IPv4 Router: Forwarding Packets

张闻曦（231880099、1019980666@qq.com）

摘　要:　在这次实验中，我们设计并实现了一个基于静态路由的 IPv4 路由器，支持分组转发、ARP 解析和超时重传功能。路由器通过读取 forwarding_table.txt 和直连接口构建转发表，利用最长前缀匹配算法确定下一跳，并通过维护 ARP 表缓存 MAC 地址以减少重复查询。对于未知 MAC 地址，路由器发送 ARP 请求并缓存待转发分组，若 5 次重传后仍未收到响应则丢弃分组。实验使用 Switchyard 框架验证了基础转发功能，并在 Mininet 中通过 ICMP 测试和 Wireshark 抓包确认了实际转发效果。实验深入理解了 IP 转发与链路层协议的交互，揭示了路由表设计与 ARP 优化对性能的影响。

1　实验名称：Forwarding Packets

2　实验目的

实现基于静态路由表的 IPv4 分组转发功能:

1. 接收并转发到达链路并且目的地是其他主机的数据包。转发过程中的一部分是在转发表中,通过最长前缀匹配算法查找转发表，进行地址查询。

2. 为没有已知以太网 MAC 地址的 IP 地址发送 ARP 请求。路由器通常需要向其他主机发送数据包，需要以太网 MAC 地址来完成这个过程。处理 ARP 请求与响应，完成 IP 地址到 MAC 地址的解析。

本次实验在 Linux 环境中进行，使用 WSL2 搭载的 Ubuntu-22.04。

3 实验内容

3.1 系统架构

```
Host (Sender)
↓
Router (Forwarding + ARP Handling)
↓
Host (Receiver)
```

路由器功能：

从 forwarding_table.txt 和直连接口构建转发表。

对非本机 IP 的分组执行转发（TTL 减 1，匹配下一跳）。

发送 ARP 请求并缓存响应结果。

3.2 Router关键实现

3.2.1 转发表构建

ForwardingTableEntry:

封装路由表条目，包含四个表项: dest(目标网络), mask(子网掩码), gateway(下一跳地址), interface(出口接口)。同时，重载 < 运算符以实现按前缀长度排序。

```python
class ForwardingTableEntry:  2用法  新*
    def __init__(self, dest, mask, gateway, interface):  新*
        self.dest = dest
        self.mask = mask
        self.gateway = gateway
        self.interface = interface
        self.prefixnet = IPv4Network("{}/{}".format( *args: dest, mask), strict=False)

    def __lt__(self, other):  新*
        return self.prefixnet.prefixlen > other.prefixnet.prefixlen
```

build_forwarding_table():

根据自己的 interface(自己的子网)和 forwarding_table.txt 建立 forwarding table。计算前缀长度，对 forwarding table 进行排序，前缀长的放在前面，从前向后遍历时第一个匹配到的就是最长前缀匹配。

```python
def build_forwarding_table(self):  1个用法  新*
    for interface in self.interfaces:
        self.forwarding_table.append(
            ForwardingTableEntry(interface.ipaddrs, interface.netmask,  gateway: None, interface.name))
    with open("forwarding_table.txt") as f:
        for line in f:
            dest, mask, gateway, interface = line.strip().split()
            self.forwarding_table.append(ForwardingTableEntry(dest, mask, gateway, interface))
    self.forwarding_table.sort()
```

3.2.2 IPV4 Packet

handle_ipv4_packet(self, recv):

