路由器转发实验报告

张翔雨 2018K8009929035

一、实验题目：路由器转发实验

二、实验内容

- 基于已有代码，实现路由器转发机制，对于给定拓扑 `(router_topo.py`)，在r1上执行路由器程序, 进行数据包的处理。
- 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
- 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
  - 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
  - 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

三、实验过程

1.完成`arp.c`,实现ARP包的回复和请求，以及对ARP包的处理。

（1）实现`arp_send_request(iface_info_t *iface, u32 dst_ip)`

函数功能：发送arp请求

函数流程：见代码及注释

```
// send an arp request: encapsulate an arp request packet, send it out through
// iface_send_packet
```

```c
void arp_send_request(iface_info_t *iface, u32 dst_ip)
{
    /*生成要发送的包，并定位各部分内容*/
    char *packet = (char *)malloc(sizeof(struct ether_header) + sizeof(struct
ether_arp));
    struct ether_header *header = (struct ether_header *)packet;
    struct ether_arp *arp = (struct ether_arp *)(packet + sizeof(struct ether_header));
    /*填充etherhead的内容*/
    header->ether_type = htons(ETH_P_ARP);
    memcpy(header->ether_shost,iface->mac, ETH_ALEN);
    memset(header->ether_dhost,0xff, ETH_ALEN);//广播包
    /*填充arp协议的内容*/
    arp->arp_hrd = htons(ARPHRD_ETHER);
    arp->arp_pro = htons(0x0800);
    arp->arp_hln = 6;
    arp->arp_pln = 4;
    arp->arp_op =  htons(ARPOP_REQUEST);//代表类型为arp请求
    memcpy(arp->arp_sha,iface->mac,ETH_ALEN);
    memset(arp->arp_tha,0, ETH_ALEN);//当为ARP请求时，Target HW Addr置空
    arp->arp_spa = htonl(iface->ip);
    arp->arp_tpa = htonl(dst_ip);
    /*发送包*/
    iface_send_packet(iface,packet,sizeof(struct ether_header) + sizeof(struct
ether_arp));
}
```

（2）实现arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)

函数功能：发送arp回复

函数流程：见代码及注释

```c
// 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)
{
    /*生成要发送的包，并定位各部分内容*/
    char *packet = (char *)malloc(sizeof(struct ether_header) + sizeof(struct
ether_arp));
    struct ether_header *header = (struct ether_header *)packet;
    struct ether_arp *arp = (struct ether_arp *)(packet + sizeof(struct ether_header));
    /*填充etherhead的内容*/
    header->ether_type = htons(ETH_P_ARP);
    memcpy(header->ether_shost,iface->mac, ETH_ALEN);
    memcpy(header->ether_dhost,req_hdr->arp_sha, ETH_ALEN);
    /*填充arp协议的内容*/
    arp->arp_hrd = htons(ARPHRD_ETHER);
    arp->arp_pro = htons(0x0800);
    arp->arp_hln = 6;
```

```
    arp->arp_pln = 4;
    arp->arp_op =  htons(ARPOP_REPLY);//代表类型为arp回复
    memcpy(arp->arp_sha,iface->mac,ETH_ALEN);
    memcpy(arp->arp_tha,req_hdr->arp_sha, ETH_ALEN);
    arp->arp_spa = htonl(iface->ip);
    arp->arp_tpa = req_hdr->arp_spa;
    /*发送包*/
    iface_send_packet(iface,packet,sizeof(struct ether_header) + sizeof(struct
ether_arp));
}
```

（3）实现handle_arp_packet(iface_info_t *iface, char *packet, int len)

函数功能：根据收到的arp包的op部分内容，执行相应操作。

```
void handle_arp_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_arp *arp = (struct ether_arp *)(packet + sizeof(struct ether_header));

    switch(ntohs(arp->arp_op))
    {
        case ARPOP_REQUEST:
            if(ntohl(arp->arp_tpa) == iface->ip)
                arp_send_reply(iface, arp);
            arpcache_insert(htonl(arp->arp_spa),arp->arp_sha);
            break;
        case ARPOP_REPLY:
            arpcache_insert(htonl(arp->arp_spa),arp->arp_sha);
            break;
        default:
            free(packet);
            break;
    }
}
```

