Lab of Computer Network: Router **Fall 2024**

Report 7 — October 29

Lecturer: Wu Qinghua *Completed by: 2022K8009929010 Zhang Jiawei*

7.1 实验内容

1. (a) 在主机上安装 arptables, iptables，用于禁止每个节点的相应功能。
   (b) 运行给定网络拓扑：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；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。
   (d) 在 h1 上进行 ping 实验：Ping 10.0.1.1 (r1)，能够 ping 通；Ping 10.0.2.22 (h2)，能够 ping 通；Ping 10.0.3.33 (h3)，能够 ping 通；Ping 10.0.3.11，返回 ICMP Destination Host Unreachable；Ping 10.0.4.1，返回 ICMP Destination Net Unreachable。
2. (a) 构造一个包含多个路由器节点组成的网络：手动配置每个路由器节点的路由表；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：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：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。

7.2 实验过程

7.2.1 IP 数据包转发

对于路由器接收到的 IP 报文，在转发之前需要先进行一次判定。倘若该报文为 ICMP 协议下的请求回复报文 (ICMP_ECHOREQUEST) 且目的地址为路由器的入端口 IP 地址，直接向源地址发送 ICMP 回复报文 (ICMP_ECHOREPLY)：

```
struct iphdr *iphdr = packet_to_ip_hdr(packet);
struct icmphdr *icmphdr = (struct icmphdr *)IP_DATA(iphdr);
u32 dest = ntohl(iphdr->daddr);
u8 protocol = iphdr->protocol;
u8 type = icmphdr->type;

// check if the packet is ICMP echo request and the destination IP address is
    equal to the IP address of the iface
if (dest == iface->ip && protocol == IPPROTO_ICMP && type == ICMP_ECHOREQUEST) {
  icmp_send_packet(packet, len, ICMP_ECHOREPLY, 0);
  free(packet);
```

```
    return;
  }
```

对于其他情况，路由器需要根据路由表进行转发。首先，对 IP 报头的 TTL 字段进行减一操作，若 TTL 字段减为 0，则将该数据包丢弃，并回复 ICMP 信息 (`ICMP_EXC_TTL`)。然后更新校验和，根据目的 IP 地址查找路由表，若找到对应的路由表项，则根据下一跳 IP 地址进行转发；若未找到对应的路由表项，则回复 ICMP 信息 (`ICMP_NET_UNREACH`)。下一跳 IP 地址的确定则需要根据路由表项的下一跳网关 (gateway) 地址进行判断，若网关地址为 0，说明目的主机在本地网络中，直接向目的主机发送数据包；若网关地址不为 0，说明目的主机在其他网络中，将数据包发送给网关：

```
  // forward the packet
  iphdr->ttl--;

  // check if the TTL is less than or equal to 0
  if (iphdr->ttl <= 0) {
    icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL);
    free(packet);
    return;
  }
  // update the checksum of the IP header
  iphdr->checksum = ip_checksum(iphdr);
  // search the routing table for the longest prefix match
  rt_entry_t *match = longest_prefix_match(dest);
  if (match == NULL) {
    icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
    free(packet);
    return;
  }

  // check if the destination IP address is in the same network with the iface
  u32 nextip;
  if (match->gw == 0)
    nextip = dest;
  else
    nextip = match->gw;
  iface_send_packet_by_arp(match->iface, nextip, packet, len);
```

还需要说明的是，查询路由表使用的是最长前缀匹配方法，若目的地址与路由表中的目的地址匹配，则比较掩码长度，选择掩码长度最长的路由表项作为匹配项：

```
   // lookup in the routing table, to find the entry with the same and longest
      prefix.
   // the input address is in host byte order
```

```c
rt_entry_t *longest_prefix_match(u32 dst)
{
    // fprintf(stderr, "TODO: longest prefix match for the packet.\n");
    rt_entry_t *entry = NULL, *match = NULL;
    list_for_each_entry(entry, &rtable, list) {
        if ((dst & entry->mask) == (entry->dest & entry->mask)) {
            if (match == NULL || entry->mask > match->mask)
                match = entry;
        }
    }
    return match;
}
```

