南京大学本科生实验报告

课程名称：**计算机网络**　　任课教师：**李文中**

学院	计算机科学与技术系	专业（方向）	计算机科学与技术系
学号	181860109	姓名	吴润泽
Email	181860109@smail.nju.edu.cn	开始/完成日期	2020/4/8-2020/4/11

1. 实验名称：Forwarding Packets

2. 实验目的

1. 实现路由器的转发和路由功能
2. 理解路由器转发表的具体作用并利用
3. 掌握网络层基础的通信流程

3. 实验过程

Task 2 实现路由表项查询

a. 实现原理

1. 建立路由表：从给的 *forwarding_table* 文件和路由器端口所在的子网建立即可，对于路由器直接相连的子网，下一跳标记为 '#'；
2. 路由表匹配：遍历所有的路由表项，如果匹配则与当前匹配的最大前缀比较，如果该子网前缀长度大，则更新匹配到的路由表项；如果匹配均失败，返回空。

b. 代码编写

```
def built_router_table(self, filename):
    #通过文件进行读取
        myfile = open(filename, 'r')
        for line in myfile.readlines():
            line = line.strip().split()#去掉首尾额外空格，并将其分割
            if len(line) == 4:
                self.router_table.append((line[0], line[1], line[2],
line[3]))
        myfile.close()
    #通过遍历路由器接口读取
        for intf in self.net.interfaces():
            self.router_table.append(
                (str(intf.ipaddr), str(intf.netmask), '#',
str(intf.name)))

def match_subnet(self, dst_ip):
        maxlen,tar_route=0,None
        for item in self.router_table:
            subnet = IPv4Network(item[0] + '/' + item[1], False)
            if dst_ip in subnet:
                if maxlen < subnet.prefixlen:#更新当前匹配到的最长前缀
项
                    tar_route, maxlen = item, subnet.prefixlen
        return tar_route
```

Task3 实现转发数据包和发送Arp请求

模块1：IP数据报的接收和转发

a. 实现原理

1. 每当到达一个IP数据包，则将其目的地址与路由表匹配，如果匹配失败则丢弃；
2. 判断目的地址是否直接在某一子网中，如果是则下一跳地址改为目的地址；
3. 判断下一跳地址是否在ARP表中，如果在直接进行转发，*goto* 7；
4. 否则，判断是否存在下一跳不在任意子网的情况（防止转发表存在错误）；
5. 如果查询下一跳的ARP请求已经发过，则不再发送；
6. 将该包以及应发送ARP的信息，加入数据包缓存队列，等待ARP回复；
7. 将以太网头的源地址改为将转发数据包的端口，目的地址改为下一跳地址，同时IP头ttl减1。

b. 代码编写

```
def process_IP_Packet(self, packet):# 进行IP数据包的处理
     tar_route = self.match_subnet(dst_ip)
        if tar_route is None:
            pass
        else:
            ...#以太网包头src和下一跳nexthop的确定
             if nexthop in self.arp_table:#若已知下一跳mac地址
                self.IP_forward(packet,port,dst_mac)
            else:#加入缓冲队列
                if tar_route[2]!='#'
                and self.match_subnet(nexthop) is None:
                        pass#error! 没有匹配到下一跳的子网
                if nexthop in self.mycache.cache_packet:
                    has_same_arp=True
                request_pkt=
                AddPacket(src_mac,src_ip,nexthop,packet,port)
                #若未发过ARP请求
                send_packet(request_pkt[1], request_pkt[0])

def IP_forward(self, packet, port, dst_mac):#将数据包转发
        packet[Ethernet].dst, packet[IPv4].ttl =
        dst_mac, packet[IPv4].ttl - 1
        self.net.send_packet(port, packet)
```

