路由器转发实验

学号： 2021K8009929010

姓名： 贾城昊

一、 实验题目：路由器转发实验

二、 实验任务：

了解路由器转发的原理，路由表结构和最长前缀匹配查找方法。了解路由器转发数据包的流程，实现路由器查询和转发模块。了解 ARP 协议内容，学习如何构建 ARP 缓存表以及如何实现 ARP 缓存机制，了解 ARP 协议报文格式，实现处理 ARP 请求和应答。了解 ICMP 协议格式，在路由器遇到路由表查找失败、ARP 查询失败、TTL 为 0 时能发送 ICMP 报文，并且支持处理 ping 请求报文。

三、 实验流程

1. 安装 arptables 和 iptables 以禁止每个节点的固有功能;安装 traceout 以实现路径测试。
2. 编写并实现路由器的 IP 包处理、ICMP 包生成、ARP 包生成处理、ARP 缓存维护等路由器功能。
3. 运行给定网络拓扑 router_topo.py，验证 ping 功能在各种情况下均处理正常。
4. 构造包含多个路由器节点的网络拓扑，手动配置路由表，并测试连通性，完成路径测试。

四、 实验过程

（一）IP 数据包的处理

1. IP 包分析

本实验中路由器将收到的报文通过以太网帧的 ether_type 分为两类，ARP 报文和 IP 报文，分别处理。对于 IP 报文，有一种情况需要特别处理，那就是该报文是 ICMP 协议的 ECHOREQUEST (请求回复)报文并且其目标 ip 地址为该端口的 ip 地址。此时不需要转发数据包，而是需要向源地址发送一个 ICMP 回复报文，即生成并发送该包对应的 ICMP 的 ping_echo 包。而对于其他情况的 IP 数据包(包括其他 ICMP 报文) ，正常转发即可。

正常转发需要依次检查路由表条目、数据包 TTL 值，其流程中有 2 种特殊情况，路由表查找失败、TTL 为 0，此时无法转发，向源发送相应 ICMP 的报错报文，回复 ICMP 的 NET_UNREACH 或 EXC_TTL 信息，并结束处理。

值得注意的是在路由器端口与目的地址不在同一网段时，应以路由表条目中的对应 IP 地址（即下一跳网关）作为下一跳地址;否则，若下一跳网关为 0，则意味着目的主机在本网络内，直接向在同一网段的目的 IP 发送即可。

具体代码如下所示:

```c
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    //fprintf(stderr, "TODO: handle ip packet.\n");
    struct iphdr *iph =  packet_to_ip_hdr(packet);
    struct icmphdr *icmph = (struct icmphdr *) IP_DATA(iph);

    u32 daddr = ntohl(iph->daddr);
    u8 protocol = iph->protocol;
    u8 type = icmph->type;

    if((daddr==iface->ip) && (protocol==IPPROTO_ICMP) && (type==ICMP_ECHOREQUEST)){
        //send ICMP echo reply
        icmp_send_packet(packet, len, ICMP_ECHOREPLY, 0);
        free(packet);
        return ;
    }

    //forward the packet

    //ttl-1
    iph->ttl --;
    if(iph->ttl <= 0){
        icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL);
        free(packet);
        return ;
    }
    //checksum
    iph->checksum = ip_checksum(iph);
    //lookup rtable
    rt_entry_t *match = longest_prefix_match(daddr);
    if(match == NULL){
        icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_NET_UNREACH);
        free(packet);
        return ;
    }
    //get next ip addr
    u32 next_ip;
    if(match->gw){
        next_ip = match->gw;
    }
    else{
        next_ip = daddr;
    }
    //forward
    iface_send_packet_by_arp(match->iface, next_ip, packet, len);
}
```

2. 路由表查询

在路由表查询过程中，遵循最长前缀匹配原则，具体代码如下：

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

考虑到路由表表项存储可能是无规律的，所以需要遍历整个路由表，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。注意 match==NULL 时也更新。这是为了保证第一次比对的时候默认更新，这样也可以方便处理默认路由（mask 为全 0）。