2.完成arpcache.c，实现ARP缓存相关操作

（1）实现int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])

函数功能：查询ARP缓存，若查找到返回1，若无返回0

函数流程：见代码和注释

```
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    pthread_mutex_lock(&arpcache.lock);//获取互斥锁，保证对缓存的操作是互斥的
```

```c
    for(int i = 0; i < MAX_ARP_SIZE; i++)
    {
        if(arpcache.entries[i].ip4 == ip4 && arpcache.entries[i].valid)//找到对应表项
        {
            memcpy(mac,arpcache.entries[i].mac,ETH_ALEN);//将找到的mac地址存入，释放互斥锁并返回
            pthread_mutex_unlock(&arpcache.lock);
            return 1;
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
    return 0;
}
```

(2)实现`void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len)`

函数功能：将要发送的数据包添加到待发送数据包队列中

```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)
{
    pthread_mutex_lock(&arpcache.lock);
    int uncached = 1;//标志请求的ip是否已经在缓存中
    struct arp_req *entry,*q;
    struct cached_pkt *pkt = (struct cached_pkt *)malloc(sizeof(struct cached_pkt));//生成待发送的数据包
    pkt->len = len;
    pkt->packet = packet;
    init_list_head(&pkt->list);
    list_for_each_entry_safe(entry, q, &arpcache.req_list,list)
    {
        if(entry->ip4 == ip4)//在缓存中找到了请求的ip
        {
            list_add_tail(&pkt->list,&(entry->cached_packets));
            uncached = 0;
            break;
        }
    }
    if(uncached == 1)//如果不在缓存中，则新生成一个req项添加到队尾，并将数据包加入，进入等待发送状态
    {
        struct arp_req *new_req = (struct arp_req *)malloc(sizeof(struct arp_req));
        init_list_head(&new_req->list);
        new_req->iface = iface;
```

```
        new_req->ip4 = ip4;
        new_req->sent = time(NULL);
        new_req->retries = 0;
        init_list_head(&new_req->cached_packets);
        list_add_tail(&new_req->list,&arpcache.req_list);
        list_add_tail(&pkt->list,&new_req->cached_packets);
        arp_send_request(iface,ip4);

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

（3）实现void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])

函数功能：将新的对应关系插入缓存，并检查是否存在等待该关系的数据包

```
// 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])
{
    pthread_mutex_lock(&arpcache.lock);
    int i = 0;
    for (i = 0; i < MAX_ARP_SIZE; i++)
    {
        if(!arpcache.entries[i].valid)//找到空闲的表项，插入
        {
            arpcache.entries[i].ip4 = ip4;
            memcpy(arpcache.entries[i].mac,mac, ETH_ALEN);
            arpcache.entries[i].added = time(NULL);
            arpcache.entries[i].valid = 1;
            break;
        }
    }
    if(i == MAX_ARP_SIZE)//没有空闲表项，随机替换一个
    {
        srand(time(NULL));
        int index = rand() % 32;
        arpcache.entries[index].ip4 =  ip4;
        memcpy(arpcache.entries[index].mac,mac,ETH_ALEN);
        arpcache.entries[index].added = time(NULL);
        arpcache.entries[index].valid = 1;
    }
    struct arp_req *entry,*q;
    list_for_each_entry_safe(entry, q, &arpcache.req_list,list)//查询是否有等待该ip地址的数据包，如果存在，将数据包发送出去
    {
        if(entry->ip4 == ip4)
        {
            struct cached_pkt *pkt_entry,*pkt;
```

