02-router

My name: 蒋鹏宇

My Student ID: 211502021

This lab took me about 10 hours to do.

Implementation Explanation:

ARP报文处理

接收到的ARP报文分为请求报文和响应。对于请求报文，当其请求的目的IP为本端口地址时，回复一个ARP响应报文，并且把源的IP和MAC映射关系加入ARP缓存。对于响应报文，当其回复的目的地址为本端口地址时，说明发出的ARP请求得到了回复，此时把源端口的IP和MAC映射关系加入ARP缓存。

实现代码较长，在此不表

ARP表管理

首先是在ARP表中查找IP地址对应的MAC地址，我们遍历所有表项，如果有表项的IP和我们要查询的IP相等并且表项有效，则选取该表项

```
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
        pthread_mutex_lock(&arpcache.lock);

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

        pthread_mutex_unlock(&arpcache.lock);
        return 0;
}
```

如果在ARP表中查找不到相应条目，我们将该分组挂到ARPCache中的一个等待队列里，然后发出相应的ARP请求报文，直到我们收到这个IP地址对应的MAC地址，再释放这个分组

```
void arpcache_append_packet(iface_info_t *iface,u32 ip4,char *packet,int len)
{
        struct arp_req *req_entry=NULL,*req_q;
        struct cached_pkt *recv_pkt;
        recv_pkt=malloc(sizeof(struct cached_pkt));
        recv_pkt->len=len;
        recv_pkt->packet=packet;

        pthread_mutex_lock(&arpcache.lock);
        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;
                }
        }

        struct arp_req *added_req_list=(struct arp_req*)malloc(sizeof(struct arp_req));
        added_req_list->iface=iface;
        added_req_list->ip4=ip4;
        added_req_list->sent=time(NULL);
        added_req_list->retries=1;

        init_list_head(&(added_req_list->cached_packets));
        list_add_tail(&(recv_pkt -> list),&(added_req_list->cached_packets));
        list_add_tail(&(added_req_list->list),&(arpcache.req_list));

        arp_send_request(iface,ip4);

        pthread_mutex_unlock(&arpcache.lock);
}
```

第三是插入IP->MAC地址映射。我们插入映射时，需要找一个地方插入，如果有空闲表项（valid=0），我们插入这个空闲表项，如果没有我们随机替换一个。所以我们首先需要遍历整个表寻找空闲表项，找不到再随机替换。插入后需要释放正在等待该地址的分组，因为我们已经查到了他们所需要的MAC地址

```
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
{
        pthread_mutex_lock(&arpcache.lock);

        // Find the entry.
        int pos=-1;

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

        if(pos==-1)
        {
                pos=time(NULL)%32;
                arpcache.entries[pos].added=time(NULL);
                arpcache.entries[pos].ip4=ip4;
                memcpy(arpcache.entries[pos].mac,mac,ETH_ALEN);
                arpcache.entries[pos].valid=1;
        }

        // Delete all the pending packets.
        struct arp_req *req_entry=NULL,*req_q;
        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_pac
                        {
                                struct ether_header *eth_hdr=(struct ether_header*)(pkt_
                                memcpy(eth_hdr->ether_dhost,mac,ETH_ALEN);
                                iface_send_packet(req_entry->iface,pkt_entry->packet,pkt

                                list_delete_entry(&(pkt_entry->list));
```

```
                                free(pkt_entry);
                        }

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

        pthread_mutex_unlock(&arpcache.lock);
}
```

最后是清理表项和分组缓存。这个主要有两部分，第一部分是清除存在时间过长的表项，第二部分是清除请求次数过多的分组

```
void *arpcache_sweep(void *arg)
{
	while(1)
	{
		sleep(1);
		pthread_mutex_lock(&(arpcache.lock));
		struct arp_req *req_entry=NULL,*req_q;
		time_t now=time(NULL);
		for (int i=0;i<MAX_ARP_SIZE;i++)
		{
			if(arpcache.entries[i].valid&&now-arpcache.entries[i].added>ARP_
				arpcache.entries[i].valid=0;
		}

		list_for_each_entry_safe(req_entry,req_q,&(arpcache.req_list),list)
		{
			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->ca
				{
					pthread_mutex_unlock(&(arpcache.lock));
					icmp_send_packet(pkt_entry->packet,pkt_entry->le
					pthread_mutex_lock(&(arpcache.lock));
					free(pkt_entry);
				}
				list_delete_entry(&(req_entry->list));
				free(req_entry);
				continue;
			}
			if(now-req_entry->sent>=1)
			{
				arp_send_request(req_entry->iface,req_entry->ip4);
				req_entry->sent=now;
				req_entry->retries++;
			}
		}
		pthread_mutex_unlock(&(arpcache.lock));
	}

	return NULL;
```

```
}
```