（二）ARP 数据包的处理

1. ARP 包分析

接收到的 ARP 报文分为 2 种，请求和回复。对于请求报文，当其请求的目的 IP 为该端口地址时，回复一个 ARP 回应报文，并且把源的 IP 和 MAC 映射关系加入 ARP 缓存。对于回应报文，当其回复的目的 IP 为该端口地址时，说明我们前面发出的一个 ARP 请求得到了回复，此时把源的 IP 和 MAC 映射关系加入 ARP 缓存。

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

    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);
        }
    }
    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 type");
    }

    free(packet);
}
```

2. ARP 包生成

ARP 包分为请求包和应答包，其直接通过端口发送，不涉及 ARP 查询等缓存相关操作。应答包和请求包的构造过程相似。由于 ARP 包的规格固定，首先可以申请固定长度的空间，填写以太网头部和 ARP 包的部分共通信息。由于请求包是广播，应答包是单播，所以二者的目的 mac 地址填写不同:在以太网头部中，前者为各位全 1，后者则是特定的目的 mac 地址;在 ARP 信息中，前者改为各位全 0，后者则为来源的 mac 地址。

```c
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, eth_broadcast_addr, 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);
    memcpy(arp->arp_tha, arp_request_addr, ETH_ALEN);
    arp->arp_tpa = htonl(dst_ip);

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

    //log(DEBUG, "handle arp send request packet\n");
}
```

```c
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;
    struct ether_arp *arp =  (struct ether_arp*)(packet + ETHER_HDR_SIZE);
    //ether header
    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);
    //arp
    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;
    //send
    iface_send_packet(iface, packet, ETHER_HDR_SIZE + sizeof(struct ether_arp));

}
```

考虑到路由表表项存储可能是无规律的，所以需要遍历整个路由表，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。注意 match==NULL 时也更新。这是为了保证第一次比对的时候默认更新，这样也可以方便处理默认路由（mask 为全 0）。

3. ARP 缓存机制

在 IP 数据包处理模块找到下一跳目的 IP 后，调用 iface_send_packet_by_arp 完成转发。这一模块先完成从目的 IP 到目的 MAC 地址的转换，如果在 arpcache 里查询到映射，完成转发;没有查到，把该数据包挂起到 ARP 等待队列。

```c
void iface_send_packet_by_arp(iface_info_t *iface, u32 dst_ip, char *packet, int len)
{
    struct ether_header *eh = (struct ether_header *)packet;
    memcpy(eh->ether_shost, iface->mac, ETH_ALEN);
    eh->ether_type = htons(ETH_P_IP);

    u8 dst_mac[ETH_ALEN];
    int found = arpcache_lookup(dst_ip, dst_mac);
    if (found) {
        // log(DEBUG, "found the mac of %x, send this packet", dst_ip);
        memcpy(eh->ether_dhost, dst_mac, ETH_ALEN);
        iface_send_packet(iface, packet, len);
    }
    else {
        // log(DEBUG, "lookup %x failed, pend this packet", dst_ip);
        arpcache_append_packet(iface, dst_ip, packet, len);
    }
}
```

（三）ARP 缓存的维护

1. ARP 缓存查找

ARP 缓存查找在数据包组建完成后、准备发送前进行。遍历缓存表中的所有有效映射条目，检查其中存储的 IP 项是否为需要查找的 IP，查找成功后填写其对应的下一跳 mac 地址。返回值表示查找情况，若查找到，返回 1，否则返回 0

```c
// 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].valid && arpcache.entries[i].ip4 == ip4){
            memcpy(mac, arpcache.entries[i].mac, ETH_ALEN);
            pthread_mutex_unlock(&arpcache.lock);
            return 1;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
    return 0;
}
```

2. 添加待应答数据包

新增等待 ARP 应答的数据包时，应申请 ARP 缓存的数据包条目空间，并填入该数据

包的指针和长度等信息。若已经向目的地址发出 ARP 请求，即存在该映射关系的等待队列，则直接将新条目加入队尾。

