# Lab9 实验报告

高鸣驹 2019K8009907015

## 一. 实验任务

1. 实验路由器，可以处理 ARP 请求和应答，进行 ARP 缓存管理，IP 地址查找和 IP 数据包的转发，以及 ICMP 数据包的发送。
2. 手动构造一个包含多个路由器节点的网络，并且进行连通性测试和路径测试。

## 二. 实验设计

### 1. ARP 请求和应答

我们在处理 ARP 请求和应答包的时候，主要是组包，我们以请求包为例进行分析：

<table>
<tr><td colspan="3">Dest Ether Addr</td></tr>
<tr><td colspan="2">Dest Ether Addr (cont.)</td><td>Src Ether Addr</td></tr>
<tr><td colspan="3">Src Ether Addr (cont.)</td></tr>
<tr><td colspan="2">Proto Type (0x0806)</td><td>ARP Header (0x01)</td></tr>
<tr><td>ARP Proto (0x0800)</td><td>HW Addr Len (6)</td><td>Proto Addr Len (4)</td></tr>
<tr><td>ARP Operation Type</td><td colspan="2">Sender HW Addr</td></tr>
<tr><td colspan="3">Sender HW Addr (cont.)</td></tr>
<tr><td colspan="3">Sender Proto Addr</td></tr>
<tr><td colspan="3">Target HW Addr</td></tr>
<tr><td colspan="2">Target HW Addr (cont.)</td><td>Target Proto Addr</td></tr>
<tr><td>Target Proto Addr</td><td colspan="2"></td></tr>
</table>

图 1：ARP 包格式

我们的 Src Ether addr 可以从当前端口的配置信息得到，目标的 MAC 地址在不知道的时候填上全 1，协议号按照上表填写 0x8086 即可：

```
memset(eh -> ether_dhost, 0xff, ETH_ALEN);
memcpy(eh -> ether_shost, iface -> mac, ETH_ALEN);
eh -> ether_type = htons(ETH_P_ARP);
```

图 2：ARP 包的 Ether Header 填写

接下来我们填写 ARP Header 的内容，这里的 Sender Proto Addr 和 Target Proto Addr 为发送方和接收方的 IP 地址，Target HW Addr 不清楚的情况下设为全 0：

```
ea -> arp_hrd = htons(ARPHRD_ETHER);
ea -> arp_pro = htons(ETH_P_IP);
ea -> arp_hln = ETH_ALEN;
ea -> arp_pln = 4;
ea -> arp_op = htons(ARPOP_REQUEST);
memcpy(ea -> arp_sha, iface -> mac, ETH_ALEN);
ea -> arp_spa = htonl(iface -> ip);
memset(ea -> arp_tha, 0, ETH_ALEN);
ea -> arp_tpa = htonl(dst_ip);
```

图 3：ARP 包的 ARP Header 部分的填写

Reply 包的格式和 Request 包的格式基本一样，但是在 Reply 包中，这个包的目标的 MAC 地址我们是知道的，所以对应字段要填写上对应的 MAC 地址，而不能是全 0 或者全 1。

## 2. ARPCache 的管理

每个表项里存有 IP 地址和 MAC 地址的映射，同时还有这个表项是否有效的 valid 位，与这个表项加入的时间 added。

### a. 查找 IP 地址对应的 MAC 地址

这个比较直观，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，我们就找到了对应的 MAC 地址：

```
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;
}
```

图 4：查询 IP 地址对应的 MAC 地址

### b. 缓存查不到相应条目而等待 ARP 应答的数据包

我们的数据包查不到 IP 地址对应的 MAC 地址时，要挂到 ARPCache 的一个等待队列里，直到我们收到这个 IP 地址对应的 MAC 地址，再释放这个包。具体等待队列的结构如下：

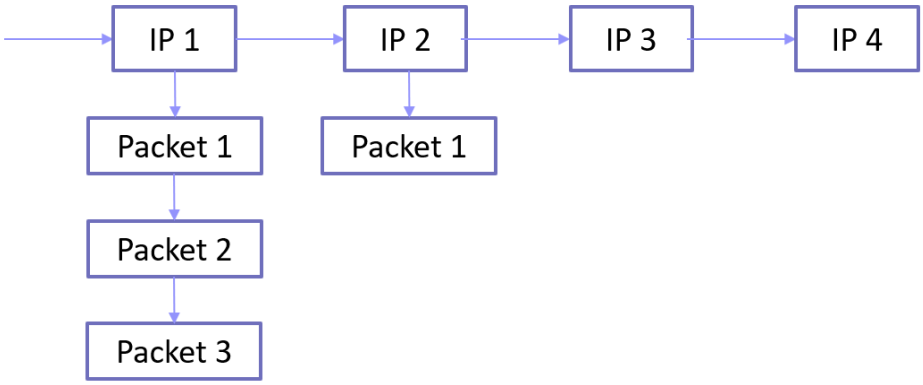


图 5：等待队列的结构

首先我们先寻找我们这个要挂载的这个包寻找的 IP 是否已经存在，如果存在，我们直接挂在这个 IP 的链表上：

```
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;
    }
}
```