```
            list_for_each_entry_safe(pkt_entry,pkt,&entry->cached_packets,list)
            {
                struct ether_header *eh = (struct ether_header *)(pkt_entry->packet);
                memcpy(eh->ether_shost, entry->iface->mac, ETH_ALEN);
                memcpy(eh->ether_dhost,mac,ETH_ALEN);
                eh->ether_type = htons(ETH_P_IP);
                iface_send_packet(entry->iface,pkt_entry->packet,pkt_entry->len);
                list_delete_entry(&pkt_entry->list);
                free(pkt_entry);
            }
            list_delete_entry(&entry->list);
            free(entry);
        }
    }
    pthread_mutex_unlock(&arpcache.lock);
}
```

（4）实现 `void *arpcache_sweep(void *arg)`

函数功能：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（Destination Host Unreachable）消息，并删除等待的数据包。

```
void *arpcache_sweep(void *arg)
{
    while (1) {
        sleep(1);
        /*生成一个用于存放需要回复消息的包的列表，在访问缓存结束后进行回复，避免对缓存的访问产生
死锁*/
        struct cached_pkt *tmp_list = (struct cached_pkt *)malloc(sizeof(struct
cached_pkt));
        init_list_head(&tmp_list->list);

        pthread_mutex_lock(&arpcache.lock);//获取互斥锁
        /*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*/
        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 arp_req *entry,*q;
        list_for_each_entry_safe(entry, q, &arpcache.req_list,list)
        {
            /*如果一个IP对应的ARP请求发出去已经超过了1秒且请求次数不大于5次，重新发送ARP请求*/
            if( (time(NULL)- entry->sent) > 1 && entry->retries <= 5)
            {
                entry->sent = time(NULL);
```

```
                entry->retries ++;
                arp_send_request(entry->iface,entry->ip4);
            }
            /*如果请求次数超过了5次*/
            else if(entry->retries > ARP_REQUEST_MAX_RETRIES)
            {
                struct cached_pkt *pkt_entry,*pkt;
                list_for_each_entry_safe(pkt_entry,pkt,&entry->cached_packets,list)
                {
                    /*生成一个需要回复数据包的复制，放入暂存列表中*/
                    struct cached_pkt *tmp = (struct cached_pkt *)malloc(sizeof(struct
cached_pkt));
                    init_list_head(&tmp->list);
                    tmp->len = pkt_entry->len;
                    tmp->packet = pkt_entry->packet;
                    list_add_tail(&tmp->list,&tmp_list->list);
                    list_delete_entry(&pkt_entry->list);//删除掉缓存中的表项
                    free(pkt_entry);
                }
                list_delete_entry(&entry->list);
                free(entry);
            }
        }
        pthread_mutex_unlock(&arpcache.lock);//放互斥锁
        /*遍历临时链表，对其中的每个条目，发送ICMP消息*/
        struct cached_pkt *pkt_entry,*pkt;
        list_for_each_entry_safe(pkt_entry,pkt,&tmp_list->list,list)
        {
            icmp_send_packet(pkt_entry->packet,pkt_entry-
>len,ICMP_DEST_UNREACH,ICMP_HOST_UNREACH);
            list_delete_entry(&pkt_entry->list);
            free(pkt_entry);
        }
    }
    return NULL;
}
```

3.完成ip_base.c，包含最长前缀查找和ip数据包发送

（1）实现void ip_init_hdr(struct iphdr *ip, u32 saddr, u32 daddr, u16 len, u8 proto)

函数功能：查找得到最长前缀对应的路由表项

```
rt_entry_t *longest_prefix_match(u32 dst)
{
    rt_entry_t *entry;
    rt_entry_t *max = NULL;//最长前缀的表项
    u32 max_mask = 0;//最长前缀值
```

```
    list_for_each_entry(entry,&rtable,list)
    {
        if(((entry->dest&entry->mask) == (dst & entry->mask)) && (entry->mask >
max_mask))//找到最长前缀
        {
            max_mask = entry->mask;
            max = entry;
        }
    }
    return max;
}
```

（2）实现void ip_send_packet(char *packet, int len)

函数功能：在发送ICMP数据包时进行的ip数据包发送。