```
pthread_mutex_lock(&arpcache.lock);

struct arp_req *req_entry = NULL, *req_q;
list_for_each_entry_safe(req_entry, req_q, &(arpcache.req_list), list) {
    //find
    if (req_entry->ip4 == ip4) {
        struct cached_pkt *pkt = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
        init_list_head(&(pkt->list));
        pkt->packet = packet;
        pkt->len = len;
        list_add_tail(&pkt->list, &req_entry->cached_packets);

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

如果没有还没有发过请求，新建一个该 IP 对应的等待队列，然后向目的 IP 发一个 ARP 查询请求。

```
//not find
req_entry = (struct arp_req *)malloc(sizeof(struct arp_req));
init_list_head(&(req_entry->list));
req_entry->iface = iface;
req_entry->ip4 = ip4;
req_entry->sent = time(NULL);
req_entry->retries = 0;
init_list_head(&(req_entry->cached_packets));
list_add_tail(&req_entry->list, &(arpcache.req_list));

struct cached_pkt *pkt = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));
pkt->packet = packet;
pkt->len = len;
list_add_tail(&pkt->list, &req_entry->cached_packets);

pthread_mutex_unlock(&arpcache.lock);

arp_send_request(iface, ip4);
```

3. 添加 ARP 缓存映射项

接收到格式正确的 ARP 响应后需要将信息填入映射表中。值得注意的是，在把映射关系加入 ARP 缓存时，需要遍历整个缓存条目表，如果有 IP 相同的条目，更新其添加时间和 MAC 地址即可，不需要再访问 ARP 等待队列了。这是因为本身映射关系就在表

中，那么肯定没有等待该条映射的数据包。

而如果没有 IP 相同的条目，则优先在无效条目上覆盖填写，若条目已满则取随机数模上 MAX_ARP_SIZE 来替换条目。

```
pthread_mutex_lock(&arpcache.lock);
int i;
for(i = 0;i < MAX_ARP_SIZE;i++){
    // if the mapping of ip to mac already exist, update
    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;
    }
}

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

if(i == MAX_ARP_SIZE){
    i = rand() % MAX_ARP_SIZE;
}

arpcache.entries[i].valid = 1;
arpcache.entries[i].ip4 = ip4;
arpcache.entries[i].added = time(NULL);
memcpy(arpcache.entries[i].mac, mac, ETH_ALEN);
```

完成条目填写后，检查等待应答的数据包队列，查看有没有正在等待该条映射的数据包。如果有，把这些数据包依次填写目的 MAC 地址，转发出去，并删除掉相应缓存数据包。

```
// send 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){
        // there are pending packets
        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);
```

4. 检查清理 ARP 缓存

使用一个 sweep 线程来维护 ARP 缓存。每秒钟运行一次 arpcache_sweep 操作，遍历整个 ARP 等待队列和 ARP 缓存条目。如果一个缓存条目在缓存中已存在超过了 15 秒，将该条目清除。如果一个 IP 对应的 ARP 请求发出去已经超过了 1 秒，重新发送 ARP 请求。如果发送超过 5 次仍未收到 ARP 应答，则对该队列下的数据包依次回复 ICMP_HOST_UNREACHABE 消息，并删除等待的数据包。

注意，这个过程中涉及到对死锁的处理，这将在讲述完 ICMP 数据包的相关处理后进行讲述。

具体代码如下图所示：

```c
void *arpcache_sweep(void *arg)
{
    while (1) {
        sleep(1);
        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;
            }
        }

        //For the pending packets
        struct list_head temp_list;
        init_list_head(&temp_list);

        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;
                    //adding to temp_list to avoid deadlock
                    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);
                    }
                    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, list){
            icmp_send_packet(pkt_entry->packet, pkt_entry->len, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
            free(pkt_entry);
        }
    }

    return NULL;
}
```