图 6：直接挂到现有 IP 的等待队列里

如果没有，我们首先要创建对应的一个这个 IP 对应的 arp_seq 链表项，再把数据包挂到这个链表项的等待队列上。挂载完后，我们发出 arp 请求查询对应的 mac 地址：

```
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);
```

图 7：创建新的 arp_seq 然后挂载数据包

c. **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：

```c
// 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;
}
```

图 8：找到插入位置并插入

之后，我们需要释放所有的 pending packets，因为我们已经查到了他们所需要的 MAC 地址：

```c
// 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_packets), list) {
            struct ether_header *eth_hdr = (struct ether_header*)(pkt_entry -> packet);
            memcpy(eth_hdr -> 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);
    }
}
```

图 9：释放所有的 pending packets

d. **清理表项和包缓存**

这个主要有两部分，第一部分是清楚存在时间过长的表项，第二部分是清除请求次数过多的 pending packets。

首先我们记录下现在的时间，和表项加入的时间对比，如果表项的存在时间超过了最大时间，则删除表项，即置 valid = 0：

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

图 10：清除过期表项

接下来我们遍历所有的 arp_seq，看他们发 arp 请求包的次数，如果没超过最大限制则继续发请求包，如果超过了，我们清除这个 arp_seq 下面挂载的所有 pending packets，然后回复 icmp 数据包，回复目的地不可达：

```
    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->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;
        }
        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));
}
```

图 11：清除请求次数过多的 pending packets

### 3. ICMP 数据包发送

icmp 数据包的格式如下：

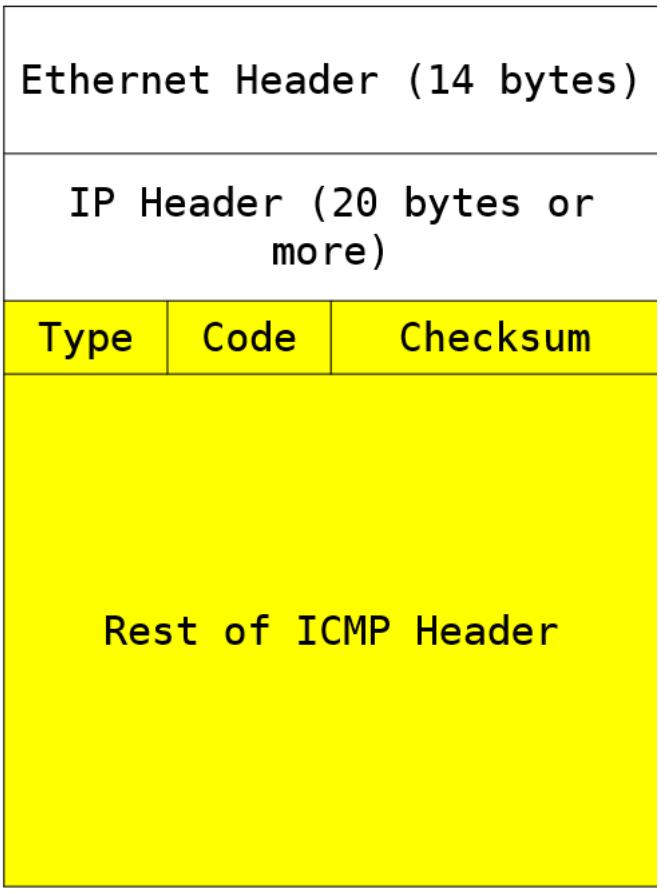


图 12：icmp 的数据包格式

我们首先确定 ICMP 数据包的长度，当收到 ping 包时，我们的长度和 ping 包是一样的，但是其它的，我们的 rest of icmp header 要拷贝收到的 IP 数据包的头部和随后的 8 字节：

```
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;
}
```

图 13：icmp 包长度的确定

确定长度后，我们开始填充各部分的内容，Ethernet Header 我们只需要填充 type 就可以，其他字段在发送时会自动填上，IP 字段需要我们查找本端口的 ip 地址，然后调用 ip_init_hdr 函数进行 ip 字段的填充，最后的 icmp 的 type 和 code 直接填即可：

```
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;
```