```
void ip_send_packet(char *packet, int len)
{
    struct iphdr *header = packet_to_ip_hdr(packet);
    rt_entry_t *entry = longest_prefix_match(ntohl(header->daddr));//查找目的ip对应的表项
    if(!entry)
    {
        free(packet);
        return;
    }
    //路由器端口与目的地址不在同一网段:entry->gw ;路由器端口与目的地址在同一网段:dst_ip
    u32 dst = entry->gw ? entry->gw : ntohl(header->daddr);
    struct ether_header *eh = (struct ether_header *)(packet);
    memcpy(eh->ether_shost, entry->iface->mac, ETH_ALEN);
    eh->ether_type = htons(ETH_P_IP);
    iface_send_packet_by_arp(entry->iface,dst,packet, len);//通过arp协议发送
}
```

4.完成ip.c,实现处理ip数据包的操作

实现void handle_ip_packet(iface_info_t *iface, char *packet, int len)

函数功能：如果收到的包目的是本路由器端口 ,并且 ICMP 首部 type为 请求 ,回应 ICMP 报文,否则转发。

```
void handle_ip_packet(iface_info_t *iface, char *packet, int len)
{
    /*定位待处理数据包的各部分*/
    struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
    struct icmphdr *icmp_hdr = (struct icmphdr *)IP_DATA(ip_hdr);
    struct ether_header *eh = (struct ether_header *)(packet);
```

```
    u32 dst = htonl(ip_hdr->daddr);
    memcpy(eh->ether_shost,iface->mac, ETH_ALEN);

    /*如果收到的包目的是本路由器端口,并且 ICMP 首部 type为 请求 ,回应 ICMP 报文*/
    if(icmp_hdr->type == ICMP_ECHOREQUEST && iface->ip == dst)//
    {
        icmp_send_packet(packet,len,ICMP_ECHOREPLY,ICMP_NET_UNREACH);
    }
    else//转发报文
    {
        /*对IP头部的TTL值进行减一操作，如果该值 <= 0，则将该数据包丢弃，并回复ICMP信息*/
        ip_hdr->ttl --;
        if(ip_hdr->ttl <= 0)
        {
            icmp_send_packet(packet,len,ICMP_TIME_EXCEEDED,ICMP_NET_UNREACH);
            free(packet);
            return;
        }
        /*IP头部数据已经发生变化，需要重新设置checksum*/
        ip_hdr->checksum = ip_checksum(ip_hdr);
        /*发送新的数据包*/
        rt_entry_t *entry = longest_prefix_match(dst);
        if(entry)
        {
            u32 dest = entry->gw ? entry->gw : dst;
            iface_send_packet_by_arp(entry->iface,dest,packet, len);
        }
        else
        {
            icmp_send_packet(packet,len,ICMP_DEST_UNREACH,ICMP_NET_UNREACH);
        }
    }
}
```

5.完成icmp.c，实现icmp数据包的发送

实现void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)

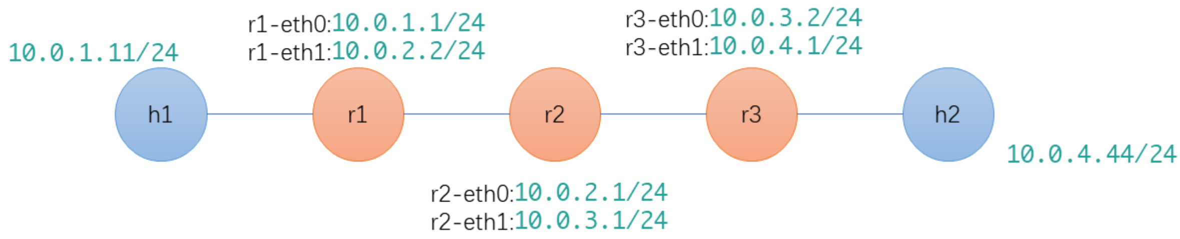
函数功能：完成icmp报文发送。

```
// 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");
    /*定位待处理的数据包的各部分*/
    struct ether_header *in_eh = (struct ether_header*)(in_pkt);
```