（四）ICMP 数据包的处理

1. ICMP 包构建

考虑到 ICMP 包的许多内容需要重新填写，所以需要额外申请存储空间。对于非 ping_echo 的 ICMP 数据包，数据长度可以用以太网头+IP 头+ICMP 头+原数据包 IP 头+拷贝数据长度进行计算;对于 ping-echo 包这一计算可以简化为原数据包长度减去原来的 IP 头长度，再加上新 IP 头长度，这是因为 ping_echo 包与 ping_request 包结构基本一

致，只可能在 IP 头部分出现长度不一的情况。

开辟 IP 数据包空间后需填写 IP 和 ICMP 信息。IP 头填写过程中，对于 ping_echo 包，源 IP 地址为原数据包的目的地址，其它情况则查询路由表，找到下一跳网关并填写；目的 IP 地址则填写原数据包的源 IP 地址。ICMP 头部则根据传入数据，填写 ICMP 类型码，然后根据 ICMP 包的类型完成数据拷贝后，计算并填入校验和，即可准备发送。

具体代码如下所示：

```
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 *iph =  packet_to_ip_hdr(in_pkt);
    char* ipdata = IP_DATA(iph);

    //length
    int res_len = 0;
    int icmp_len = 0;
    if(type == ICMP_ECHOREPLY){
        icmp_len = ntohs(iph->tot_len) - IP_HDR_SIZE(iph);
    }
    else{
        icmp_len = ICMP_HDR_SIZE + IP_HDR_SIZE(iph) + ICMP_COPIED_DATA_LEN;
    }
    res_len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + icmp_len;

    //malloc
    char *res = (char *)malloc(res_len);
    memset(res, 0, res_len);
    // init iph
    struct iphdr *res_iph = packet_to_ip_hdr(res);
    if(type == ICMP_ECHOREPLY){
        ip_init_hdr(res_iph, ntohl(iph->daddr), ntohl(iph->saddr), IP_BASE_HDR_SIZE+icmp_len, IPPROTO_ICMP);
    }
    else{
        rt_entry_t *match = longest_prefix_match(ntohl(iph->saddr));
        if(match==NULL){
            free(res);
            return ;
        }
        ip_init_hdr(res_iph, match->iface->ip, ntohl(iph->saddr), IP_BASE_HDR_SIZE+icmp_len, IPPROTO_ICMP);
    }
    // init icmp
    char *res_ipdata = IP_DATA(res_iph);
    struct icmphdr *icmph = (struct icmphdr*)res_ipdata;
    if(type == ICMP_ECHOREPLY){
        memcpy(res_ipdata, ipdata, icmp_len);
    }
    else{
        memcpy(res_ipdata + ICMP_HDR_SIZE, iph, icmp_len-ICMP_HDR_SIZE);
    }

    icmph->type = type;
    icmph->code = code;
    icmph->checksum = icmp_checksum(icmph,icmp_len);
    //send
    ip_send_packet(res, res_len);
}
```

2. ICMP 包发出

发包前需要先根据先前填写的目的 IP 查找路由表，确定发送端口。选择下一跳 IP 的过程与直接处理 IP 包的过程类似。最后同样通过，iface_send_packet_by_arp 函数的 ARP 机制发出，如下所示：

```
void ip_send_packet(char *packet, int len)
{
    //fprintf(stderr, "TODO: send ip packet.\n");
    struct iphdr *iph =  packet_to_ip_hdr(packet);
    u32 daddr = ntohl(iph->daddr);
    //lookup rtable
    rt_entry_t *match = longest_prefix_match(daddr);
    if(match == NULL){
        free(packet);
        return ;
    }
    //get next ip addr

    u32 next_ip;
    if(match->gw){
        next_ip = match->gw;
    }
    else{
        next_ip = daddr;
    }
    //forward
    iface_send_packet_by_arp(match->iface, next_ip, packet, len);
}
```

