**实验七：路由器转发实验**

范子墨

2021K8009929006

1. **实验内容**
2. 实验内容一
   1. 在主机上安装arptables, iptables，用于禁止每个节点的相应功能
      1. sudo apt install arptables iptables
   2. 运行给定网络拓扑(router\_topo.py)
      1. 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
      2. 终端节点h1-h3上执行脚本disable\_offloading.sh
   3. 在r1上执行路由器程序
      1. 在r1中运行./router，进行数据包的处理
   4. 在h1上进行ping实验
      1. Ping 10.0.1.1 (r1)，能够ping通
      2. Ping 10.0.2.22 (h2)，能够ping通
      3. Ping 10.0.3.33 (h3)，能够ping通
      4. Ping 10.0.3.11，返回ICMP Destination Host Unreachable
      5. Ping 10.0.4.1，返回ICMP Destination Net Unreachable
3. 实验内容二
   1. 构造一个包含多个路由器节点组成的网络
      1. 手动配置每个路由器节点的路由表
      2. 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
   2. 连通性测试
      1. 终端节点ping每个路由器节点的入端口IP地址，能够ping通
   3. 路径测试
      1. 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

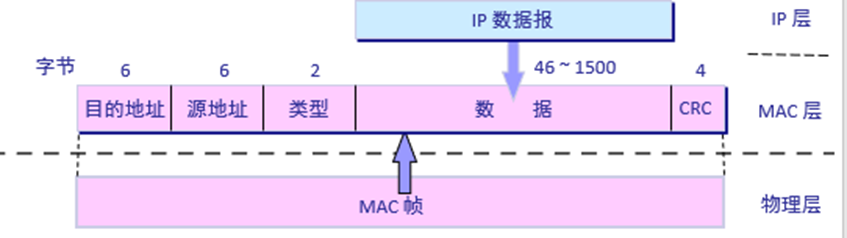
**二、路由器程序**

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。

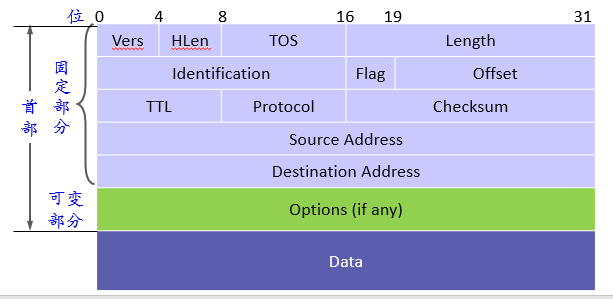
其中，handle\_ip\_packet在对数据包进行处理，找出下一跳的目的ip地址，此时完成了网络层的工作，想要真正转发，还需要知道ip地址对应的mac地址，所以每个具有三层（指网络层、数据链路层、物理层）结点中都有一个ARP高速缓存，用于存储结点所在局域网内各结点的ip地址到其硬件地址的映射表。因此在iface\_send\_packet\_by\_arp函数中实现的是数据链路层的工作，查找目的ip地址是否在arpcache中，若有，则将ip数据包加上头部和尾部发送，否则跳转至arpcache\_append\_packet。

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

1. 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;

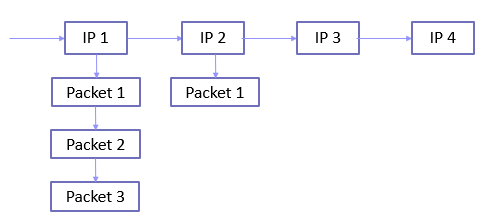
        }

    }

    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。

1. 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);

            }

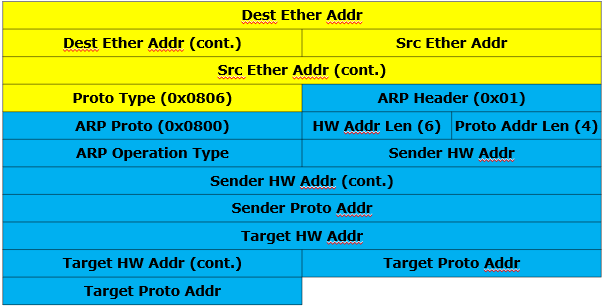
            list\_delete\_entry(&(req\_entry->list));

            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。

1. 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);

