实验七：路由器转发实验

范子墨
2021K8009929006

一、 实验内容

1、 实验内容一

a) 在主机上安装 arptables, iptables，用于禁止每个节点的相应功能
   i. sudo apt install arptables iptables
b) 运行给定网络拓扑(router_topo.py)
   i. 路由器节点 r1 上执行脚本 (disable_arp.sh, disable_icmp.sh, disable_ip_forward.sh)，禁止协议栈的相应功能
   ii. 终端节点 h1-h3 上执行脚本 disable_offloading.sh
c) 在 r1 上执行路由器程序
   i. 在 r1 中运行./router，进行数据包的处理
d) 在 h1 上进行 ping 实验
   i. Ping 10.0.1.1 (r1)，能够 ping 通
   ii. Ping 10.0.2.22 (h2)，能够 ping 通
   iii. Ping 10.0.3.33 (h3)，能够 ping 通
   iv. Ping 10.0.3.11，返回 ICMP Destination Host Unreachable
   v. Ping 10.0.4.1，返回 ICMP Destination Net Unreachable

2、 实验内容二

a) 构造一个包含多个路由器节点组成的网络
   i. 手动配置每个路由器节点的路由表
   ii. 有两个终端节点，通过路由器节点相连，两节点之间的跳数不少于 3 跳，手动配置其默认路由表
b) 连通性测试
   i. 终端节点 ping 每个路由器节点的入端口 IP 地址，能够 ping 通
c) 路径测试
   i. 在一个终端节点上 traceroute 另一节点，能够正确输出路径上每个节点的 IP 信息

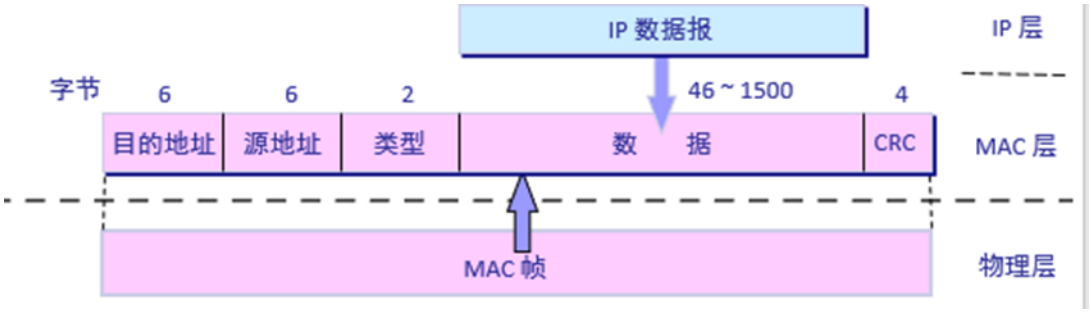
二、路由器程序

1、 原有代码思路

在 main 文件用 while(1)循环进行持续监听，在收到数据包之后使用 handle_packet 函数进行处理，并根据以太网帧中上层协议的类型分别跳转至 handle_ip_packet 和 handle_arp_packet，两个函数均需要自己进行填充。

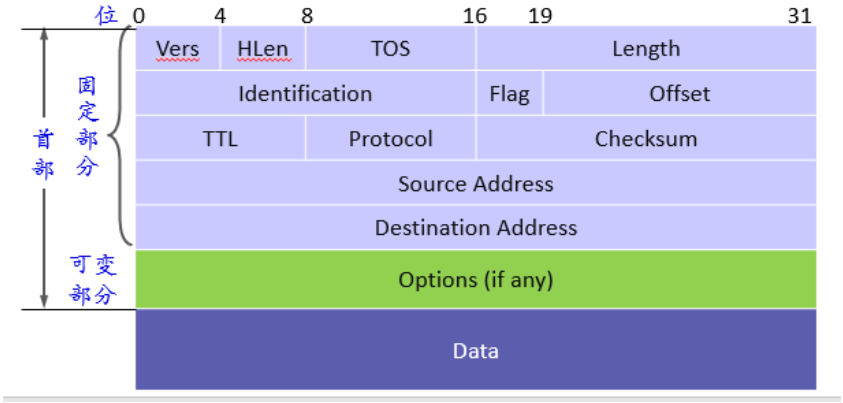
其中，handle_ip_packet 在对数据包进行处理，找出下一跳的目的 ip 地址，此时完成了网络层的工作，想要真正转发，还需要知道 ip 地址对应的 mac 地址，所以每个具有三层（指网络层、数据链路层、物理层）结点中都有一个 ARP 高速缓存，用于存储结点所在局域网内各结点的 ip 地址到其硬件地址的映射表。因此在 iface_send_packet_by_arp 函数中实现的是数据链路层的工作，查找目的 ip 地址是否在 arpcache 中，若有，则将 ip 数据包加上头部和尾部发送，否则跳转至 arpcache_append_packet。