7.2.2 ARP 数据包处理

对于路由器接收到的 ARP 报文，首先分析报文的类型。若为 ARP 请求报文 (ARPOP_REQUEST)，则判断目的 IP 地址是否为路由器的入端口 IP 地址，若是则回复 ARP 应答报文，且将该 ARP 请求报文的源 MAC 地址和 IP 地址添加到 ARP 缓存中；若为 ARP 回复报文 (ARPOP_REPLY)，则说明该报文是对路由器发送的 ARP 请求报文的回复，当回复的目的 IP 地址为路由器的入端口 IP 地址时，将该 ARP 回复报文的源 MAC 地址和 IP 地址添加到 ARP 缓存中：

```c
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    // fprintf(stderr, "TODO: process arp packet: arp request & arp reply.\n");
    struct ether_arp *arp = (struct ether_arp *)(packet + ETHER_HDR_SIZE);

    // arp request
    if (ntohs(arp->arp_op) == ARPOP_REQUEST) {
        if (ntohl(arp->arp_tpa) == iface->ip) {
            arpcache_insert(ntohl(arp->arp_spa), arp->arp_sha);
            arp_send_reply(iface, arp);
        }
    }
    // arp reply
    else if (ntohs(arp->arp_op) == ARPOP_REPLY)
        if (ntohl(arp->arp_tpa) == iface->ip)
            arpcache_insert(ntohl(arp->arp_spa), arp->arp_sha);
    else
        fprintf(stderr, "Unknown ARP packet\n");
    free(packet);
}
```

当然，我们也需要发送 ARP 报文，仍然分为 ARP 请求报文和 ARP 回复报文。这两个报文的构造比较类似，首先申请空间然后填入相应的字段，需要注意的是，ARP 请求报文的目的 MAC 地址为广播地址(全 1)，而 ARP 回复报文的目的 MAC 地址为 ARP 请求报文的源 MAC 地址：

```
// send an arp request: encapsulate an arp request packet, send it out through
// iface_send_packet
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    // fprintf(stderr, "TODO: send arp request when lookup failed in arpcache.\n");
    char *packet = (char *)malloc(ETHER_HDR_SIZE + sizeof(struct ether_arp));
    memset(packet, 0, ETHER_HDR_SIZE + sizeof(struct ether_arp));

    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(eh->ether_dhost, "\xff\xff\xff\xff\xff\xff", ETH_ALEN);
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    eh->ether_type = htons(ETH_P_ARP);

    struct ether_arp *arp = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
    arp->arp_hrd = htons(ARPHRD_ETHER);
    arp->arp_pro = htons(ETH_P_IP);
    arp->arp_hln = ETH_ALEN;
    arp->arp_pln = 4;
    arp->arp_op = htons(ARPOP_REQUEST);
    memcpy(arp->arp_sha, iface->mac, ETH_ALEN);
    arp->arp_spa = htonl(iface->ip);
    memset(arp->arp_tha, 0, ETH_ALEN);
    arp->arp_tpa = htonl(dst_ip);

    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));
}

// send an arp reply packet: encapsulate an arp reply packet, send it out
// through iface_send_packet
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)
{
    // fprintf(stderr, "TODO: send arp reply when receiving arp request.\n");
    char *packet = (char *)malloc(ETHER_HDR_SIZE + sizeof(struct ether_arp));
    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(eh->ether_dhost, req_hdr->arp_sha, ETH_ALEN);
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    eh->ether_type = htons(ETH_P_ARP);

    struct ether_arp *arp = (struct ether_arp *)(packet + ETHER_HDR_SIZE);
```

```c
    arp->arp_hrd = htons(ARPHRD_ETHER);
    arp->arp_pro = htons(ETH_P_IP);
    arp->arp_hln = ETH_ALEN;
    arp->arp_pln = 4;
    arp->arp_op = htons(ARPOP_REPLY);
    memcpy(arp->arp_sha, iface->mac, ETH_ALEN);
    arp->arp_spa = htonl(iface->ip);
    memcpy(arp->arp_tha, req_hdr->arp_sha, ETH_ALEN);
    arp->arp_tpa = req_hdr->arp_spa;

    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));
}
```

7.2.3 ARP 缓存管理

在之前提到的 IP 数据包转发中，最后是通过查找 ARP 缓存来确定下一跳的 MAC 地址。在本实验中，ARP 缓存的管理主要包括查找 ARP 缓存、添加待应答数据包、插入 ARP 缓存和清理 ARP 缓存三个操作。