1.包检查：检查数据包的总长度是否等于以太网头的长度加上 IPv4 头中指定的总长度。

2.ip 判断：如果是发给自己 ip 的包，那么不处理。

3.匹配路由：在 forwarding table 里面查找目标 IP 的最长前缀匹配条目对应的表项，如果没有就不处理。

4.确定下一跳：有 gateway 即为下一跳，则下一跳是目标 IP 本身。

5.MAC 缓存检查：若 ARP 表中有下一跳 IP 的 MAC 地址，直接封装以太网头部并转发；若不存在，发送 ARP 请求来查询(如果已发过，不再次发送) 并缓存待转发分组，等待响应后重发。

```
def handle_ipv4_packet(self, recv):  1 个用法  新 *
    timestamp, ifaceName, packet = recv
    ipv4 = packet.get_header(IPv4)
    eth = packet.get_header(Ethernet)
    dst_ip = ipv4.dst
    if len(eth) + ipv4.total_length != packet.size():
        return
    if dst_ip in self.ip_list:
        return
    entry = self.get_forwarding_entry(dst_ip)
    if entry is None:
        return

    next_hop_ip = dst_ip if entry.gateway is None else ip_address(entry.gateway)
    next_hop_mac = self.arp_table.get(next_hop_ip)

    if next_hop_mac is not None:
        packet[Ethernet].src = self.net.interface_by_name(entry.interface).ethaddr
        packet[Ethernet].dst = next_hop_mac
        packet[IPv4].ttl -= 1
        self.net.send_packet(entry.interface, packet)

    else:
        if next_hop_ip not in self.waiting_ip.keys():
            self.waiting_ip[next_hop_ip] = (time.time(), 1)
            arp_request = create_ip_arp_request(self.net.interface_by_name(entry.interface).ethaddr,
                                                self.net.interface_by_name(entry.interface).ipaddrs, next_hop_ip)
            self.net.send_packet(entry.interface, arp_request)
        if next_hop_ip not in self.waiting_packet.keys():
            self.waiting_packet[next_hop_ip] = []
        self.waiting_packet[next_hop_ip].append(packet)
```

3.2.3 ARP Packet

handle_arp_packet(self, recv):

1.ip 判断：如果不是发给自己 ip 的包，那么丢弃。

2.对于 arp 请求：记录请求方的 (IP, MAC) 映射，加入自己的 arptable。发回自己的 MAC 地址。

3.对于 arp 回复：如果源地址是广播地址，那么这个数据包可能是错误的或者被篡改过，应该被丢弃；如果原地址不是广播地址，发送等待队列中缓存的待相应 ip 转发分组并更新 ARP 表删除对应项目。（对于 arp 回复，如果源地址是广播地址，那么这个数据包可能是错误的或者被篡改过，应该被丢弃）

```
def handle_arp_packet(self, recv):  1个用法  新*
    timestamp, ifaceName, packet = recv
    arp = packet.get_header(Arp)
    eth = packet.get_header(Ethernet)
    src_ip = arp.senderprotoaddr
    src_mac = arp.senderhwaddr
    dst_ip = arp.targetprotoaddr
    if dst_ip not in self.ip_list:
        return

    if arp.operation == ArpOperation.Request:
        self.arp_table[src_ip] = src_mac
        arp_reply = create_ip_arp_reply(self.net.interface_by_ipaddr(dst_ip).ethaddr, src_mac, dst_ip, src_ip)
        self.net.send_packet(ifaceName, arp_reply)

    elif arp.operation == ArpOperation.Reply:
        if eth.src == 'ff:ff:ff:ff:ff:ff':
            return
        self.arp_table[src_ip] = src_mac
        if src_ip in self.waiting_ip.keys():
            for packet in self.waiting_packet[src_ip]:
                packet[Ethernet].src = self.net.interface_by_name(ifaceName).ethaddr
                packet[Ethernet].dst = src_mac
                packet[IPv4].ttl -= 1
                self.net.send_packet(ifaceName, packet)
            del self.waiting_packet[src_ip]
            del self.waiting_ip[src_ip]
```

3.2.4 包处理

handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):

1.ip 判断：如果既不是发给自己的包，也不是广播的包，那么不处理。

2.包类型判断：分 arp 和 ipv4 两种情况处理，其余情况不处理

```
def handle_packet(self, recv: switchyard.llnetbase.ReceivedPacket):  1 个用法  新 *
    timestamp, ifaceName, packet = recv
    arp = packet.get_header(Arp)
    eth = packet.get_header(Ethernet)
    ipv4 = packet.get_header(IPv4)

    if eth.dst != self.net.interface_by_name(ifaceName).ethaddr and eth.dst != 'ff:ff:ff:ff:ff:ff':
        return
    if eth.ethertype != EtherType.ARP and eth.ethertype != EtherType.IPv4:
        return

    if arp:
        self.handle_arp_packet(recv)
        return
    if ipv4:
        self.handle_ipv4_packet(recv)
        return
```

3.2.5 超时处理

handle_timeout(self):