handle_arp_packet 主要是对于 arp 请求和 arp 回复的处理，维护 arpcache 中的 IP、MAC 映射条目和查找不到相应条目而等待 ARP 应答的数据包。



图表 1：mac 帧首部和尾部

2、 ip 数据包处理



图表 2：IP 数据报首部

首先获取数据包的首部，获取目的地址。

```
struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
```

首先判断是否与接收接口一致，若一致，则说明该数据包已经到达目的地址，进一步判断是否是 ICMP 数据包，即首部中的 protocol 字段是否是 IPPROTO_ICMP，若一致则说明该包为测试是否 ping 通的数据包，返回 reply 即可，否则直接丢弃该包。

```
if (destip == iface_ip) {
        if (ip_hdr->protocol == IPPROTO_ICMP) {
            // log(DEBUG, "Receive ICMP echo request!");
            struct icmphdr *icmp_hdr = (struct icmphdr *)(IP_DATA(ip_hdr));
            if (icmp_hdr->type == ICMP_ECHOREQUEST) {
                icmp_send_packet(packet, len, ICMP_ECHOREPLY, 0);
            }
        } else {
            free(packet);
        }
    }
```

若目的地址与接收接口不同，则该包需要继续转发，按照最长前缀匹配查找路由表。最长前缀匹配遍历路由表，从匹配结果中选出具有最长前缀的路由。具体来说，是路由表中的每一条均包含目的 ip 地址、掩码和下一跳地址。由于最长匹配虽然应该比较掩码位数，但掩码是从高到低连续掩码位数个 1，后续补零，因此比较掩码即比较掩码位数。

```
list_for_each_entry(p, &rtable, list) {
    if((p->mask > longest_prefix) && ((p->dest & p->mask) == (dst & p->mask))){
    longest_prefix = p->mask;
```

```
        longest_match = p;
        }
}
```

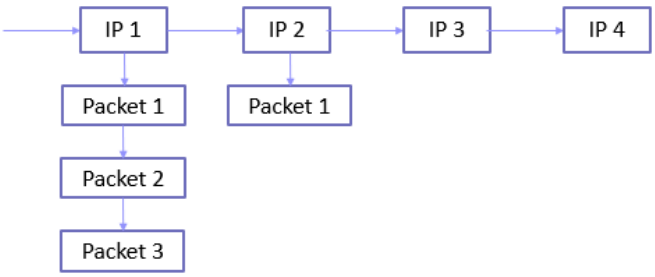
若无匹配，则发回 icmp 路由表查找失败。否则判断该包跳数是否为已经耗尽，若耗尽，则发回 icmp 超时。其余情况判断该数据包是否已经到达目的地址所在网络，若到达，则下一跳地址定为目的地址直接交付，否则为网关地址，而后进入数据链路层的处理部分。

```
u32 nxt_hop = entry->gw == 0 ? destip : entry->gw;
ip_hdr->checksum = ip_checksum(ip_hdr);
iface_send_packet_by_arp(entry->iface, nxt_hop, packet, len);
```

使用 arpcache_lookup 查找目的地址是否存在，由于 arpcache 是一个共享变量，因此要设置临界区，防止访问出现问题。

```
pthread_mutex_lock(&arpcache.lock);
    for (int i = 0; i < MAX_ARP_SIZE; i++) {
        if (ip4 == arpcache.entries[i].ip4 && arpcache.entries[i].valid != 0) {
            memcpy(mac,arpcache.entries[i].mac,ETH_ALEN);
            pthread_mutex_unlock(&arpcache.lock);
            return 1;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
    return 0;
```