模块2：数据包缓存队列

a. 实现原理

1. 数据结构：以字典的方式存储，关键字为应查询的ip地址，值为一个列表。列表第一项记录该待发送队列的停留时间，查询次数和发送ARP请求需要的信息，其余项为依次加入的数据包；
2. 每加入一项，则发回应发送的ARP请求包以及端口，由调用逻辑判断是否需要发送；

b. 代码编写

```
class PktCache:
    def __init__(self):
        self.cache_packet = dict()
    def AddPacket(self, src_mac, src_ip, dst_ip, packet, port):
        if dst_ip not in self.cache_packet:
            self.cache_packet[dst_ip] = list()
            self.cache_packet[dst_ip].append(
                [time.time(), 1, src_ip, src_mac, port])
        self.cache_packet[dst_ip].append(packet)
        return (create_ip_arp_request(src_mac, src_ip, dst_ip), port)
```

模块3：缓存队列ARP请求的处理

a. 实现原理

1. 将缓存队列按照第一次ARP发送请求的时间来排序，以保证发送顺序正确性；
2. 如果发送次数多于四次或者存活时间大于4秒，则将该队列中对应列表中所有包丢弃；
3. 否则，将查询队列每项ip地址的ARP请求包放入待查询队列，并返回；
4. 在路由接收包的循环中，若1s内没有收到包则调用缓存队列的ARP请求函数；

b. 代码编写

```
class PktCache:
    def GapArpQuery(self):
        now = time.time()
        cache_packet_list = list(self.cache_packet.items())
        cache_packet_list.sort(key=lambda x: x[1][0][0])
        arp_packets = list()
        for item in cache_packet_list:
            if item[1][0][1] <= 4
            and now - item[1][0][0] - item[1][0][1] >= 0.0:
                ...# port, src_mac, src_ip, nexthop相应的赋值
                arp_packets.append(
                    (create_ip_arp_request(src_mac, src_ip, nexthop)
                     , port))
                self.cache_packet[item[0]][0][1] += 1
            elif item[1][0][1] >= 5 or now - item[1][0][0] >= 5.0:
                self.cache_packet.pop(item[0])
        return arp_packets

class Router(object):
    while True:
            try:
                timestamp, dev, pkt = self.net.recv_packet(timeout=1.0)
            except NoPackets:
                self.arp_repeat()#调用处
```

模块4：ARP表更新的处理

a. 实现原理

1. 每当ARP请求或者ARP回复到达路由器时，都应更新ARP表；
2. 判断该ARP包的发送方是否在待查询队列中，若不在则返回；
3. 否则得到缓冲队列中对应的列表，并获取应转发数据包的端口，依次发送；

4. 最终将该ip地址对应的缓存列表清空。

b. 代码编写

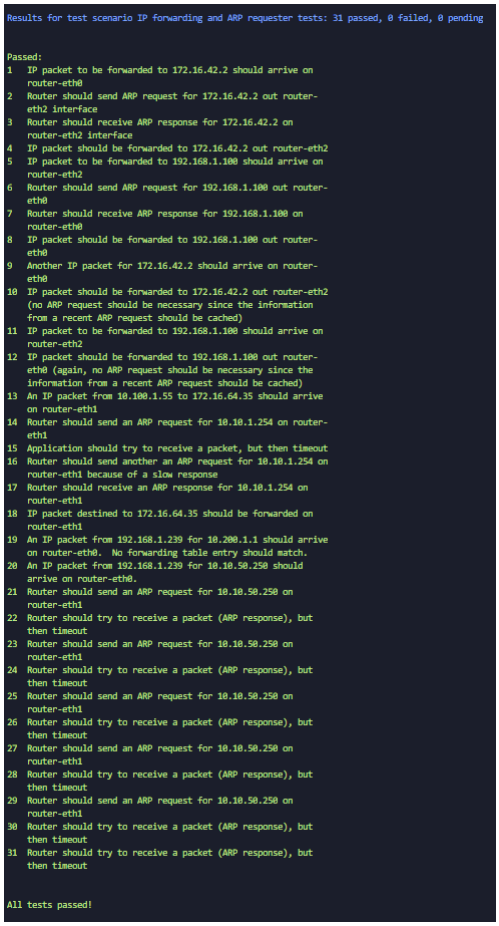
```
class PktCache:
    def GetArpReply(self, get_ip):
        if get_ip in self.cache_packet:
            return self.cache_packet[get_ip]
        return list()

class Router(object):
    def add_arp_table(self,src_ip,src_mac):
        self.arp_table[src_ip] = (src_mac, time.time())#更新ARP表
        if src_ip in self.mycache.cache_packet:
            cache_pkts = self.mycache.GetArpReply(src_ip)
            port = cache_pkts[0][4] #得到对应的端口，并发送
            for i in range(1, len(cache_pkts)):
                self.IP_forward(cache_pkts[i], port, src_mac)
            self.mycache.cache_packet.pop(src_ip)

    def process_arp_reply(self, port, packet):
        ...# 得到ARP回复包中的相应数据src_mac, src_ip, dst_mac, dst_ip
        # 进行缓冲队列是否命中的检查
        self.add_arp_table(src_ip,src_mac)
        return

    def process_arp_request(self, port, packet):
        ...# 得到ARP请求包中的相应数据src_mac, src_ip, dst_mac, dst_ip
        # 进行缓冲队列是否命中的检查
        self.add_arp_table(src_ip, src_mac)
        ...# 检测目的ip是否为路由器端口，并发送相应的ARP回复
```