```
    struct iphdr *in_iph = packet_to_ip_hdr(in_pkt);

    int packet_len;//待发送数据包的长度

    if (type == ICMP_ECHOREPLY && code == ICMP_NET_UNREACH)
    {
        packet_len = len;//与原数据包相同
    }
    else
    {
        packet_len = ETHER_HDR_SIZE+IP_BASE_HDR_SIZE+ICMP_HDR_SIZE+IP_HDR_SIZE(in_iph) + 8;//需要拷贝收到数据包的IP头部 (>= 20字节) 和随后的8字节
    }
    /*生成待发送的数据包*/
    char *packet = (char *)malloc(packet_len);
    struct ether_header *eh = (struct ether_header *)(packet);
    struct iphdr *iph = packet_to_ip_hdr(packet);
    struct icmphdr *icmph = (struct icmphdr *)(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);

    eh->ether_type = htons(ETH_P_IP);
    memcpy(eh->ether_dhost, in_eh->ether_dhost, ETH_ALEN);
    memcpy(eh->ether_shost, in_eh->ether_dhost, ETH_ALEN);

    u32 saddr = ntohl(in_iph->saddr);
    rt_entry_t *entry = longest_prefix_match(saddr);
    ip_init_hdr(iph, entry->iface->ip,saddr, packet_len-ETHER_HDR_SIZE, 1);

    icmph->type = type;
    icmph->code = code;

    char *rest_1 = (char *)((char *)in_iph + IP_HDR_SIZE(in_iph) + ICMP_HDR_SIZE - 4);//待答复数据包的剩余部分
    char *rest_2 = (char *)((char *)icmph + ICMP_HDR_SIZE - 4);//新数据包的剩余部分


    if (type == ICMP_ECHOREPLY && code == ICMP_NET_UNREACH)
    {
        memcpy(rest_2, rest_1, len - ETHER_HDR_SIZE - IP_HDR_SIZE(in_iph) - ICMP_HDR_SIZE + 4);
        icmph->checksum = icmp_checksum(icmph, packet_len - ETHER_HDR_SIZE - IP_HDR_SIZE(in_iph));//重新计算checksum值
    }
    else
    {
        memset(rest_2, 0, 4);//前4字节设置为0
        memcpy(rest_2 + 4, in_iph, IP_HDR_SIZE(in_iph) + 8);//接着拷贝收到数据包的IP头部 (>= 20字节) 和随后的8字节
        icmph->checksum = icmp_checksum(icmph, IP_HDR_SIZE(in_iph) + 8 + ICMP_HDR_SIZE);//重新计算checksum值
    }
    ip_send_packet(packet, packet_len);
```