（五）死锁处理

icmp_send_packet 函数中需要我们构建 ICMP 报文并发送。此时发送的目标 IP 地址即是发来这条出错数据包的源 IP 地址。然后端口根据目的 IP 查找路由表可得。那么 MAC 地址则会根据目的 IP 查找 ARP 缓存，但是这样就出现了死锁问题：ARP sweep 进程执行开始时会获取 ARP 缓存互斥锁，遇到重发 ARP 请求超过 5 次的数据包，需要发送 ICMP 不可达消息。如果发送 ICMP 时又去查找 ARP 表，就需要再次申请互斥锁，导致

死锁。

解决方法就是：将老化操作分成两个阶段：1、申请 ARP 缓存互斥锁，遍历链表，将所有需要老化的条目捡出来，放到一临时链表中，释放互斥锁；2、遍历临时链表，对其中的每个条目，发送 ICMP 消息，如下所示：

```
//For the pending packets
struct list_head temp_list;
init_list_head(&temp_list);

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;
            //adding to temp list to avoid deadlock
            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);
            }
            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, list){
    icmp_send_packet(pkt_entry->packet, pkt_entry->len, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
    free(pkt_entry);
}
```

五、 实验结果与分析

（一）给定拓扑下的 ping 测试

执行 router_topo.py 脚本，在 r1 结点上启动 router 程序，在 h1 结点上对分别对 10.0.1.1 (路由器结点)和 10.0.2.22 和 10.0.3.1 和 10.0.4.44 使用 ping 指令，得到的结果如下图所示：

```
root@Computer:~/workspace/Network/lab7/07-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=1.00 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.153 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.053 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.070 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3170ms
rtt min/avg/max/mdev = 0.053/0.320/1.005/0.397 ms
```

```
root@Computer:~/workspace/Network/lab7/07-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=1.21 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.060 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.083 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.064 ms

--- 10.0.2.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3009ms
rtt min/avg/max/mdev = 0.060/0.355/1.215/0.496 ms
```

```
root@Computer:~/workspace/Network/lab7/07-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=0.992 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.086 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.125 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.072 ms

--- 10.0.3.33 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3010ms
rtt min/avg/max/mdev = 0.072/0.318/0.992/0.389 ms
```

```
root@Computer:~/workspace/Network/lab7/07-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

--- 10.0.3.11 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3018ms
pipe 4
```

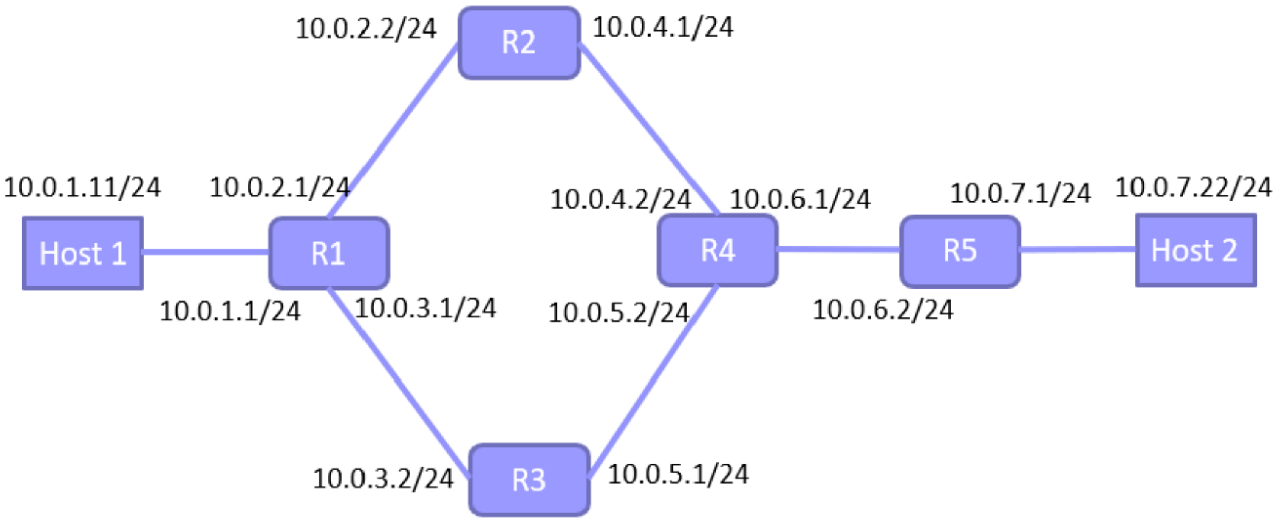
```
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.4.1 -c 4
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
From 10.0.1.1 icmp_seq=1 Destination Net Unreachable
From 10.0.1.1 icmp_seq=2 Destination Net Unreachable
From 10.0.1.1 icmp_seq=3 Destination Net Unreachable
From 10.0.1.1 icmp_seq=4 Destination Net Unreachable

--- 10.0.4.1 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3036ms
```