c. 实现测试

I. test scenario测试

给定测试文件测试结果

运行$swyard\ -t\ routertests2.srpy\ myrouter.py$，结果如下：



测试文档说明

路由器端口沿用所给定的测试模板，同时测试代码行数较多且较为相似，故仅说明其主要功能。

测试代码检测以下方面的正确性：

1. 已知转发端口情况下，对包头的修改处理；
2. 其Arp请求和回复响应功能，以及重复请求的处理和丢弃；
3. 路由表匹配时直接命中某一路由子网，或者下一跳在某一子网这两种情况的处理；
4. 路由器中缓存队列各项Arp请求发送的处理。

- case 1：发送的目的 ip 为路由器端口，直接丢弃：

```
    req_ping = mk_ping('', '', '1.1.1.1','172.16.42.1')
    s.expect(PacketInputEvent('router-eth0', req_ping,
display=IPv4))
    s.expect(PacketInputTimeoutEvent(1.5))
```

- case 2：一个不在路由器子网中的 ip 向子网特定ip发送 ping 请求：

    1. 该ip 不在转发表文件中，而在路由器端口所在子网，故没有下一跳；
    2. 首先在对应子网端口发送Arp请求，询问该ip的mac地址；
    3. 设定该ip的mac地址，并发送Arp回复到该端口；
    4. 路由器修改数据包以太网头和 ttl ，并从该端口转发；
    5. 主机发送对应的ping回复，由于目的ip不在子网，故直接丢弃。

```
    req_ping = mk_ping('', '', '2.2.2.2','10.10.1.254')
    s.expect(PacketInputEvent('router-eth0', req_ping,
display=IPv4))
    otroarp = mk_arpreq("10:00:00:00:00:02", "10.10.0.1",
"10.10.1.254")
    s.expect(PacketOutputEvent('router-eth1', otroarp,
display=Arp))
    otroarpresponse = mk_arpresp(otroarp, "11:00:00:00:00:00")
    s.expect(PacketInputEvent("router-eth1", otroarpresponse,
display=Arp))
    req_ping := req_ping.ttl-1
    s.expect(PacketOutputEvent('router-eth1', req_ping,
display=IPv4))
    rep_ping = mk_ping("11:00:00:00:00:00",
"10:00:00:00:00:02", "10.10.1.254",'2.2.2.2', True)
    s.expect(PacketInputEvent('router-eth1', rep_ping,
display=IPv4))
    s.expect(PacketInputTimeoutEvent(1))
```