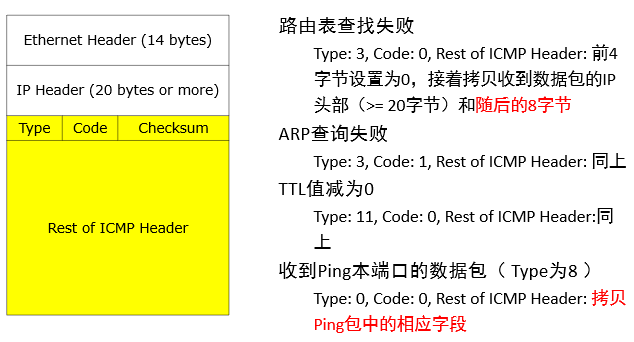
    req\_entry->sent = now;

req\_entry->retries ++;

}

1. 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。



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;

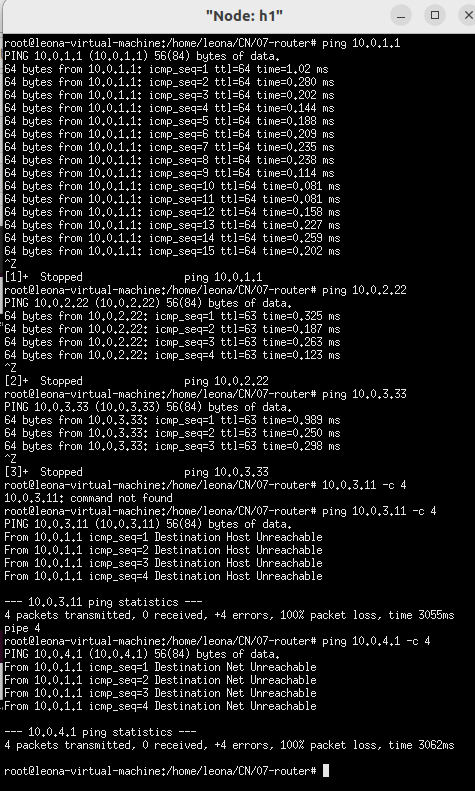
struct iphdr \*iph = packet\_to\_ip\_hdr(send\_pkt);

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。

1. 结果

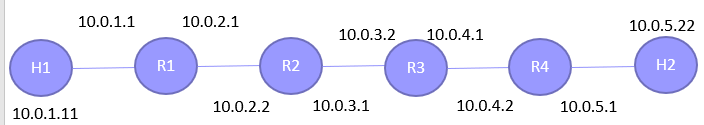


图表 6：连通性测试结果

结果符合预期。

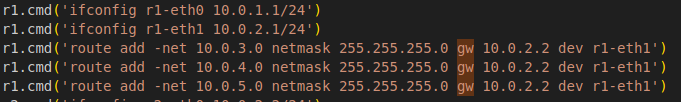
* 1. **构造多个路由器节点**

1. 设计思路

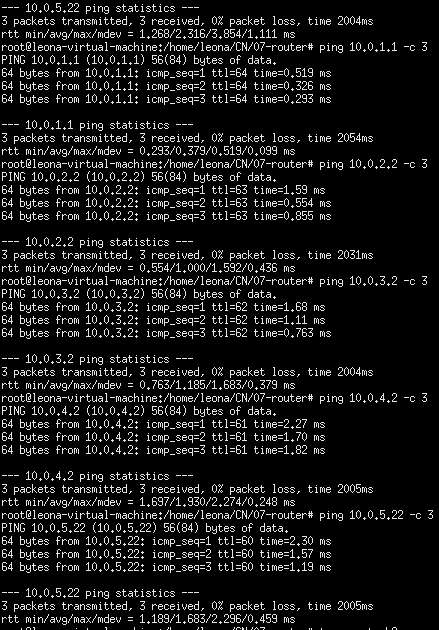


图表 7：预期网络

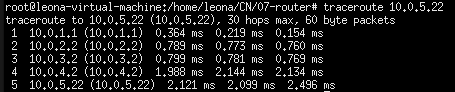
只需给每个路由器给出初始路由表即可。



1. 结果



图表 8：连通性测试



图表 9：traceroute测试

结果符合预期。

* 1. **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。

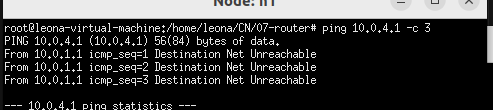
struct icmphdr \*icmph = (struct icmphdr \*)(send\_pkt + ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE);

2、字节序转换函数错误

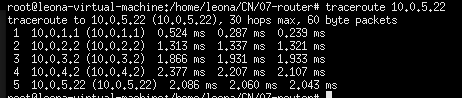


图表 10：测试点错误

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。