可见 ICMP 包生成转发正确。

（二）多路由器节点的复杂拓扑下测试

构造一个多路由器节点的复杂拓扑结构，如下图所示：



对应代码如下所示：

```
h1.cmd('ifconfig h1-eth0 10.0.1.11/24')
h2.cmd('ifconfig h2-eth0 10.0.7.22/24')

h1.cmd('route add default gw 10.0.1.1')
h2.cmd('route add default gw 10.0.7.1')

r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
r1.cmd('ifconfig r1-eth1 10.0.2.1/24')
r1.cmd('ifconfig r1-eth2 10.0.3.1/24')

r2.cmd('ifconfig r2-eth0 10.0.2.2/24')
r2.cmd('ifconfig r2-eth1 10.0.4.1/24')

r3.cmd('ifconfig r3-eth0 10.0.3.2/24')
r3.cmd('ifconfig r3-eth1 10.0.5.1/24')

r4.cmd('ifconfig r4-eth0 10.0.4.2/24')
r4.cmd('ifconfig r4-eth1 10.0.5.2/24')
r4.cmd('ifconfig r4-eth2 10.0.6.1/24')

r5.cmd('ifconfig r5-eth0 10.0.6.2/24')
r5.cmd('ifconfig r5-eth1 10.0.7.1/24')

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 10.0.3.2 dev r1-eth2')
r1.cmd('route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')
r1.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.2.2 dev r1-eth1')

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 10.0.3.0 netmask 255.255.255.0 gw 10.0.2.1 dev r2-eth0')
r2.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.4.2 dev r2-eth1')
r2.cmd('route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.4.2 dev r2-eth1')
r2.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.4.2 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')
r3.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.5.2 dev r3-eth1')
r3.cmd('route add -net 10.0.6.0 netmask 255.255.255.0 gw 10.0.5.2 dev r3-eth1')
r3.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.5.2 dev r3-eth1')

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 10.0.5.1 dev r4-eth1')
r4.cmd('route add -net 10.0.7.0 netmask 255.255.255.0 gw 10.0.6.2 dev r4-eth2')

r5.cmd('route add -net 10.0.1.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.2.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
r5.cmd('route add -net 10.0.5.0 netmask 255.255.255.0 gw 10.0.6.1 dev r5-eth0')
```