路由表查询与数据报转发

遍历整个路由表，当地址匹配且mask大于当前最大mask时，更新目标表项和最大mask,另外还要考虑默认路由的掩码为全0的情况

实现:

```
rt_entry_t *longest_prefix_match(u32 dst)
{
        rt_entry_t *pos,*res=NULL;
        u32 max_mask=0;
        list_for_each_entry(pos,&rtable,list)
        {
                if((pos->mask&pos->dest)==(pos->mask&dst))
                {
                        if(pos->mask>max_mask)
                        {
                                res=pos;
                                max_mask=pos->mask;
                        }
                }
        }
        return res;
}
```

在得到对应的路由器表项后，我们查看其网关地址，如果为0，意味着目的主机在同一子网内，此时直接向目的IP转发数据报；否则向下一跳网关转发数据报

```
void ip_send_packet(char *packet, int len)
{
        struct ether_header *eh=(struct ether_header*)packet;
        struct iphdr *ih=packet_to_ip_hdr(packet);

        rt_entry_t *find_rt=longest_prefix_match(ntohl(ih->daddr));
        if(find_rt==NULL)
        {
                free(packet);
                return;
        }

        u32 next_ip;
        if (find_rt->gw)
                next_ip=find_rt->gw;
        else
                next_ip=ntohl(ih->daddr);

        iface_send_packet_by_arp(find_rt->iface,next_ip,packet,len);
}
```

我们收到ip数据报后，首先查看目的地址和目前的iface的addr是不是一样，如果一样说明这是ICMP报文的请求回复报文，我们返回相应的reply;如果不一样则在路由表查找下一跳ip地址，如果查不到，说明我们无法到达，返回icmp unreachable异常，查的到的话再处理ttl，如果还能转发的话就将ip数据报转发出去

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
	struct iphdr *ip_hdr=packet_to_ip_hdr(packet);
	u32 daddr=ntohl(ip_hdr -> daddr);

	// ICMP packet
	if(daddr==iface->ip)
	{
		struct iphdr *ip_hdr=packet_to_ip_hdr(packet);
		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);
		return;
	}

	// Search daddr in router table.
	rt_entry_t *p_rt=longest_prefix_match(daddr);
	if(p_rt==NULL)
	{
		icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
		return;
	}

	// ttl
	ip_hdr->ttl--;
	if(ip_hdr->ttl<=0)
	{
		icmp_send_packet(packet,len,ICMP_TIME_EXCEEDED,ICMP_EXC_TTL);
		return;
	}
	ip_hdr->checksum=ip_checksum(ip_hdr);

	// Get the next jump.
	u32 next_jump=p_rt->gw?p_rt->gw:daddr;

	// forward packet by arp protocol.
	iface_send_packet_by_arp(p_rt->iface,next_jump,packet,len);
}
```

ICMP数据报发送

考虑到ICMP数据报的格式，我们首先要确定其长度，主要是区分ping数据报和其他类型数据报；确定长度后就开始填充ICMP报文