查找操作是根据目的 IP 地址查找有效 ARP 缓存中是否存在对应的 MAC 地址，若存在则返回 1，否则返回 0：

```c
// lookup the IP->mac mapping
//
// traverse the table to find whether there is an entry with the same IP
// and mac address with the given arguments
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    // fprintf(stderr, "TODO: lookup ip address in arp cache.\n");
    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;
}
```

但是，ARP 缓存应答并非是立即返回的，而是需要等待 ARP 应答报文的到来。因此，需要添加待应答数据包，即将数据包添加到 ARP 缓存的待应答数据包队列中。这里又分为两种情况，一种是同一 IP 地址的数据包已经在待应答数据包队列中，这说明 ARP 请求报文已经发送过了，只需要将数据包添加到该 IP 地址的待应答

数据包队尾；另一种是同一 IP 地址的数据包不在待应答数据包队列中，这说明 ARP 请求报文还未发送，则需要新建与该 IP 地址对应的待应答数据包队列，然后发送 ARP 请求报文：

```c
// append the packet to arpcache
//
// Lookup in the list which stores pending packets, if there is already an
// entry with the same IP address and iface (which means the corresponding arp
// request has been sent out), just append this packet at the tail of that entry
// (the entry may contain more than one packet); otherwise, malloc a new entry
// with the given IP address and iface, append the packet, and send arp request.
void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len)
{
   // fprintf(stderr, "TODO: append the ip address if lookup failed, and send arp
       request if necessary.\n");
   pthread_mutex_lock(&arpcache.lock);
   struct arp_req *req_entry = NULL;
   struct arp_req *req_q;

   // find the entry with the same IP address
   list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
      if (req_entry->ip4 == ip4) {
         struct cached_pkt *pkt_entry = (struct cached_pkt *)malloc(sizeof(struct
            cached_pkt));
         init_list_head(&(pkt_entry->list));
         pkt_entry->packet = packet;
         pkt_entry->len = len;
         list_add_tail(&(pkt_entry->list), &(req_entry->cached_packets));
         pthread_mutex_unlock(&arpcache.lock);
         return;
      }
   }

   // no entry found, malloc a new entry
   req_entry = (struct arp_req *)malloc(sizeof(struct arp_req));
   init_list_head(&(req_entry->list));
   init_list_head(&(req_entry->cached_packets));
   req_entry->iface = iface;
   req_entry->ip4 = ip4;
   req_entry->sent = time(NULL);
   req_entry->retries = 0;
   list_add_tail(&(req_entry->list), &(arpcache.req_list));

   struct cached_pkt *pkt_entry = (struct cached_pkt *)malloc(sizeof(struct
       cached_pkt));
```

```
    pkt_entry->packet = packet;
    pkt_entry->len = len;
    list_add_tail(&(pkt_entry->list), &(req_entry->cached_packets));

    pthread_mutex_unlock(&arpcache.lock);
    arp_send_request(iface, ip4);
}
```

并不是所有的 IP-MAC 映射都存在于缓存中的。当需要向 ARP 缓存中插入新的 ARP 缓存项时，首先查找是否存在相同的 IP 地址，若存在则更新该 ARP 缓存项的 MAC 地址和时间戳；若不存在则查找是否有无效的 ARP 缓存项，若有则直接覆盖内容，倘若所有 ARP 缓存项都是有效的，则取随机条目进行覆盖。之后再查找待应答数据包队列中是否有该 IP 地址的数据包，若有则直接填写目的 MAC 地址并发送：

```
// insert the IP->mac mapping into arpcache, if there are pending packets
// waiting for this mapping, fill the ethernet header for each of them, and send
// them out
void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
{
    // fprintf(stderr, "TODO: insert ip->mac entry, and send all the pending
        packets.\n");
    pthread_mutex_lock(&arpcache.lock);
    int i;

    // update the entry if it already exists
    for (i = 0; i < MAX_ARP_SIZE; i++) {
        if (arpcache.entries[i].valid && arpcache.entries[i].ip4 == ip4){
            arpcache.entries[i].added = time(NULL);
            memcpy(arpcache.entries[i].mac, mac, ETH_ALEN);
            pthread_mutex_unlock(&arpcache.lock);
            return;
        }
    }

    // search for an empty entry
    for (i = 0; i < MAX_ARP_SIZE; i++)
        if (!arpcache.entries[i].valid)
            break;

    // if no empty entry, replace a random entry
    if (i == MAX_ARP_SIZE)
        i = rand() % MAX_ARP_SIZE;

    arpcache.entries[i].ip4 = ip4;
```

```
    arpcache.entries[i].added = time(NULL);
    arpcache.entries[i].valid = 1;
    memcpy(arpcache.entries[i].mac, mac, ETH_ALEN);

    // send all the pending packets with the same IP address of the new entry
    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) {
                memcpy(pkt_entry->packet, 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);
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
}
```