h1 节点 ping 每个路由器节点的入端口 IP 地址，结果如下：

```
root@Computer:~/workspace/Network/lab7/07-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=0.457 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.066 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.059 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.063 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3086ms
rtt min/avg/max/mdev = 0.059/0.161/0.457/0.171 ms
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.2.2 -c 4
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=0.473 ms
64 bytes from 10.0.2.2: icmp_seq=2 ttl=63 time=0.154 ms
64 bytes from 10.0.2.2: icmp_seq=3 ttl=63 time=0.169 ms
64 bytes from 10.0.2.2: icmp_seq=4 ttl=63 time=0.141 ms

--- 10.0.2.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3018ms
rtt min/avg/max/mdev = 0.141/0.234/0.473/0.138 ms
```

```
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.3.2 -c 4
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=63 time=0.376 ms
64 bytes from 10.0.3.2: icmp_seq=2 ttl=63 time=0.145 ms
64 bytes from 10.0.3.2: icmp_seq=3 ttl=63 time=0.161 ms
64 bytes from 10.0.3.2: icmp_seq=4 ttl=63 time=0.131 ms

--- 10.0.3.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3112ms
rtt min/avg/max/mdev = 0.131/0.203/0.376/0.100 ms
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.4.2 -c 4
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=62 time=0.371 ms
64 bytes from 10.0.4.2: icmp_seq=2 ttl=62 time=0.162 ms
64 bytes from 10.0.4.2: icmp_seq=3 ttl=62 time=0.208 ms
64 bytes from 10.0.4.2: icmp_seq=4 ttl=62 time=0.234 ms

--- 10.0.4.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3007ms
rtt min/avg/max/mdev = 0.162/0.243/0.371/0.080 ms
```

```
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.5.2 -c 4
PING 10.0.5.2 (10.0.5.2) 56(84) bytes of data.
64 bytes from 10.0.5.2: icmp_seq=1 ttl=62 time=0.465 ms
64 bytes from 10.0.5.2: icmp_seq=2 ttl=62 time=0.141 ms
64 bytes from 10.0.5.2: icmp_seq=3 ttl=62 time=0.178 ms
64 bytes from 10.0.5.2: icmp_seq=4 ttl=62 time=0.155 ms

--- 10.0.5.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3013ms
rtt min/avg/max/mdev = 0.141/0.234/0.465/0.134 ms
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.6.2 -c 4
PING 10.0.6.2 (10.0.6.2) 56(84) bytes of data.
64 bytes from 10.0.6.2: icmp_seq=1 ttl=61 time=0.397 ms
64 bytes from 10.0.6.2: icmp_seq=2 ttl=61 time=0.149 ms
64 bytes from 10.0.6.2: icmp_seq=3 ttl=61 time=0.171 ms
64 bytes from 10.0.6.2: icmp_seq=4 ttl=61 time=1.74 ms

--- 10.0.6.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3211ms
rtt min/avg/max/mdev = 0.149/0.615/1.746/0.660 ms
```

```
root@Computer:~/workspace/Network/lab7/07-router# ping 10.0.7.22 -c 4
PING 10.0.7.22 (10.0.7.22) 56(84) bytes of data.
64 bytes from 10.0.7.22: icmp_seq=1 ttl=60 time=0.429 ms
64 bytes from 10.0.7.22: icmp_seq=2 ttl=60 time=0.149 ms
64 bytes from 10.0.7.22: icmp_seq=3 ttl=60 time=0.160 ms
64 bytes from 10.0.7.22: icmp_seq=4 ttl=60 time=0.187 ms

--- 10.0.7.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3050ms
rtt min/avg/max/mdev = 0.149/0.231/0.429/0.115 ms
```

可见各 ICMP 包生成转发正确。

在一个 h1 节点上 traceroute h2,能够正确输出路径上每个节点的 IP 信息,结果如下：

```
root@Computer:~/workspace/Network/lab7/07-router# traceroute 10.0.7.22
traceroute to 10.0.7.22 (10.0.7.22), 64 hops max
  1   10.0.1.1  0.093ms  0.103ms  0.119ms
  2   10.0.2.2  0.153ms  0.126ms  0.053ms
  3   10.0.4.2  0.123ms  0.120ms  0.055ms
  4   10.0.6.2  0.141ms  0.053ms  0.050ms
  5   10.0.7.22  0.069ms  0.054ms  0.055ms
```

六、 实验总结

通过本次实验，我了解了路由器转发的原理、路由表结构，并且实现了最长前缀匹配查找方法以及路由器的查询和转发功能，理解了路由器转发数据包的流程。本次实验中，我还学到了如何构建 ARP 缓存表、如何实现 ARP 缓存机制、以及如何处理 ARP 请求和应答，这让我对 ARP 协议有了更多的理解。除此之外，我还了解了 ICMP 协议，它在 IP 报文转发中起到了重要的报错、查询功能。