- case 3：在路由器子网中的源和目的进行ping通信，但是目的不进行Arp回复：
    1. 大体逻辑与case2 相同，但没有Arp回复，发送五次Arp请求后，将包丢弃。

```
    req_ping = mk_ping("22:00:00:00:00:00",
'ff:ff:ff:ff:ff:ff', '192.168.1.2','10.10.1.254')
    s.expect(PacketInputEvent('router-eth0', req_ping,
display=IPv4))
    req_ping = mk_ping('10:00:00:00:00:02','11:00:00:00:00:00',
                       '192.168.1.2','10.10.1.254',ttl=63)
    ...
    s.expect(PacketOutputEvent('router-eth0', req_arp,
display=Arp))
    repeat 4 times:
        s.expect(PacketInputTimeoutEvent(1.5))
        s.expect(PacketOutputEvent('router-eth0', req_arp,
display=Arp))
```

- case 4：目的与源的mac地址均已经获知，检测其通信功能是否正常，总共有两次接收和两次发送（请求和回复），：

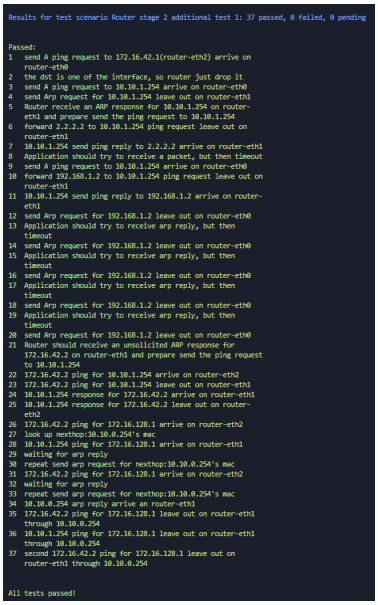
```
req_ping = mk_ping('33:00:00:00:00:00', '10:00:00:00:00:03',
'172.16.42.2','10.10.1.254')
...#对应的Ip发送和回复包的接收和转发
s.expect(PacketInputEvent("router-eth2", req_ping,
display=IPv4))
s.expect(PacketOutputEvent("router-eth1", req_ping,
display=IPv4))
s.expect(PacketInputEvent("router-eth1", rep_ping,
display=IPv4))
s.expect(PacketOutputEvent("router-eth2", rep_ping,
display=IPv4))
```

- case 5：目的地址相同的多个数据包到达路由器，，每等待1s都会有ARP请求包发送，在Arp回复到达后按到达顺序发送：

```
    req_ping_1 =
mk_ping('33:00:00:00:00:00','10:00:00:00:00:03',
                            '172.16.42.2','172.16.128.1',ttl=5)
    req_ping_2 =
mk_ping('11:00:00:00:00:00','10:00:00:00:00:02',
                            '10.10.1.254','172.16.128.1',ttl=4)
    req_ping_3 =
mk_ping('33:00:00:00:00:00','10:00:00:00:00:03',
                            '172.16.42.2','172.16.128.1',ttl=3)
    ...
    s.expect(PacketInputTimeoutEvent(1), 'waiting for arp
reply')
    s.expect(PacketOutputEvent("router-eth1", req_arp_1,
display=Arp))
```

测试结果

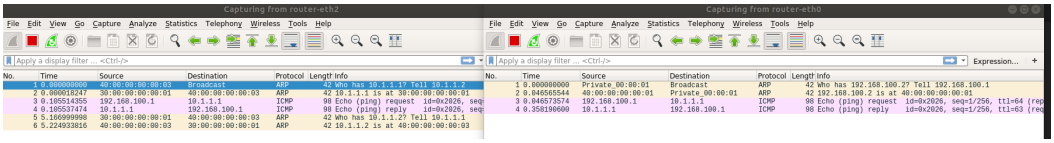
运行$swyard\ -t\ lab_4routertests.py\ myrouter_to$，结果如下：