图 14：icmp 包部分字段的填充

接下来就是 rest of icmp header 的拷贝，具体的拷贝内容开始已经提过，逻辑如下：

```
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_SIZE(in_ip_hdr) + 4));
} 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);
}
```

图 15：rest of icmp header 的填充

最后发送这个数据包。

## 4. IP 地址查找与数据包的转发

我们首先完成转发 ip 数据包的操作，首先我们需要进行最长的前缀匹配，即遍历路由表，找到对应的路由器表项且 mask 位数最多，然后进行转发。

```
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;
}
```

图 26：最长前缀匹配

在得到对应的路由器表项后，我们看它的网关地址，如果为 0，说明数据包已经到达了目的主机，这时我们下一跳地址设为目的地址，否则为查到的路由器表项的网关地址：

```c
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);
}
```

图 27：路由器转发的逻辑

我们收到 ip 数据包，也要对数据包进行解析，这就需要 handle_ip_packet 函数。

首先我们先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 echo 请求，我们返回相应的 reply 即可：

```c
// 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;
}
```

图 28：ICMP ECHO 数据包的处理

第二部分我们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：

```c
// 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;
}
```

图 29：ICMP_DEST_UNREACH 的处理

第三部分我们处理是否超时：

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

图 30：超时的处理

最后一部分，我们处理 ip 数据包的转发：

```
    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);
```

图 31：ip 数据包的转发

# 三. 实验过程

## 1. 路由器功能的测试

我们在 h1 上 ping h2, h3：

```
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# ping 10.0.2.22
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=0.257 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.288 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.065 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.104 ms
^Z
[2]+  Stopped                 ping 10.0.2.22
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# ping 10.0.3.33
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=0.065 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.061 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.067 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.066 ms
^Z
```

图 32：在 h1 上 ping h2, h3 的结果

我们可以看出，可以 ping 通。

我们 ping 10.0.3.11：

```
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# ping 10.0.3.11
PING 10.0.3.11 (10.0.3.11) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Host Unreachable
From 10.0.1.1 icmp_seq=2 Destination Host Unreachable
From 10.0.1.1 icmp_seq=3 Destination Host Unreachable
From 10.0.1.1 icmp_seq=4 Destination Host Unreachable
From 10.0.1.1 icmp_seq=5 Destination Host Unreachable
From 10.0.1.1 icmp_seq=6 Destination Host Unreachable
From 10.0.1.1 icmp_seq=7 Destination Host Unreachable
^Z
```

图 33：在 h1 Ping 10.0.3.11 的结果

我们可以看出，这里符合预期结果：Destination Host Unreachables

我们接着 ping 10.0.4.1：

```
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# ping 10.0.4.1
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Net Unreachable
From 10.0.1.1 icmp_seq=2 Destination Net Unreachable
From 10.0.1.1 icmp_seq=3 Destination Net Unreachable
^Z
```

图 34：在 h1 ping 10.0.4.1

符合预期：Destination Net Unreachable.