```
}
```

6.构造新的路由器拓扑，在新的拓扑上进行连通性验证和路径测试。



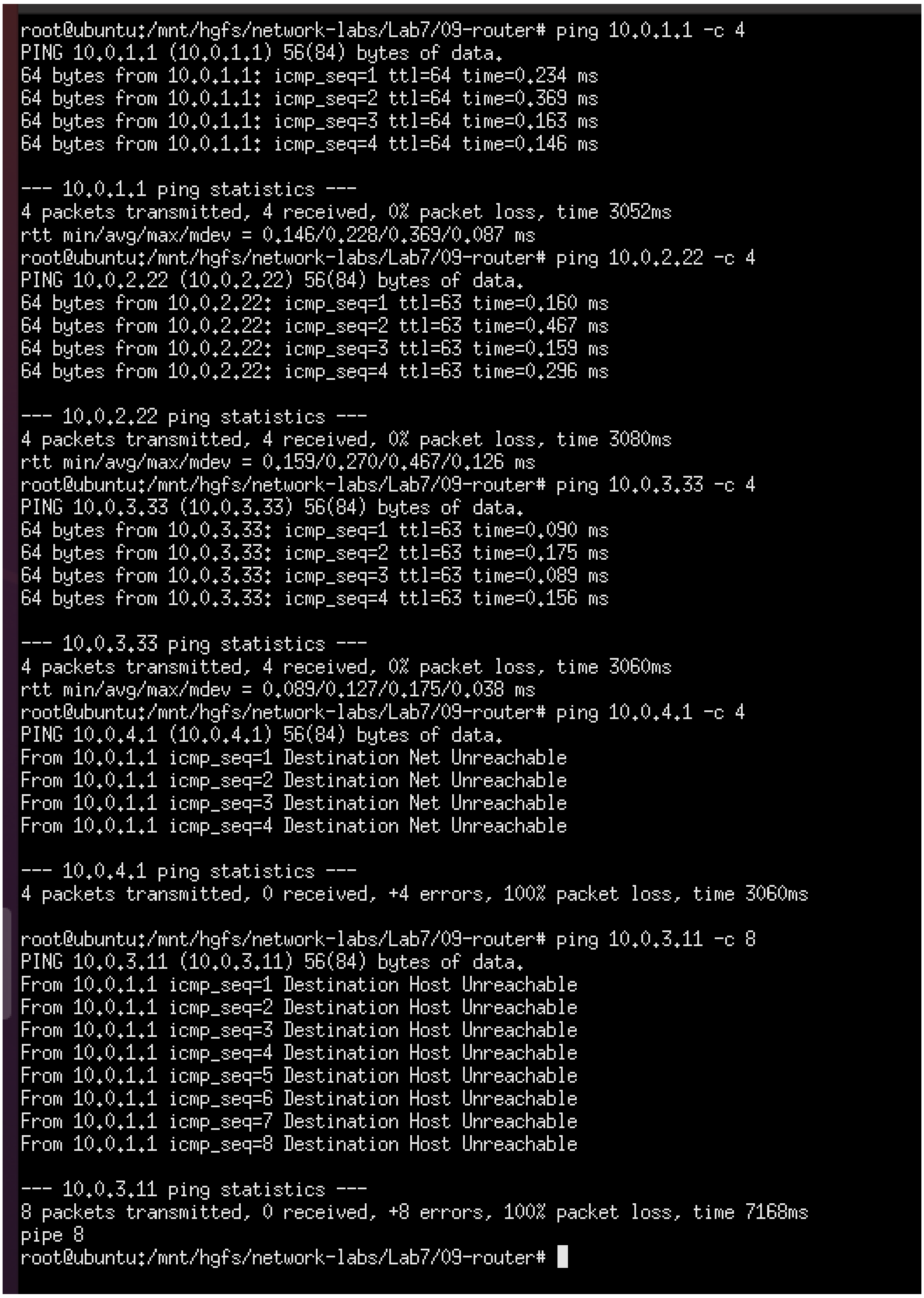
构造如上图所示的3路由器2节点拓扑，进行连通性验证和路径测试，节点连接代码如下：

```
    h1.cmd('ifconfig h1-eth0 10.0.1.11/24')
    h2.cmd('ifconfig h2-eth0 10.0.4.44/24')
    h1.cmd('route add default gw 10.0.1.1')
    h2.cmd('route add default gw 10.0.4.1')

    r1.cmd('ifconfig r1-eth0 10.0.1.1/24')
    r1.cmd('ifconfig r1-eth1 10.0.2.2/24')
    r1.cmd('route add -net 10.0.4.0 netmask 255.255.255.0 gw 10.0.2.1 dev r1-eth1')
    r1.cmd('route add -net 10.0.3.0 netmask 255.255.255.0 gw 10.0.2.1 dev r1-eth1')


    r2.cmd('ifconfig r2-eth0 10.0.2.1/24')
    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 10.0.2.2 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('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')
```

四、实验结果

1.router_topo.py测试结果

运行拓扑脚本，进行连通性验证，得到结果如下：



```
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-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=0.234 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.369 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.163 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.146 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3052ms
rtt min/avg/max/mdev = 0.146/0.228/0.369/0.087 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-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=0.160 ms
64 bytes from 10.0.2.22: icmp_seq=2 ttl=63 time=0.467 ms
64 bytes from 10.0.2.22: icmp_seq=3 ttl=63 time=0.159 ms
64 bytes from 10.0.2.22: icmp_seq=4 ttl=63 time=0.296 ms