ARP 缓存也有相应的老化和清理机制，当 ARP 缓存项的时间戳超过 15 秒时，清除该 ARP 缓存项。而且，当每一秒发送一次的 ARP 请求报文的重传次数超过 5 次但仍未收到 ARP 回复报文时，对相应等待队列中的数据包回复 ICMP 无法到达信息 (ICMP_HOST_UNREACH) 并删除数据包：

```
// sweep arpcache periodically
//
// For the IP->mac entry, if the entry has been in the table for more than 15
// seconds, remove it from the table.
// For the pending packets, if the arp request is sent out 1 second ago, while
// the reply has not been received, retransmit the arp request. If the arp
// request has been sent 5 times without receiving arp reply, for each
// pending packet, send icmp packet (DEST_HOST_UNREACHABLE), and drop these
// packets.
void *arpcache_sweep(void *arg)
{
    while (1) {
        sleep(1);
        // fprintf(stderr, "TODO: sweep arpcache periodically: remove old entries,
            resend arp requests .\n");
```

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

        struct list_head temp;
        init_list_head(&temp);

        struct arp_req *req_entry = NULL, *req_q;
        list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
            if (time(NULL) - req_entry->sent >= 1) {
                req_entry->retries++;
                req_entry->sent = time(NULL);
                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) {
                        list_delete_entry(&(pkt_entry->list));
                        list_add_tail(&(pkt_entry->list), &temp);
                    }
                    list_delete_entry(&(req_entry->list));
                    free(req_entry);
                }
                else
                    arp_send_request(req_entry->iface, req_entry->ip4);
            }
        }
        pthread_mutex_unlock(&arpcache.lock);

        struct cached_pkt *pkt_entry = NULL, *pkt_q;
        list_for_each_entry_safe(pkt_entry, pkt_q, &temp, list) {
            icmp_send_packet(pkt_entry->packet, pkt_entry->len, ICMP_DEST_UNREACH,
                ICMP_HOST_UNREACH);
            free(pkt_entry);
        }
    }

    return NULL;
}
```

7.3 ICMP 数据包处理

ICMP 数据包的构造需要花一些功夫。申请存储空间时需要注意数据包的长度，对于非 ICMP_ECHOREPLY 报文，长度为以太网头 +IP 头 +ICMP 头 + 原数据包；对于 ICMP_ECHOREPLY 报文，长度计算简化为原数据包长度-原数据包 IP 头长度 +IP 头长度。然后填充相应的字段，对于 ICMP_ECHOREPLY 报文，源 IP 地址就是数据包的目的 IP 地址，其余情况下则查询路由表，找到下一跳网关的 IP 地址并填充；目的 IP 地址则是数据包的源 IP 地址；ICMP 类型和代码直接填充；校验和需要计算。最后即可发送：

```c
// send icmp packet
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    // fprintf(stderr, "TODO: malloc and send icmp packet.\n");
    struct iphdr *iphdr = packet_to_ip_hdr(in_pkt);
    char *ipdata = IP_DATA(iphdr);

    // calculate icmp packet length
    int icmp_len;
    if (type == ICMP_ECHOREPLY)
        icmp_len = ntohs(iphdr->tot_len) - IP_HDR_SIZE(iphdr);
    else
        icmp_len = ICMP_HDR_SIZE + IP_HDR_SIZE(iphdr) + ICMP_COPIED_DATA_LEN;
    int res_len = IP_BASE_HDR_SIZE + ETHER_HDR_SIZE + icmp_len;

    char *res = (char *)malloc(res_len);
    memset(res, 0, res_len);