若目的地址在 arpcache 中存在，则直接发送数据包，否则使用 arpcache_append_packet 处理。arpcache 中缓存查找不到相应条目而等待 arp 应答的数据包。



图表 3：arpcache 请求队列

由于访问共享变量，因此首先设置临界区。而后查找现有请求队列中是否有同一目的 ip 的队列，若有，只需添加到该队列中。添加后不需要再次发送 arp 请求，因为该队列在创建时已经发送率 arp 请求，但是还未收到 reply。

```
list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
        if(req_entry->iface == iface && req_entry->ip4 == ip4){
            list_add_tail(&(recv_pkt->list),&(req_entry->cached_packets));
            pthread_mutex_unlock(&arpcache.lock);
            return ;
        }
    }
```

否则添加一个新的队列，将该请求加入队列中。而后发送该队列中的请求。

arp_send_request 与后文 arp_send_reply 处理类似，在此不做赘述。

3、 arp 数据包处理

在 handle_arp_packet 首先检查接收接口 ip 与该包目的地址 ip 是否相同，由于 arp 包只在同一局域网中，因此应当只有一跳，所以收到的应该是已经到达的数据包。而后对 arp 数据包进行分类，若为回复 mac 地址的数据包，则跳转至 arpcache_insert 函数，添加 arpcache 中 ip 地址与 mac 地址的对应。若为请求数据包，则跳转至 arp_send_reply 函数，将自己的 mac 地址发出去。其余情况为不符合要求的数据包，需丢弃。

对于 arpcache_insert 函数，由于访问共享变量，需要上锁。而后遍历 arpcache 查找是否有空闲条目，即 valid=0。若有，则直接将 ip 地址与 mac 地址的对应插入该条目。

```
for (int i = 0; i < MAX_ARP_SIZE; i++) {
        if (arpcache.entries[i].valid == 0) {
            arpcache.entries[i].ip4 = ip4;
            memcpy(&arpcache.entries[i].mac, mac, ETH_ALEN);
            arpcache.entries[i].valid = 1;
            arpcache.entries[i].added = time(NULL);
            found_valid_entry = 1;
            break;
        }
    }
```

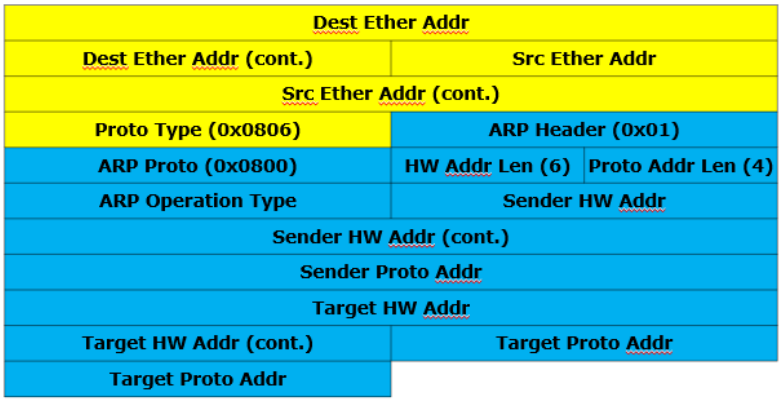
若无空闲条目，则在当前所有 arpcache 条目中随机选择一个进行替换，并且更新时间戳。

由于 arpcache 中新增了条目，因此要将目的地址为该 ip 地址的数据包全部发送出去，即遍历请求队列，将 mac 地址赋给 IP 地址对应的包的 mac 帧首部，而后发送。

```
list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
        if (req_entry->ip4 == ip4) {
            struct cached_pkt *pkt_entry = NULL, *pkt_q;
            list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list) {
                struct ether_header *eh = (struct ether_header *)(pkt_entry->packet);
                memcpy(eh->ether_dhost, mac, ETH_ALEN);
                iface_send_packet(req_entry->iface, pkt_entry->packet, pkt_entry->len);
                list_delete_entry(&(pkt_entry->list));
                free(pkt_entry);
            }

            list_delete_entry(&(req_entry->list));
            free(req_entry);
        }
    }
```

若要回复 arp 请求，则在 arp_send_reply 中，按照 arp 的格式对每一部分进行赋值。



图表 4：arp 首部

首先对 MAC 帧头部进行赋值，没有偏移量。

```
struct ether_header *eth_header = (struct ether_header *)reply_packet;
```