--- 10.0.2.22 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3080ms
rtt min/avg/max/mdev = 0.159/0.270/0.467/0.126 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-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=0.090 ms
64 bytes from 10.0.3.33: icmp_seq=2 ttl=63 time=0.175 ms
64 bytes from 10.0.3.33: icmp_seq=3 ttl=63 time=0.089 ms
64 bytes from 10.0.3.33: icmp_seq=4 ttl=63 time=0.156 ms

--- 10.0.3.33 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3060ms
rtt min/avg/max/mdev = 0.089/0.127/0.175/0.038 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-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 3060ms

root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-router# ping 10.0.3.11 -c 8
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
From 10.0.1.1 icmp_seq=8 Destination Host Unreachable

--- 10.0.3.11 ping statistics ---
8 packets transmitted, 0 received, +8 errors, 100% packet loss, time 7168ms
pipe 8
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-router#
```

结果分析如下：

- `Ping 10.0.1.1 (r1)` ： ping路由器入端口ip，能够ping通
- `Ping 10.0.2.22 (h2)` 或 `Ping 10.0.3.33 (h3)` ： ping能够连接到的节点，能够ping通
- `Ping 10.0.3.11` : ping不存在的节点，返回 `ICMP Destination Host Unreachable`
- `Ping 10.0.4.1` ： ping不存在的网段，返回 `ICMP Destination Net Unreachable`

与理论结果相同，验证成功。

2.three_router_topo.py测试结果

```
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-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=0.207 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=0.128 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=0.092 ms
64 bytes from 10.0.1.1: icmp_seq=4 ttl=64 time=0.374 ms

--- 10.0.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3056ms
rtt min/avg/max/mdev = 0.092/0.200/0.374/0.108 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-router# ping 10.0.2.1 -c 4
PING 10.0.2.1 (10.0.2.1) 56(84) bytes of data.
64 bytes from 10.0.2.1: icmp_seq=1 ttl=63 time=0.692 ms
64 bytes from 10.0.2.1: icmp_seq=2 ttl=63 time=0.481 ms
64 bytes from 10.0.2.1: icmp_seq=3 ttl=63 time=0.747 ms
64 bytes from 10.0.2.1: icmp_seq=4 ttl=63 time=0.458 ms

--- 10.0.2.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3041ms
rtt min/avg/max/mdev = 0.458/0.594/0.747/0.126 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-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=62 time=0.572 ms
64 bytes from 10.0.3.2: icmp_seq=2 ttl=62 time=0.925 ms
64 bytes from 10.0.3.2: icmp_seq=3 ttl=62 time=0.613 ms
64 bytes from 10.0.3.2: icmp_seq=4 ttl=62 time=0.824 ms

--- 10.0.3.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3018ms
rtt min/avg/max/mdev = 0.572/0.733/0.925/0.146 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-router# traceroute 10.0.4.44 -m 10
traceroute to 10.0.4.44 (10.0.4.44), 10 hops max, 60 byte packets
 1  10.0.1.1 (10.0.1.1)  0.281 ms  0.251 ms  0.246 ms
 2  10.0.2.1 (10.0.2.1)  0.410 ms  0.409 ms  0.406 ms
 3  10.0.3.2 (10.0.3.2)  0.625 ms  0.630 ms  0.628 ms
 4  10.0.4.44 (10.0.4.44)  0.626 ms  0.622 ms  0.620 ms
root@ubuntu:/mnt/hgfs/network-labs/Lab7/09-router#
```

结果分析：

- 终端节点 `ping` 每个路由器节点的入端口IP地址：能够ping通
- 在 `h1` 上 `traceroute h2` ，正确输出路径上每个节点的IP信息

与预期结果相同，证明连通性良好，路径正确。

五、实验总结

本次实验相较于之前的实验有了很明显的难度提升，代码量增加的非常多，其中运用到了多种协议转发模式，刚开始写的时候比较容易迷惑，通过写代码并进行测试，逐步输出函数之间的调用关系，加快了我对实验内容的理解，并最终完成了本次实验。