    // fill ip header
    struct iphdr *res_iphdr = packet_to_ip_hdr(res);
    if (type == ICMP_ECHOREPLY)
        ip_init_hdr(res_iphdr, ntohl(iphdr->daddr), ntohl(iphdr->saddr), icmp_len +
            IP_BASE_HDR_SIZE, IPPROTO_ICMP);
    else{
        rt_entry_t *match = longest_prefix_match(ntohl(iphdr->saddr));
        if (match == NULL) {
            free(res);
            return;
        }
        ip_init_hdr(res_iphdr, match->iface->ip, ntohl(iphdr->saddr), icmp_len +
            IP_BASE_HDR_SIZE, IPPROTO_ICMP);
    }

    char *res_ipdata = IP_DATA(res_iphdr);
```

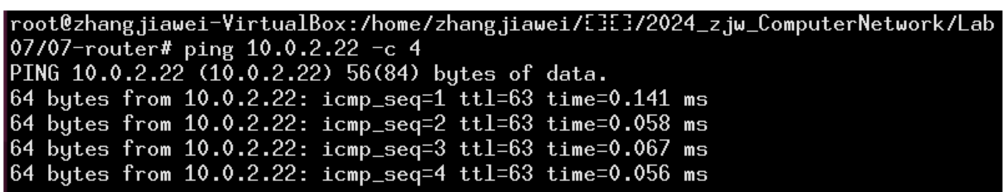
```c
    // fill icmp header and data
    struct icmphdr *res_icmphdr = (struct icmphdr *)res_ipdata;
    if (type == ICMP_ECHOREPLY)
        memcpy(res_ipdata, ipdata, icmp_len);
    else
        memcpy(res_ipdata + ICMP_HDR_SIZE, iphdr, icmp_len - ICMP_HDR_SIZE);

    res_icmphdr->type = type;
    res_icmphdr->code = code;
    res_icmphdr->checksum = icmp_checksum(res_icmphdr, icmp_len);

    // send icmp packet
    ip_send_packet(res, res_len);
}
```

ICMP 数据包的发出过程与发送 IP 数据包类似，都是以同样的方式确定下一跳的 IP 地址，最后以 ARP 机制发送数据包，这里不再赘述。

7.4 实验结果

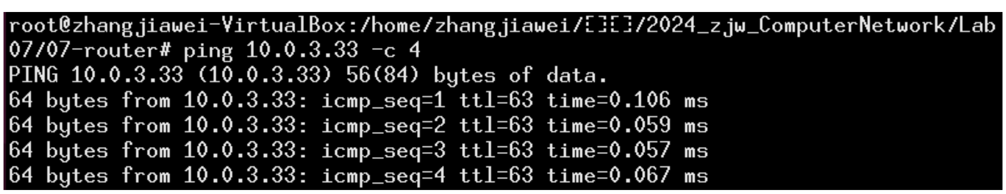
7.4.1 给定网络拓扑

实验结果如下：



```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/□□/2024_zjw_ComputerNetwork/Lab
07/07-router# ping 10.0.2.22 -c 4
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=0.141 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.058 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.067 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.056 ms
```

(a) h1 ping h2



```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/□□/2024_zjw_ComputerNetwork/Lab
07/07-router# ping 10.0.3.33 -c 4
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=0.106 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.059 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.057 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.067 ms
```

(b) h1 ping h3

可见，h1 能够 ping 通 h2、h3 和 r1，但无法 ping 通其他主机，说明实验结果符合预期。

7.4.2 自定义网络拓扑

自定义网络拓扑如下：

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/□□/2024_zjw_ComputerNetwork/Lab
07/07-router# ping 10.0.1.1 -c 4
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=0.351 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.074 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.043 ms
```

(c) h1 ping r1

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/□□/2024_zjw_ComputerNetwork/Lab
07/07-router# ping 10.0.3.11 -c 4
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
```

(d) h1 ping unreachble1