在包和包的间隙，等待下一个包时进行 timeout 的检查：遍历所有待解析的 IP（waiting_ip 字典），检查是否超时（1 秒）；若超时且重试次数 <5，重新发送 ARP 请求；否则清除缓存，删除所有相应包。

```
def handle_timeout(self):  1 个用法  新 *
    for ip in list(self.waiting_ip.keys()):
        timestamp = self.waiting_ip[ip][0]
        retries = self.waiting_ip[ip][1]
        if time.time() - timestamp > 1:
            if retries >= 5:
                del self.waiting_ip[ip]
                del self.waiting_packet[ip]
            else:
                self.waiting_ip[ip] = (time.time(), retries + 1)
                arp_request = create_ip_arp_request(
                    self.net.interface_by_name(self.get_forwarding_entry(ip).interface).ethaddr,
                    self.net.interface_by_name(self.get_forwarding_entry(ip).interface).ipaddrs,
                    ip
                )
                self.net.send_packet(self.get_forwarding_entry(ip).interface, arp_request)
```

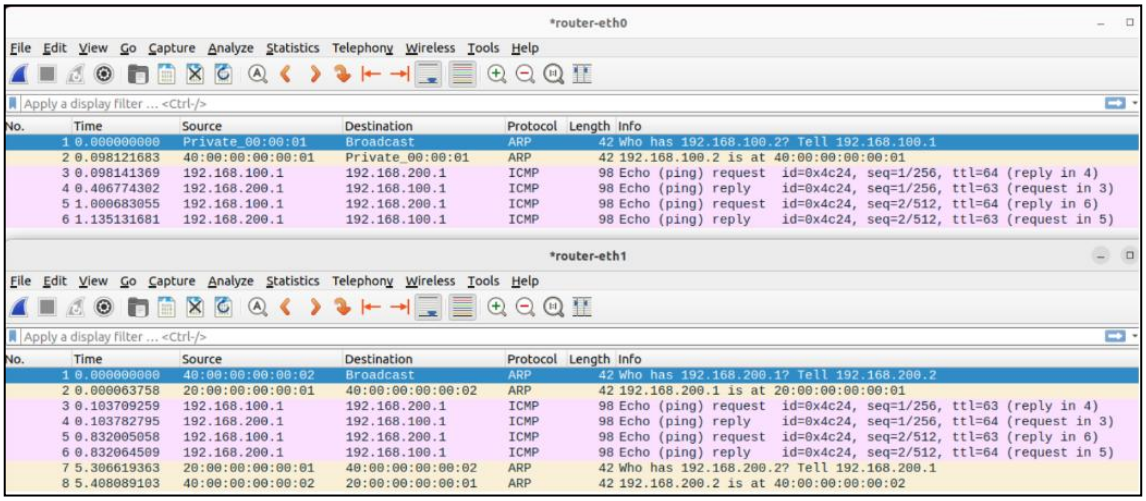
4 实验结果

4.1.1 Test

```
1196Ping request from 31.0.5.1 should arrive on eth5
1197Ping request from 31.0.5.1 should arrive on eth5
1198Router should not do anything
1199Ping request from 31.0.6.1 should arrive on eth6
1200Ping request from 31.0.6.1 should arrive on eth6
1201Ping request from 31.0.6.1 should arrive on eth6
1202Ping request from 31.0.6.1 should arrive on eth6
1203Ping request from 31.0.6.1 should arrive on eth6
1204Ping request from 31.0.6.1 should arrive on eth6
1205Router should not do anything
1206Bonus: V2FybWluZyB1cA==
1207Bonus: V2FybWVkIHVw
1208Bonus: V2h1dCBkJyB5YSBob3BlIHQnIGZpbmQgaGVyZT8=
1209Bonus: SGFsZndheQ==
1210Bonus: Tm90aGluJyBmb3IgeWEgdCcgZmluZCBoZXJlIQ==
1211Bonus: Q29uZ3JhdHMh


All tests passed!
```

4.1.2 Mininet



5 实验总结

5.1 遇到的问题与解决

在实验过程中因为实验手册未明确指出配置依赖的问题，在配置环境上花费了大量时间。

在实验过程中遇到 switchyard 的版本和兼容性问题，在一些方法上不得不作出修改。

5.2 实验启示

通过实现转发表和 ARP 交互，深入理解了路由器如何通过分层协议（IP + Ethernet）完成分组转发，为后续学习动态路由奠定基础。

致谢　在此，我们向对本文的工作给予支持和建议的老师与同学表示感谢。