而后对 arp 数据包内容进行赋值，偏移量为 mac 帧头部长度。

```
struct ether_arp *eth_arp = (struct ether_arp *)(reply_packet + ETHER_HDR_SIZE);
```

填充完后使用 iface_send_packet 发送即可。

4、 arpcache_sweep

在 main 函数中还有一个线程运行 arpcache_sweep 函数，该函数每一秒运行一次，对 arpcache 中的条目和 arp 请求进行定期处理。

对于 arpcache 中的条目，如果超过十五秒，则将 valid 置 0，表示该条目已经无效。

```
for (int i = 0; i < MAX_ARP_SIZE; i++) {
            time_t now = time(NULL);
            if ((now - arpcache.entries[i].added) > ARP_ENTRY_TIMEOUT &&
arpcache.entries[i].valid) {
                arpcache.entries[i].valid = 0;
            }
        }
```

对于请求队列，如果已经发送了五次请求，则认为目的地不可达，对每个包发回 icmp 请求失败，同时删除该队列和队列内所有包。

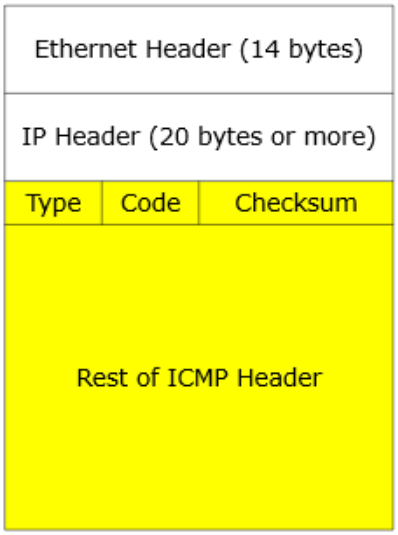
```
if(req_entry->retries > ARP_REQUEST_MAX_RETRIES){
                struct cached_pkt *pkt_entry = NULL, *pkt_q;
                list_for_each_entry_safe(pkt_entry, pkt_q, &(req_entry->cached_packets), list) {
                    pthread_mutex_unlock(&(arpcache.lock));
                    icmp_send_packet(pkt_entry->packet, pkt_entry->len,
ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
                    pthread_mutex_lock(&(arpcache.lock));
                    free(pkt_entry);
                }
                list_delete_entry(&(req_entry->list));
                free(req_entry);
                continue;
            }
```

而如果还未发出五次请求且已经过了一秒还没收到回复，重新发送请求。

```
if(now - req_entry->sent >= 1){、
      arp_send_request(req_entry->iface, req_entry->ip4);
    req_entry->sent = now;
     req_entry->retries ++;
}
```

