part1.py
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> iperf h1 h4
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['27.1 Gbits/sec', '27.1 Gbits/sec']
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=30965>
<Host h2: h2-eth0:10.0.0.2 pid=30983>
<Host h3: h3-eth0:10.0.0.3 pid=30985>
<Host h4: h4-eth0:10.0.0.4 pid=30998>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None,s1-eth4:None pid=31003>
<Controller c0: 127.0.0.1:6653 pid=30958>
```

图 1: Part 1 实验结果截图 (Pingall, Iperf, Dump)

2 Part 2: 使用 POX 构建 SDN 防火墙

2.1 实验目的

实现一个基于 OpenFlow 的简单 Layer 2 防火墙。控制器需要下发流表规则，满足以下安全策略：

- 允许所有 ARP 流量通过（泛洪）。
- 允许所有 ICMP 流量通过（泛洪）。
- **丢弃**其他所有 IPv4 流量（如 TCP/UDP）。

2.2 控制器逻辑实现

修改文件：`project-1/pox/a1part2controller.py`

```
    # 规则 1: 允许并泛洪 ARP 流量 (dl_type = 0x0806)
    msg_arp = of.ofp_flow_mod()
    msg_arp.match.dl_type = 0x0806
    msg_arp.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    self.connection.send(msg_arp)

    # 规则 2: 允许并泛洪 ICMP 流量 (Priority=20)
    msg_icmp = of.ofp_flow_mod()
    msg_icmp.match.dl_type = 0x0800
    msg_icmp.match.nw_proto = 1
    msg_icmp.priority = 20
    msg_icmp.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    self.connection.send(msg_icmp)

    # 规则 3: 丢弃其他所有 IPv4 流量 (Priority=10, 无 Action 即丢弃)
    msg_drop = of.ofp_flow_mod()
    msg_drop.match.dl_type = 0x0800
    msg_drop.priority = 10
    self.connection.send(msg_drop)
```

Listing 2: Part 2 防火墙规则代码

2.3 实验结果与分析

如图 2 所示：

1. **Pingall 结果：**同一子网内主机（如 h1-h4）Ping 成功，证明 ICMP 规则生效且优先级正确（高于 Drop 规则）。不同子网不通是由于缺乏三层路由功能，符合预期。

2. **Iperf 结果：**`iperf h1 h4` 命令挂起（被中断），这是因为 Iperf 使用 TCP 协议。TCP 既不是 ARP 也不是 ICMP，因此匹配到了优先级为 10 的 `drop` 规则，数据包被丢弃。

3. **流表验证 (Dump-flows)：**

   - `arp actions=FLOOD`：处理了 ARP 包。
   - `priority=20,icmp actions=FLOOD`：处理了 Ping 包。
   - `priority=10,ip actions=drop`：处理了 TCP 包。

```
vagrant@mnvm:~$ sudo python3 ~/project-1/topos/part2.py
Unable to contact the remote controller at 127.0.0.1:6653
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X h4
h2 -> X h3 X
h3 -> X h2 X
h4 -> h1 X X
*** Results: 66% dropped (4/12 received)
mininet> iperf h1 h4
*** Iperf: testing TCP bandwidth between h1 and h4
^C
Interrupt
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=252.837s, table=0, n_packets=14, n_bytes=588, arp actions=FLOOD
 cookie=0x0, duration=252.837s, table=0, n_packets=14, n_bytes=1036, priority=10,ip actions=drop
 cookie=0x0, duration=252.837s, table=0, n_packets=8, n_bytes=784, priority=20,icmp actions=FLOOD
mininet>
```

图 2: Part 2 实验结果截图 (Ping 成功, Iperf 失败, 流表确认)

3 Part 3: 复杂网络与静态路由策略

3.1 实验目的

模拟企业网络环境，包含核心交换机（cores21）、楼层交换机和数据中心。需要在核心交换机上实现：

- **静态路由**：根据目的 IP 子网转发到特定端口。
- **访问控制列表 (ACL)**：
  1. 禁止外部不可信主机 `hnotrust1` 向内网发送 ICMP。
  2. 禁止 `hnotrust1` 访问服务器 `serv1` 的所有 IP 流量。

3.2 控制器逻辑实现

修改文件：`project-2/pox/a2part1controller.py`

```
def cores21_setup(self):
    # --- 安全规则 (Security Rules, Priority=20) ---

    # Rule 1: 禁止 hnotrust 向内网发送 ICMP
    msg = of.ofp_flow_mod()
    msg.priority = 20
    msg.match.dl_type = 0x0800 # IPv4
    msg.match.nw_proto = 1     # ICMP
    msg.match.nw_src = IPS["hnotrust"]
    self.connection.send(msg) # Drop

    # Rule 2: 禁止 hnotrust 向 serv1 发送任何 IP 流量
    msg = of.ofp_flow_mod()
    msg.priority = 20
    msg.match.dl_type = 0x0800 # IPv4
    msg.match.nw_src = IPS["hnotrust"]
    msg.match.nw_dst = IPS["serv1"]
    self.connection.send(msg) # Drop