```
void icmp_send_packet(const char *in_pkt,int len,u8 type,u8 code)
{
        struct iphdr *in_ip_hdr=packet_to_ip_hdr(in_pkt);

    int pkt_len=0;
    if (type==ICMP_ECHOREPLY)
        pkt_len=len;
        else
        pkt_len=ETHER_HDR_SIZE+IP_BASE_HDR_SIZE+ICMP_HDR_SIZE+IP_HDR_SIZE(in_ip_hdr)+8;

    char *sent_pkt=(char*)malloc(pkt_len);
    struct ether_header *eh=(struct ether_header*)sent_pkt;
    struct iphdr *ip_hdr=packet_to_ip_hdr(sent_pkt);
    struct icmphdr *icmp_hdr=(struct icmphdr*)(sent_pkt+ETHER_HDR_SIZE+IP_BASE_HDR_SIZE)

    eh -> ether_type=htons(ETH_P_IP);

    rt_entry_t *entry=longest_prefix_match(ntohl(in_ip_hdr->saddr));
    ip_init_hdr(ip_hdr,entry->iface->ip,ntohl(in_ip_hdr->saddr),pkt_len-ETHER_HDR_SIZE,1

    icmp_hdr->code=code;
    icmp_hdr->type=type;

    if(type==0)
        {
        memcpy(sent_pkt+ETHER_HDR_SIZE+IP_HDR_SIZE(ip_hdr)+4,\
        in_pkt+ETHER_HDR_SIZE+IP_HDR_SIZE(in_ip_hdr)+4,pkt_len-(ETHER_HDR_SIZE+IP_HDR_SI
    }
        else
        {
        memset(sent_pkt+ETHER_HDR_SIZE+IP_HDR_SIZE(ip_hdr)+4,0,4);
                memcpy(sent_pkt+ETHER_HDR_SIZE+IP_HDR_SIZE(ip_hdr)+4+4,\
        in_ip_hdr,IP_HDR_SIZE(in_ip_hdr)+8);
    }

    icmp_hdr->checksum=icmp_checksum(icmp_hdr,pkt_len-ETHER_HDR_SIZE-IP_BASE_HDR_SIZE);
        ip_send_packet(sent_pkt,pkt_len);
}
```

Screenshots:

Host连通性测试:

h1 ping h2:

```
mininet> r1 ./router &
DEBUG: find the following interfaces:  r1-eth0 r1-eth1 r1-eth2.
mininet> h1 ping h2
PING 10.0.2.22 (10.0.2.22) 56(84) bytes of data.
64 bytes from 10.0.2.22: icmp_seq=1 ttl=63 time=1.81 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.313 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.415 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.456 ms
64 bytes from 10.0.2.22: icmp_seq=5 ttl=63 time=0.717 ms
64 bytes from 10.0.2.22: icmp_seq=6 ttl=63 time=0.638 ms
^C
--- 10.0.2.22 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5054ms
rtt min/avg/max/mdev = 0.313/0.724/1.806/0.502 ms
```

h1 ping h3:

```
mininet> h1 ping h3
PING 10.0.3.33 (10.0.3.33) 56(84) bytes of data.
64 bytes from 10.0.3.33: icmp_seq=1 ttl=63 time=0.413 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.387 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.673 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.106 ms
64 bytes from 10.0.3.33: icmp_seq=5 ttl=63 time=0.148 ms
64 bytes from 10.0.3.33: icmp_seq=6 ttl=63 time=0.066 ms
^C
--- 10.0.3.33 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5117ms
rtt min/avg/max/mdev = 0.066/0.298/0.673/0.214 ms
```

h2 ping h1:

```
mininet> h2 ping h1
PING 10.0.1.11 (10.0.1.11) 56(84) bytes of data.
64 bytes from 10.0.1.11: icmp_seq=1 ttl=63 time=0.064 ms
64 bytes from 10.0.1.11: icmp_seq=2 ttl=63 time=0.270 ms
64 bytes from 10.0.1.11: icmp_seq=3 ttl=63 time=0.144 ms
64 bytes from 10.0.1.11: icmp_seq=4 ttl=63 time=0.118 ms
64 bytes from 10.0.1.11: icmp_seq=5 ttl=63 time=0.065 ms
64 bytes from 10.0.1.11: icmp_seq=6 ttl=63 time=0.122 ms
^C
--- 10.0.1.11 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5097ms
rtt min/avg/max/mdev = 0.064/0.130/0.270/0.069 ms
```

h2 ping h3:

```
mininet> h2 ping h3
PING 10.0.3.33 (10.0.3.33) 56(84) bytes of data.
64 bytes from 10.0.3.33: icmp_seq=1 ttl=63 time=12.0 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.311 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.319 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.067 ms
64 bytes from 10.0.3.33: icmp_seq=5 ttl=63 time=0.100 ms
64 bytes from 10.0.3.33: icmp_seq=6 ttl=63 time=0.089 ms
^C
--- 10.0.3.33 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5096ms
rtt min/avg/max/mdev = 0.067/2.148/12.005/4.409 ms
```

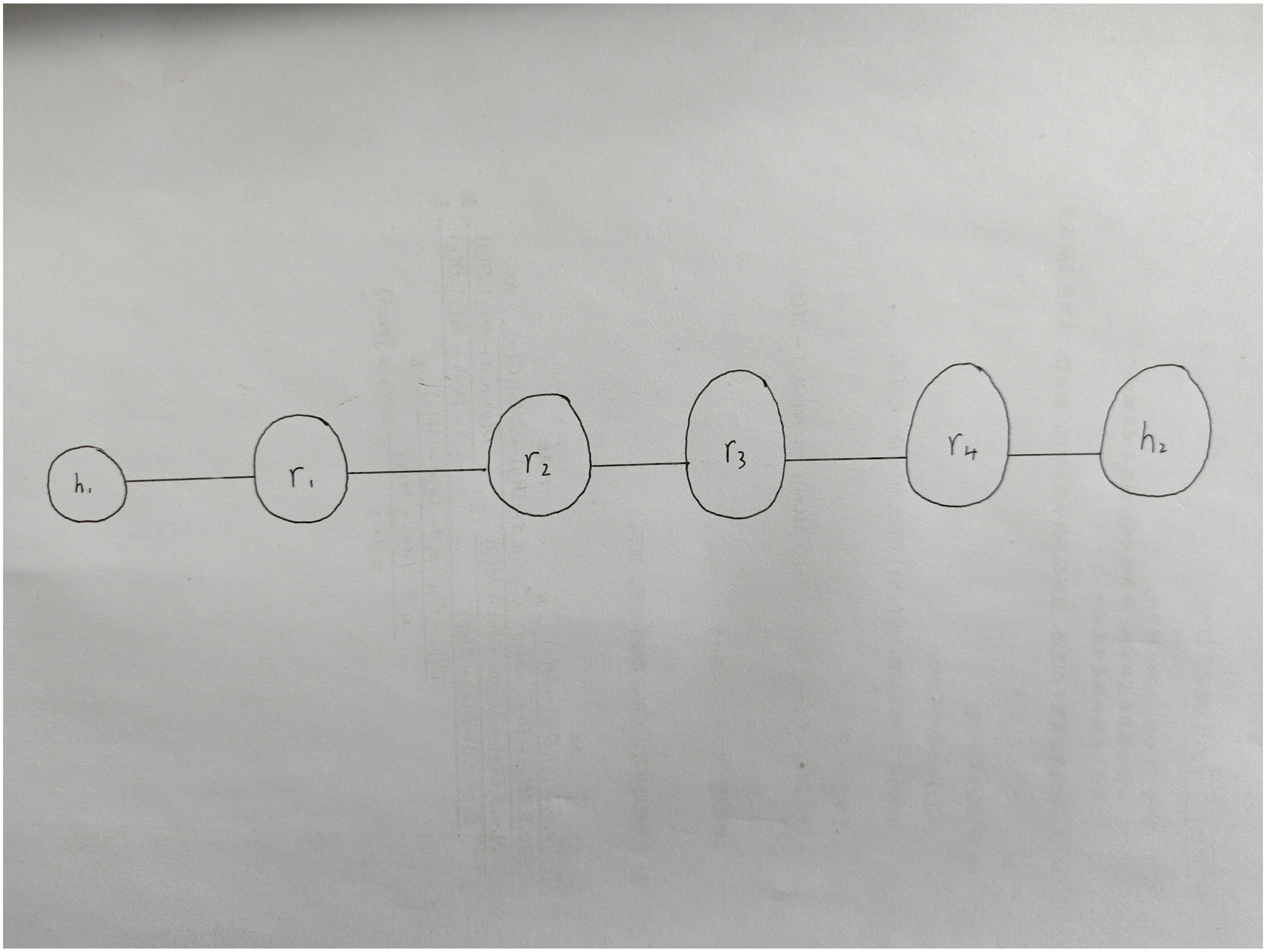
h3 ping h1:

```
mininet> h3 ping h1
PING 10.0.1.11 (10.0.1.11) 56(84) bytes of data.
64 bytes from 10.0.1.11: icmp_seq=1 ttl=63 time=0.085 ms
64 bytes from 10.0.1.11: icmp_seq=2 ttl=63 time=0.207 ms
64 bytes from 10.0.1.11: icmp_seq=3 ttl=63 time=0.168 ms
64 bytes from 10.0.1.11: icmp_seq=4 ttl=63 time=0.106 ms
64 bytes from 10.0.1.11: icmp_seq=5 ttl=63 time=0.155 ms
64 bytes from 10.0.1.11: icmp_seq=6 ttl=63 time=0.221 ms
^C
--- 10.0.1.11 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5179ms
rtt min/avg/max/mdev = 0.085/0.157/0.221/0.049 ms
```

h3 ping h2:

```
mininet> h3 ping h2
PING 10.0.2.22 (10.0.2.22) 56(84) bytes of data.
64 bytes from 10.0.2.22: icmp_seq=1 ttl=63 time=0.272 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.151 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.135 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.146 ms
64 bytes from 10.0.2.22: icmp_seq=5 ttl=63 time=0.146 ms
64 bytes from 10.0.2.22: icmp_seq=6 ttl=63 time=0.131 ms
^C
--- 10.0.2.22 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5110ms
rtt min/avg/max/mdev = 0.131/0.163/0.272/0.049 ms
```

手动构建的拓扑网络：




```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=32174>
<Host h2: h2-eth0:10.0.0.2 pid=32176>
<Host r1: r1-eth0:10.0.0.3,r1-eth1:None pid=32178>
<Host r2: r2-eth0:10.0.0.4,r2-eth1:None pid=32180>
<Host r3: r3-eth0:10.0.0.5,r3-eth1:None pid=32182>
<Host r4: r4-eth0:10.0.0.6,r4-eth1:None pid=32184>
```

连通性测试：

```
mininet> h1 ping h2
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=2.97 ms
64 bytes from 10.0.5.22: icmp_seq=2 ttl=60 time=2.45 ms
64 bytes from 10.0.5.22: icmp_seq=3 ttl=60 time=2.03 ms
64 bytes from 10.0.5.22: icmp_seq=4 ttl=60 time=3.10 ms
64 bytes from 10.0.5.22: icmp_seq=5 ttl=60 time=2.35 ms
^C
--- 10.0.5.22 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 2.032/2.579/3.098/0.397 ms
```

```
mininet> h2 ping h1
PING 10.0.1.11 (10.0.1.11) 56(84) bytes of data.
64 bytes from 10.0.1.11: icmp_seq=1 ttl=60 time=0.961 ms
64 bytes from 10.0.1.11: icmp_seq=2 ttl=60 time=0.437 ms
64 bytes from 10.0.1.11: icmp_seq=3 ttl=60 time=1.12 ms
64 bytes from 10.0.1.11: icmp_seq=4 ttl=60 time=1.86 ms
64 bytes from 10.0.1.11: icmp_seq=5 ttl=60 time=1.12 ms
^C
--- 10.0.1.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4035ms
rtt min/avg/max/mdev = 0.437/1.099/1.859/0.455 ms
```

traceout测试:

```
mininet> h1 traceroute h2
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)  2.501 ms  0.078 ms  0.010 ms
 2  10.0.2.2 (10.0.2.2)  0.071 ms  0.015 ms  0.115 ms
 3  10.0.3.2 (10.0.3.2)  0.355 ms  0.345 ms  0.336 ms
 4  10.0.4.2 (10.0.4.2)  0.548 ms  0.634 ms  0.516 ms
 5  10.0.5.22 (10.0.5.22)  0.506 ms  0.496 ms  0.141 ms
```

Remaining Bugs:

受制于能力，目前暂未发现