5、 ICMP

用于网络诊断和控制：主机或路由器用于报告差错、异常情况。



路由表查找失败

Type: 3, Code: 0, Rest of ICMP Header: 前4字节设置为0，接着拷贝收到数据包的IP头部（>= 20字节）和随后的8字节

ARP查询失败

Type: 3, Code: 1, Rest of ICMP Header: 同上

TTL值减为0

Type: 11, Code: 0, Rest of ICMP Header:同上

收到Ping本端口的数据包（Type为8）

Type: 0, Code: 0, Rest of ICMP Header: 拷贝Ping包中的相应字段

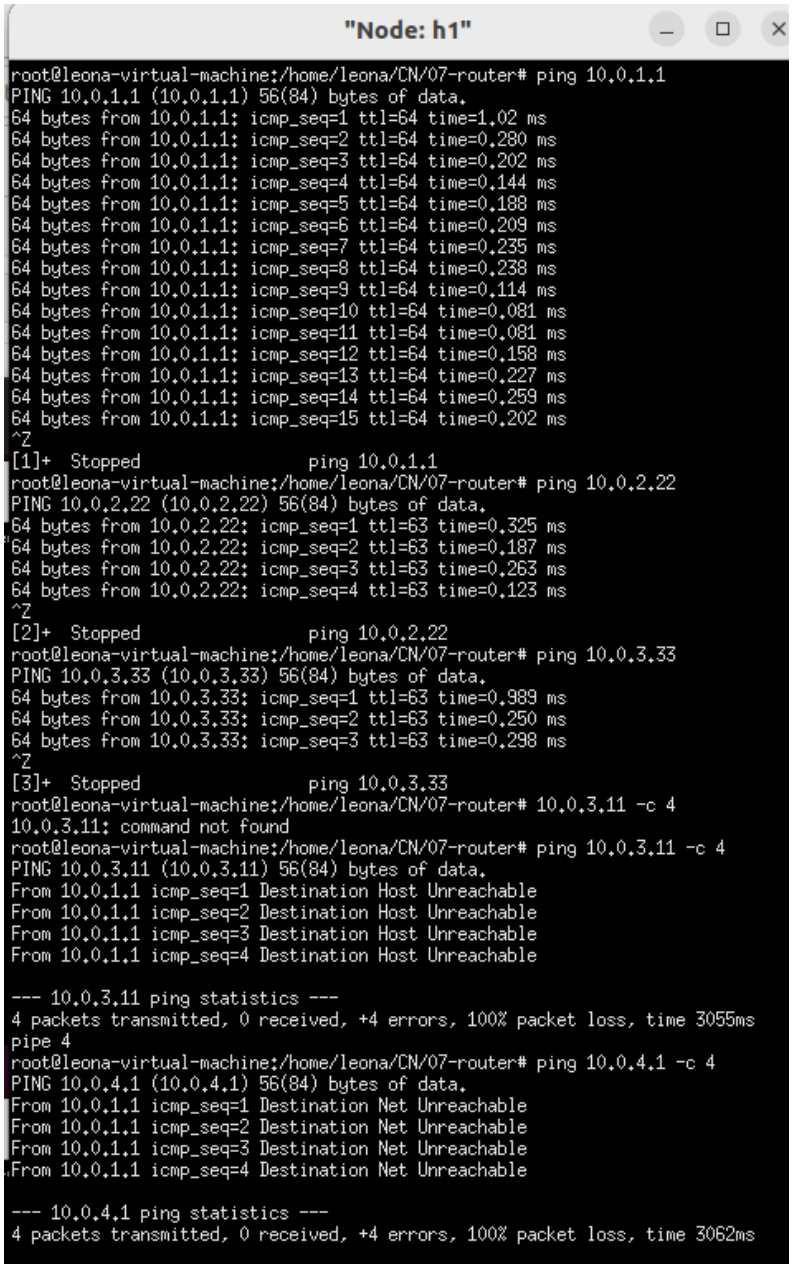
图表 5：ICMP 结构及类型

```
char *send_pkt = malloc(packet_len * sizeof(char));
struct ether_header *eh = (struct ether_header *) send_pkt;
struct iphdr *iph = packet_to_ip_hdr(send_pkt);
struct icmphdr *icmph = (struct icmphdr *)(send_pkt + ETHER_HDR_SIZE +
IP_BASE_HDR_SIZE);
```

如果是 ICMP 回复（ICMP_ECHOREPLY），则长度为原始数据包长度；否则，长度将包括以太网帧头、IP 帧头、ICMP 帧头、原始 IP 数据包内容。而后对 mac 帧首部，IP 帧首部，icmph 首部分别赋值后发出数据包。

这里大部分使用指针进行操作，这里只解释其中一个。icmph 指针提供了一个方便的方式来访问 send_pkt 内存中的 ICMP 头部，而不需要复制或更改 send_pkt 的内容。通过 icmph 修改 ICMP 头部字段时，这些更改将反映在 send_pkt 指向的内存区域中。

6、 结果

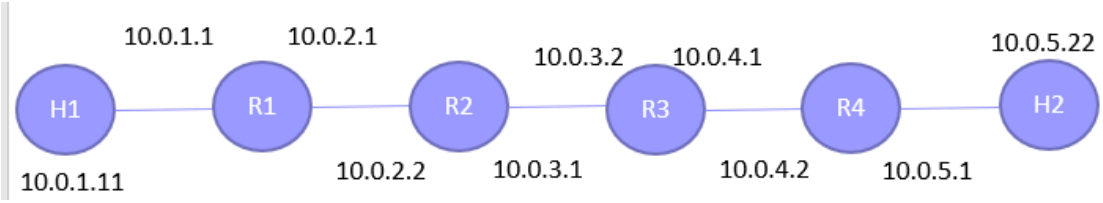


图表 6：连通性测试结果

结果符合预期。

三、 构造多个路由器节点

1、 设计思路



图表 7：预期网络

只需给每个路由器给出初始路由表即可。

```
r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
r1.cmd('ifconfig r1-eth1 10.0.2.1/24')
r1.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
r1.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
r1.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
```