    # --- 转发规则 (Forwarding Rules, Priority=10) ---
    # 示例: 转发到 h10 所在子网 (Port 1)
    msg = of.ofp_flow_mod()
    msg.priority = 10
    msg.match.dl_type = 0x0800
    msg.match.nw_dst = SUBNETS["h10"]
    msg.actions.append(of.ofp_action_output(port=1))
    self.connection.send(msg)
```

```
28      # ... (其他端口同理)
```

Listing 3: Part 3 核心交换机配置代码

3.3 实验结果与分析

如图 3 所示：

1. **Pingall 测试：**丢包率为 40%。
   - `hnotrust1` 无法 Ping 通任何主机，且任何主机 Ping `hnotrust1` 也失败（显示为 X）。这是因为双向的 ICMP 交互中，源自 `hnotrust1` 的请求或回复包均被 Rule 1 丢弃。
   - 内网主机（h10, h20 等）之间互通，说明静态路由配置正确。
2. **TCP 连通性 (hnotrust1 -> h10)：**测试成功（28.3 Gbits/sec）。说明只封禁了 ICMP，未封禁 TCP，符合“允许正常通信”的要求。
3. **服务器保护 (hnotrust1 -> serv1)：**测试失败（挂起）。说明 Rule 2 生效，成功保护了服务器。

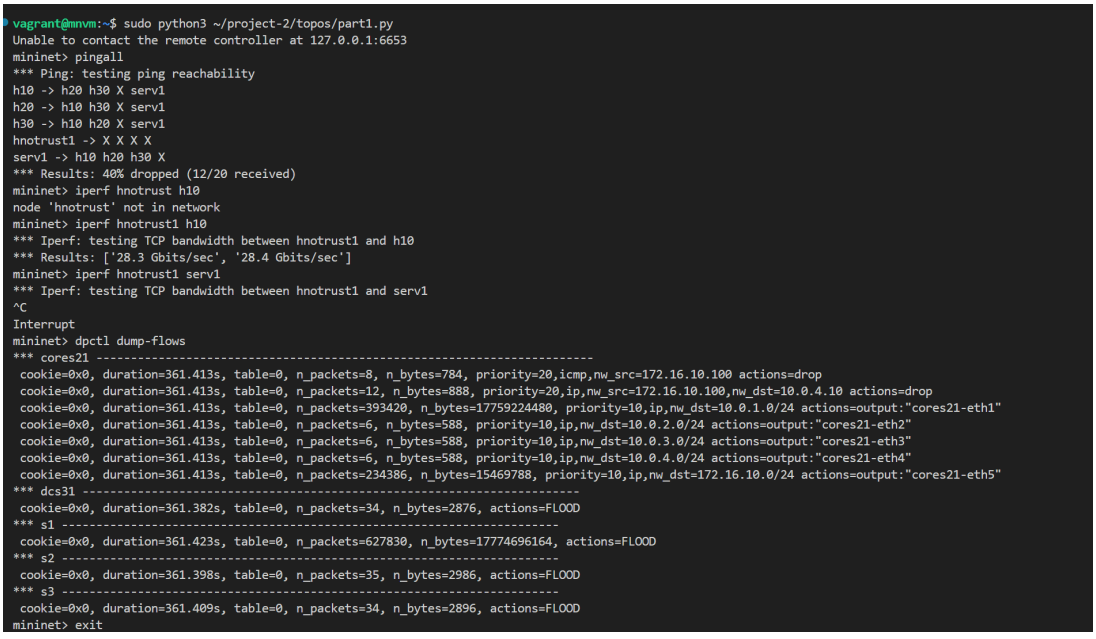

图 3: Part 3 实验结果截图 (ACL 生效验证)

4 Part 4: 学习型路由器 (Learning Router)

4.1 实验目的

实现一个真正的三层路由器。与 Part 3 不同，路由器不能使用静态路由表，而必须：

- 响应主机的 ARP 请求（扮演网关角色）。
- 通过 ARP 报文和 IP 报文动态学习 IP 与 (MAC, Port) 的映射关系。
- 根据学习到的 ARP 表进行数据包转发，并在转发时修改源/目的 MAC 地址。
- 依然保留 Part 3 中的安全策略。

4.2 核心代码实现

修改文件：project-2/pox/a2part2controller.py

1. ARP 响应与动态学习机制：

```
# 1. 动态学习 (Dynamic Learning)
# 只要收到包，就记录源 IP 对应的 (MAC, Port)
if src_ip is not None:
    self.arp_table[src_ip] = (packet.src, event.port)

# 2. 处理网关 ARP 请求 (Gateway ARP Reply)
if packet.type == ethernet.ARP_TYPE:
    if arp_payload.opcode == arp.REQUEST and arp_payload.protodst in
    GATEWAYS:
        # 构造 ARP Reply，源 MAC 为路由器虚拟 MAC
        reply = arp()
        reply.opcode = arp.REPLY
        reply.hwsrc = ROUTER_MAC
        reply.protosrc = arp_payload.protodst
        # ... 发送 Reply ...
```

Listing 4: ARP 处理与学习逻辑

2. IP 路由与流表安装：