H6 II. 抓包测试

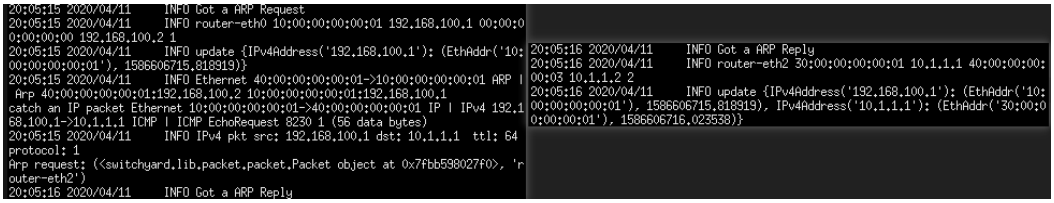
运行mininet，开启router运行$swyard\ myrouter.py$，并在wireshark中监视$eth0$和$eth2$的抓包情况；

运行$xterm\ server1$，在server1中$ping\ -c1$ 10.1.1.1，观察两个端口抓包结果：

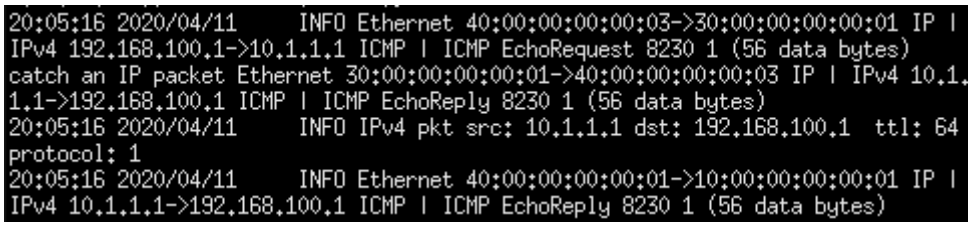


首先在端口 $eth2$（与10.1.1.1）相连的接口上收到了查询 10.1.1.1 的ARP 请求结果，并向 $40:00:00:00:00:03$ (ip： 192.168.100.1) 发送ARP回复；此后server1发送ping 请求与client进行通信，并成功收到回复。

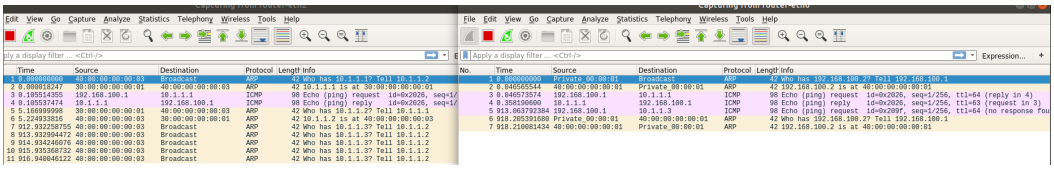
观察路由中的调试信息，也收到了相应的ARP请求和回复信息。



以及对应的ping ICMP包。



紧接着在server1中运行 $ping\ -c1\ 10.1.1.3$ ，可以观察到，在发送了五次查询 10.1.1.3 的ARP请求包后，便停止发包，ping失败，因为没有信息可以得知除了 $server1, server2, client$ 以为结点的mac地址。与文档描述相符，故测试正确。



H3 4. 总结与感想

这次实验的整体难度直线上升，从前期的理解，板块设计，功能划分，以及运行调试都充满着艰难险阻。

本着先实现再优化的想法，将所有结构揉在一个类中，很多功能相似代码重复的地方，代码逻辑异常混乱。在调试时，最开始使用输出调试法，由于输出内容过多过乱，测试一度陷入僵局，并且遇到很多不明所以的错误信息，反复查阅switchyard文档并改用ide debug测试才变得顺利。在理论知识的学习过程中，很多事情逻辑都是理解个大概，细节很少去考虑，在实现匹配路由表和转发时才发现事情没有那么简单。将代码模块分成两个类，让结构变得更清晰。

实验的完成让人很有成就感，希望可以保持对这种知识探究的渴望和乐趣。

H3 5. 文档结构