2、 结果

```
--- 10.0.5.22 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 1.268/2.316/3.854/1.111 ms
root@leona-virtual-machine:/home/leona/CN/07-router# ping 10.0.1.1 -c 3
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=0.519 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.326 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.293 ms

--- 10.0.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2054ms
rtt min/avg/max/mdev = 0.293/0.379/0.519/0.099 ms
root@leona-virtual-machine:/home/leona/CN/07-router# ping 10.0.2.2 -c 3
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=63 time=1.59 ms
64 bytes from 10.0.2.2: icmp_seq=2 ttl=63 time=0.554 ms
64 bytes from 10.0.2.2: icmp_seq=3 ttl=63 time=0.855 ms

--- 10.0.2.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2031ms
rtt min/avg/max/mdev = 0.554/1.000/1.592/0.436 ms
root@leona-virtual-machine:/home/leona/CN/07-router# ping 10.0.3.2 -c 3
PING 10.0.3.2 (10.0.3.2) 56(84) bytes of data.
64 bytes from 10.0.3.2: icmp_seq=1 ttl=62 time=1.68 ms
64 bytes from 10.0.3.2: icmp_seq=2 ttl=62 time=1.11 ms
64 bytes from 10.0.3.2: icmp_seq=3 ttl=62 time=0.763 ms

--- 10.0.3.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.763/1.185/1.683/0.379 ms
root@leona-virtual-machine:/home/leona/CN/07-router# ping 10.0.4.2 -c 3
PING 10.0.4.2 (10.0.4.2) 56(84) bytes of data.
64 bytes from 10.0.4.2: icmp_seq=1 ttl=61 time=2.27 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=61 time=1.70 ms
64 bytes from 10.0.4.2: icmp_seq=3 ttl=61 time=1.82 ms

--- 10.0.4.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 1.697/1.930/2.274/0.248 ms
root@leona-virtual-machine:/home/leona/CN/07-router# ping 10.0.5.22 -c 3
PING 10.0.5.22 (10.0.5.22) 56(84) bytes of data.
64 bytes from 10.0.5.22: icmp_seq=1 ttl=60 time=2.30 ms
64 bytes from 10.0.5.22: icmp_seq=2 ttl=60 time=1.57 ms
64 bytes from 10.0.5.22: icmp_seq=3 ttl=60 time=1.19 ms

--- 10.0.5.22 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 1.189/1.683/2.296/0.459 ms
```

图表 8：连通性测试

```
root@leona-virtual-machine:/home/leona/CN/07-router# traceroute 10.0.5.22
traceroute to 10.0.5.22 (10.0.5.22), 30 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.364 ms  0.219 ms  0.154 ms
 2  10.0.2.2 (10.0.2.2)  0.789 ms  0.773 ms  0.760 ms
 3  10.0.3.2 (10.0.3.2)  0.799 ms  0.781 ms  0.769 ms
 4  10.0.4.2 (10.0.4.2)  1.988 ms  2.144 ms  2.134 ms
 5  10.0.5.22 (10.0.5.22)  2.121 ms  2.099 ms  2.496 ms
```

图表 9：traceroute 测试

结果符合预期。

四、 实验中遇到的一些问题

1、 指针的使用

刚开始阅读原有代码梳理思路时就发现使用了一些我没有见过的指针用法。

```
static inline struct iphdr *packet_to_ip_hdr(const char *packet)
{
    return (struct iphdr *)(packet + ETHER_HDR_SIZE);
}
```

packet 是一个指向字符的指针，通常用于存储一个以太网数据帧，包括以太网头部和 IP 报头。ETHER_HDR_SIZE 是一个表示以太网头部的大小的常数或宏定义。通过将这个偏移量添加到 packet 指针上，将 packet 指针从以太网头部的起始位置移动到 IP 报头的起始位置。struct iphdr * 表示一个指向 IP 报头的指针。通过将移动后的 packet 指针强制类型转换为 struct iphdr * 类型，创建了一个指向 IP 报头的指针。最后，函数返回这个指向 IP 报头的指针，这就可以在程序中使用这个指针来访问和操作 IP 报头中的字段，比如源 IP 地址、目标 IP 地址等，同时也可以修改指针所指内容。