```
# 3. 处理 IP 路由
if packet.type == ethernet.IP_TYPE:
    dst_ip = packet.payload.dstip
    # 查表
    if dst_ip in self.arp_table:
        dst_mac, dst_port = self.arp_table[dst_ip]

        # 安装流表 (Priority=15)
```

```
        msg = of.ofp_flow_mod()
        msg.match.nw_dst = dst_ip
        # 修改源 MAC 为路由器，目的 MAC 为下一跳
        msg.actions.append(of.ofp_action_dl_addr.set_src(ROUTER_MAC))
        msg.actions.append(of.ofp_action_dl_addr.set_dst(dst_mac))
        msg.actions.append(of.ofp_action_output(port=dst_port))
        self.connection.send(msg)

        # 转发当前包
        # ... (Packet Out) ...
    else:
        # 目的未知，主动发送 ARP 请求探测 (代码略)
        pass
```

Listing 5: IP 路由转发逻辑

4.3 实验结果与分析

如图 4 所示：

1. **Pingall 稳定性：**经过多次 Pingall 后，结果稳定在 **40% Dropped**。
   - `hnotrust1` 行全为 X：符合防火墙规则（禁止 ICMP）。
   - 内网主机及服务器之间互通：说明路由器成功学习到了各主机的 MAC 地址和端口，并正确改写 MAC 头进行了转发。
2. **流表验证 (Dump-flows)：**
   - **安全规则:**可以看到 `icmp...actions=drop` 拦截了 23 个包,`ip...dst=10.0.4.10 actions=drop` 拦截了 5 个包（对应失败的 Iperf 测试）。
   - **路由规则:**可以看到大量 `priority=15` 的规则,且包含 `mod_dl_src` 和 `mod_dl_dst` 动作。例如：

     ```
     nw_dst=10.0.1.10 actions=mod_dl_src:00:00:00:00:00:01,
                              mod_dl_dst:00:00:00:00:00:01,output:"cores21-eth1"
     ```

     这证明控制器成功下发了修改 MAC 地址的规则，实现了三层路由器的功能。
3. **业务测试：**`iperf hnotrust1 h10` 成功（27.9 Gbps），证明正常 TCP 流量未受影响且路由正确。

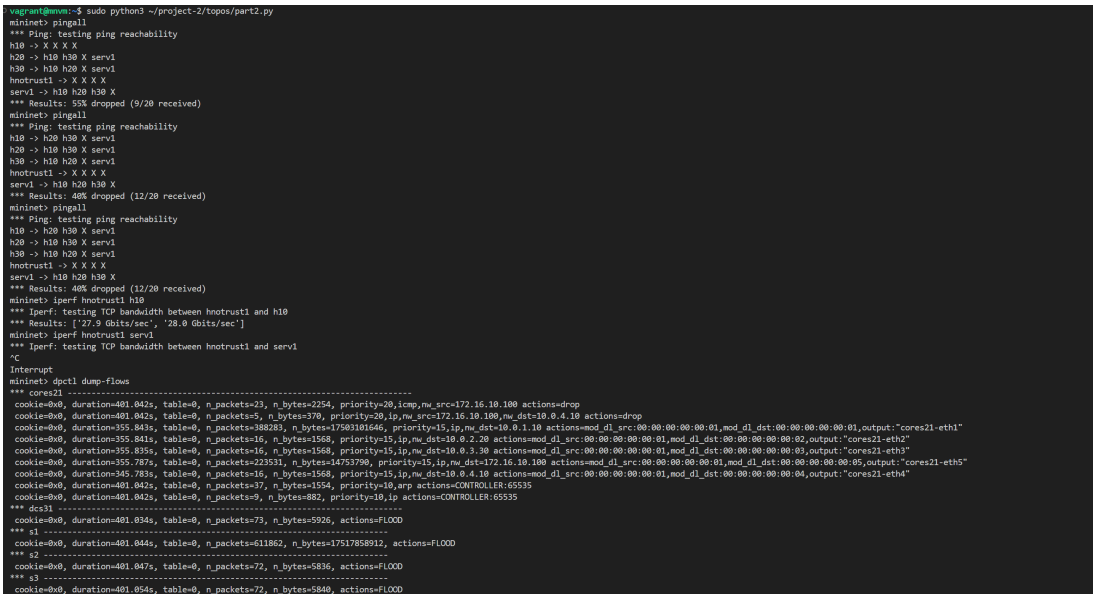

图 4: Part 4 实验结果截图 (学习型路由器验证)

5 实验总结

本次实验通过 Mininet 和 POX 控制器，从基础的拓扑构建进阶到实现具备防火墙和动态学习功能的三层路由器。

1. 掌握了 Mininet Python API 的使用。

2. 深入理解了 OpenFlow 协议中的匹配（Match）、动作（Action）和优先级（Priority）机制。

3. 成功实现了 ARP 代理响应与基于 ARP 表的动态路由转发，并在软件定义网络中复现了传统路由器的核心行为。