## 2. 连通性和路径测试

我们的结构如下图所示：

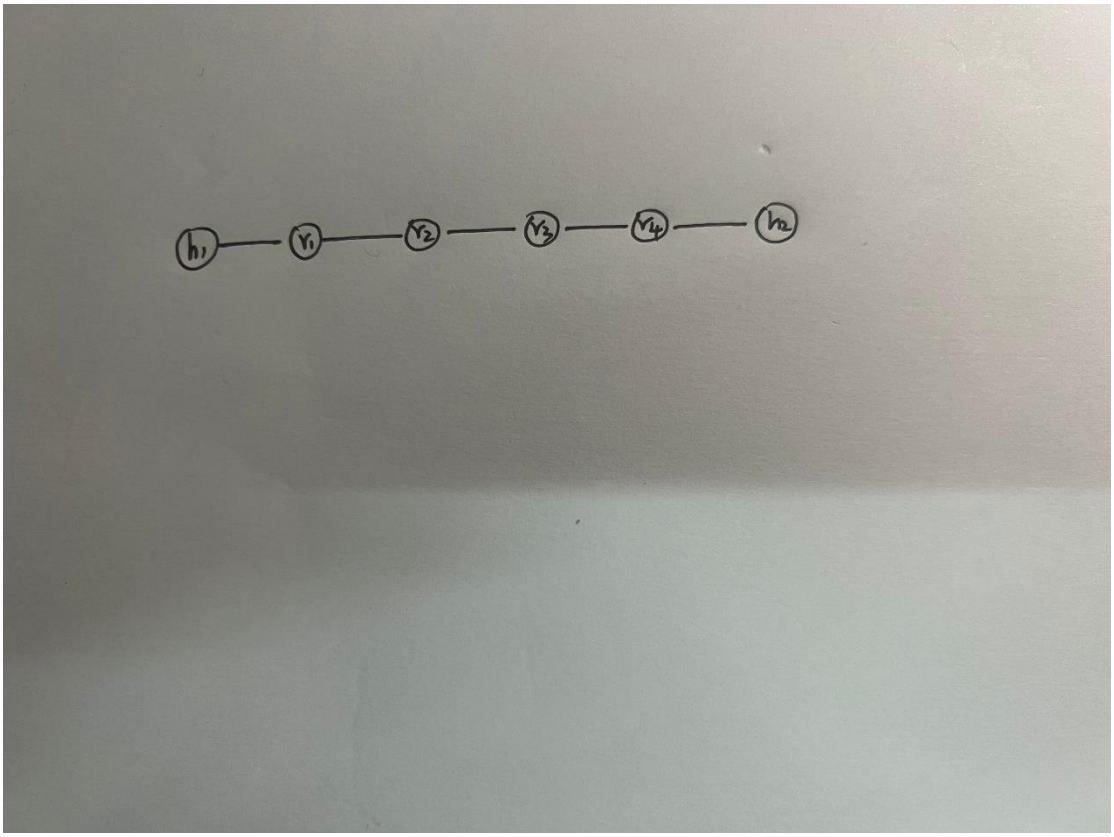


图 35：拓扑结构

我们进行连通性测试：

```
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# ping 10
.0.5.22
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=2.27 ms
64 bytes from 10.0.5.22: icmp_seq=2 ttl=60 time=0.781 ms
64 bytes from 10.0.5.22: icmp_seq=3 ttl=60 time=0.726 ms
64 bytes from 10.0.5.22: icmp_seq=4 ttl=60 time=0.810 ms
^Z
[2]+  Stopped                 ping 10.0.5.22
```

图 36：h1 ping h2

```
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# ping 10
.0.1.11
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=2.01 ms
64 bytes from 10.0.1.11: icmp_seq=2 ttl=60 time=0.832 ms
64 bytes from 10.0.1.11: icmp_seq=3 ttl=60 time=0.620 ms
64 bytes from 10.0.1.11: icmp_seq=4 ttl=60 time=0.447 ms
64 bytes from 10.0.1.11: icmp_seq=5 ttl=60 time=0.514 ms
64 bytes from 10.0.1.11: icmp_seq=6 ttl=60 time=0.502 ms
64 bytes from 10.0.1.11: icmp_seq=7 ttl=60 time=0.442 ms
z64 bytes from 10.0.1.11: icmp_seq=8 ttl=60 time=0.490 ms
64 bytes from 10.0.1.11: icmp_seq=9 ttl=60 time=0.486 ms
^Z
[1]+  Stopped                 ping 10.0.1.11
```

图 37：h2 ping h1

我们可知，连通性测试通过。

我们的 traceout 测试：

```
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router# tracero
ute 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)  0.741 ms  0.707 ms  0.704 ms
 2  10.0.2.2 (10.0.2.2)  1.455 ms  1.456 ms  1.527 ms
 3  10.0.3.2 (10.0.3.2)  4.544 ms  4.727 ms  4.727 ms
 4  10.0.4.2 (10.0.4.2)  5.210 ms  5.305 ms  5.305 ms
 5  10.0.5.22 (10.0.5.22)  5.304 ms  5.302 ms  5.302 ms
root@gasaiyuno-VirtualBox:/home/gasaiyuno/Desktop/cnlab/lab9/09-router#
```

图 38：h2 ping h1 的 traceroute 测试

我们可以看出经过的路径是正确的，从 r1-r2-r3-r4-h2:

```
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')
r2.cmd('ifconfig r2-eth0 10.0.2.2/24')
r2.cmd('ifconfig r2-eth1 10.0.3.1/24')
r2.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.2.1 dev r2-eth0')
r2.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.3.2 dev r2-eth1')
r2.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.3.2 dev r2-eth1')
r3.cmd('ifconfig r3-eth0 10.0.3.2/24')
r3.cmd('ifconfig r3-eth1 10.0.4.1/24')
r3.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.3.1 dev r3-eth0')
r3.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.3.1 dev r3-eth0')
r3.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.4.2 dev r3-eth1')
r4.cmd('ifconfig r4-eth0 10.0.4.2/24')
r4.cmd('ifconfig r4-eth1 10.0.5.1/24')
r4.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.4.1 dev r4-eth0')
r4.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.4.1 dev r4-eth0')
r4.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.4.1 dev r4-eth0')
```

图 39：r1-r4 的地址