刚开始很难理解这种用法，而且把不准偏移量。逐渐使用之后好了一点点，可以更好的对一份数据报中的内容进行分类后再赋值。

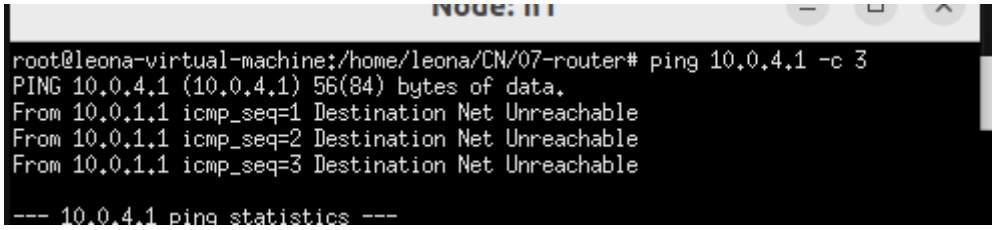
```
struct icmphdr *icmph = (struct icmphdr *)(send_pkt + ETHER_HDR_SIZE +
IP_BASE_HDR_SIZE);
```

2、字节序转换函数错误

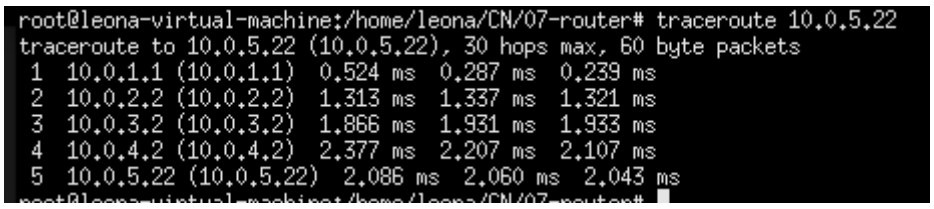
日期	用户名	实验名称	作业状态	描述
2023-11-01 17:13	2021K8009929006	路由器转发实验	解答错误	h1_ping_r1_test:Pass; h1_ping_h2_test:Pass; h1_ping_h3_test:Pass; h1_ping_uh_test:Pass; h1_ping_un_test:Fail; hop3_ping_test:Pass; hop3_traceroute_test:Fail;

图表 10：测试点错误

将文件上传后发现有两个测试点没有通过，即 h1 在 ping 未知 net 时得到正确的反馈和节点之间的路由选择正确。经过在我本地上的测试，发现这两个可以正常输出。



图表 11：ping 未知 net 输出正常



图表 12：traceroute 输出正常

因此梳理这两个测试点需要经历的过程，检查经过的每个函数。对于 h1 在 ping 未知 net 时，计算机会向目标主机发送一个小数据包，然后等待目标主机的响应。

根据当前实验的拓扑文件设置，该文件会发送给路由器(10.0.1.1 为路由器端口 r1-eth0，之后由路由器处理该数据包。

```
h1.cmd('route add default gw 10.0.1.1 dev h1-eth0')
```

在 handle_ip_packet 中由于没有匹配的地址，路由器也没有设置默认网关，因此跳转到 icmp_send_packet 函数，而后跳转到 ip_send_packet 函数，再之后跳转至 iface_send_packet_by_arp,由于在路由表中没有找到相对应的条目，因此会发送 Destination Net Unreachable。在整个跳转过程中没有发现什么问题。

```
icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
```

而对于另一个测试点，由于设置了默认路由，因此可以正常发送。转而查询剩余所有自己写的函数，再次检查所有 arp 请求和接收函数，发现有一个地方位宽出现问题。

```
else if(ntohl(eth_arp->arp_op) == ARPOP_REQUEST){
```

更改后如下

```
else if (ntohs(arp->arp_op) == ARPOP_REQUEST){
```

变量位数错误确实会导致问题，因此在改正后发现 traceroute 时间较更改后时间更长，可能由于在广播后才采用了默认路径，耗费了一些时间。