```
root@zhangjiawei-VirtualBox:/home/zhangjiawei/□□/2024_zjw_ComputerNetwork/Lab
07/07-router# ping 10.0.4.1 -c 4
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
From 10.0.1.1 icmp_seq=4 Destination Net Unreachable
```

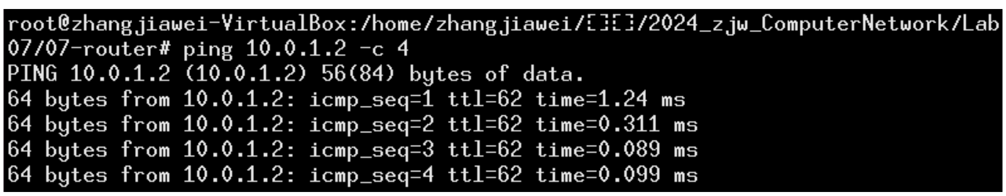
(e) h1 ping unreachble2

图 7.1. 给定网络拓扑实验结果

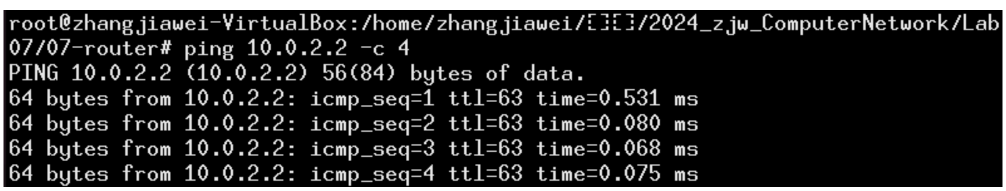
```python
h1, h2, r1, r2, r3 = net.get('h1', 'h2', 'r1', 'r2', 'r3')

# 配置 IP 地址
h1.cmd('ifconfig h1-eth0 10.0.1.1/24')
h2.cmd('ifconfig h2-eth0 10.0.4.1/24')

r1.cmd('ifconfig r1-eth0 10.0.1.2/24')
r1.cmd('ifconfig r1-eth1 10.0.2.1/24')

r2.cmd('ifconfig r2-eth0 10.0.2.2/24')
r2.cmd('ifconfig r2-eth1 10.0.3.1/24')

r3.cmd('ifconfig r3-eth0 10.0.3.2/24')
r3.cmd('ifconfig r3-eth1 10.0.4.2/24')

# 配置路由表
h1.cmd('route add default gw 10.0.1.2')
h2.cmd('route add default gw 10.0.4.2')

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')
```

```python
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')

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')

# 执行脚本以禁用某些功能
for n in (h1, h2, r1, r2, r3):
    n.cmd('./scripts/disable_offloading.sh')
    n.cmd('./scripts/disable_ipv6.sh')

for r in (r1, r2, r3):
    n.cmd('./scripts/disable_arp.sh')
    n.cmd('./scripts/disable_icmp.sh')
    n.cmd('./scripts/disable_ip_forward.sh')
    n.cmd('./scripts/disable_ipv6.sh')
```

实验结果如下:



(a) h2 ping r1



(b) h2 ping r2

图 7.2. 自定义网络拓扑 ping 实验结果

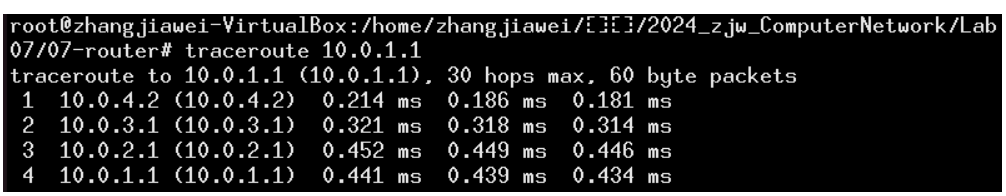


图 7.3. 自定义网络拓扑 traceroute 实验结果

可见，h2 能够 ping 通 r1 和 r2，traceroute 也能够正确输出路径上每个节点的 IP 信息，说明实验结果符合预期。

7.5 实验总结

这是一次任务量较大的实验，但同时也是一次收获颇丰的实验。通过本次实验，我学会了如何处理 ARP 数据包、IP 数据包和 ICMP 数据包，实现了路由器的基本功能。在实验过程中，我对路由表的查找、ARP 缓存的管理、ICMP 数据包的构造和发送等方面有了更深入的了解，对网络协议栈的实现有了更深刻的认识。同时，我也学会了如何构建自定义网络拓扑，进行 ping 和 traceroute 实验，对网络通信有了更深入的了解。