中国科学院大学
University of Chinese Academy of Sciences

计算机网络实验作业

IP 路由转发实验

小组成员

陈秀林	2017E8000961103	计算机技术
孙志浩	2017E8020261116	计算机技术
刘新江	2017E8020261052	电子与通信工程

2017 年 12 月

目录

1 实验目的 2

2 实验设备 3

3 实验内容 4

3.1 数据结构 4

3.2 实现逻辑 10

4 实验步骤 22

5 实验结果与分析 23

1 实验目的

本次大作业，我们选择 IP 路由转发实验。首先在理论层面上，掌握路由转发的基本原理，随后，在给定网络拓扑以及节点的路由表配置的基础上，实现路由器的转发功能，使得各节点之间能够相互传送数据，网络拓扑图如图 1 所示：

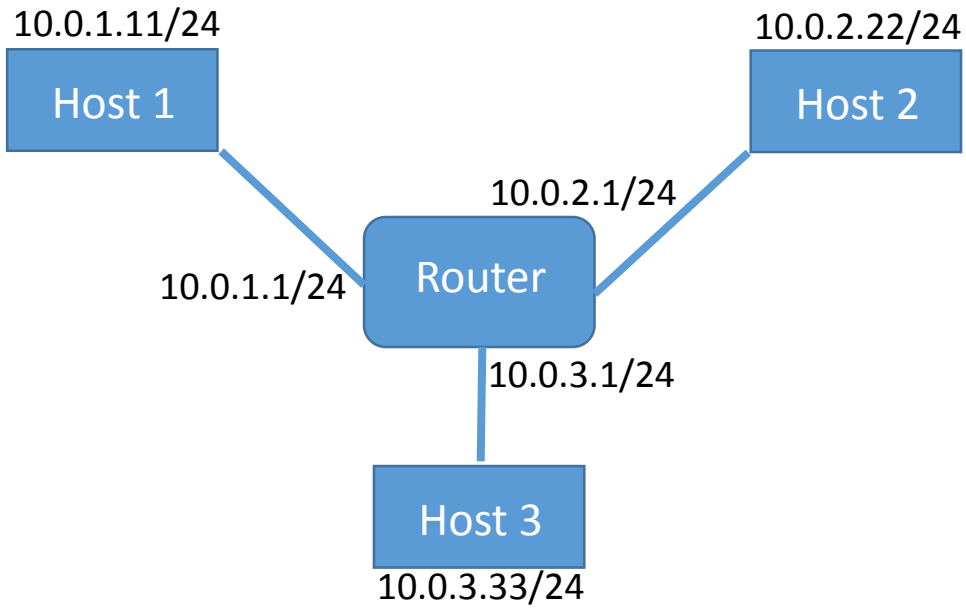

图 1:：路由实验网络拓扑

由图 1 可以看出，本次实验的网络拓扑包括一个路由器和 3 台主机，每个主机和路由器建立物理连接且分别处在各自特定的网段。路由表格式如下所示：

```
1.  /*路由表格式*/
2.  typedef struct {
3.      struct list_head list;      //链表实现
4.      u32 dest;                   // 目的网络地址
5.      u32 gw;                     // 下一跳网关地址
6.      u32 mask;                   // 网络掩码
7.      int flags;                  // 转发表条目标识（可忽略）
8.      char if_name[16];           // 转出端口名字（eg.r1-eth0）
9.      iface_info_t *iface;        // 转出端口（指针）
10. } rt_entry_t;
11.
12. extern struct list_head rtable; //路由表
```

2 实验设备

在 Linux 环境下使用 Mininet 工具，Mininet 是由一些虚拟的终端节点、交换机、路由器连接而成的一个网络仿真器，可以简单、迅速地创建一个支持用户自定义的网络拓扑，如图 1 所示那样，终端节点就是 Host 主机，Host 就像真实的电脑一样工作，可以使用 ssh 登录，启动应用程序，程序可以向以太网端口发送数据包，数据包会被交换机、路由器接收并处理。有了这个网络，就可以灵活地为网络添加新的功能并进行相关测试，缩短开发测试周期。

3 实验内容

IP 路由转发实验大致分为如下几个部分：

- 启动 Mininet，运行给定的网络拓扑
- 在仿真路由器 r1 上，执行路由器程序
- 在仿真主机 h1 进行 ping 实验

整个实验流程就是以上三个步骤，十分简单，然而每一步背后都隐含着大量的工作，下面我们将从几个方面来详细解释实验的结构和流程。

3.1 数据结构

整个实验涉及到的数据结构数量大，逻辑复杂，从两个方面对这些数据结构分类并分析：

3.1.1 基于数据包 packet 的一系列数据结构

首先，建立起传统 ISO/OSI 的七层网络模型和 TCP/IP 协议栈四层模型的概念，以及两者的对应关系，如图 2 所示：

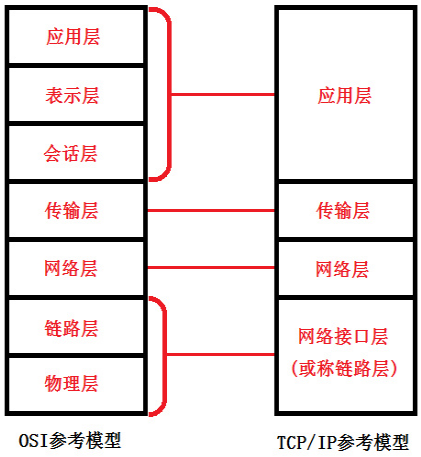

图 2：OSI 和 TCP/IP 的对比

OSI(Open System Interconnection)开放系统互联模型是由 ISO(International Organization for Standardization)国际标准化组织定义的网络分层模型，共有七层；TCP/IP 网络协议栈分为应用层（Application）、传输层（Transport）、网络层（Network）和链路层（Link）四层。

数据包的结构如图 3 所示：

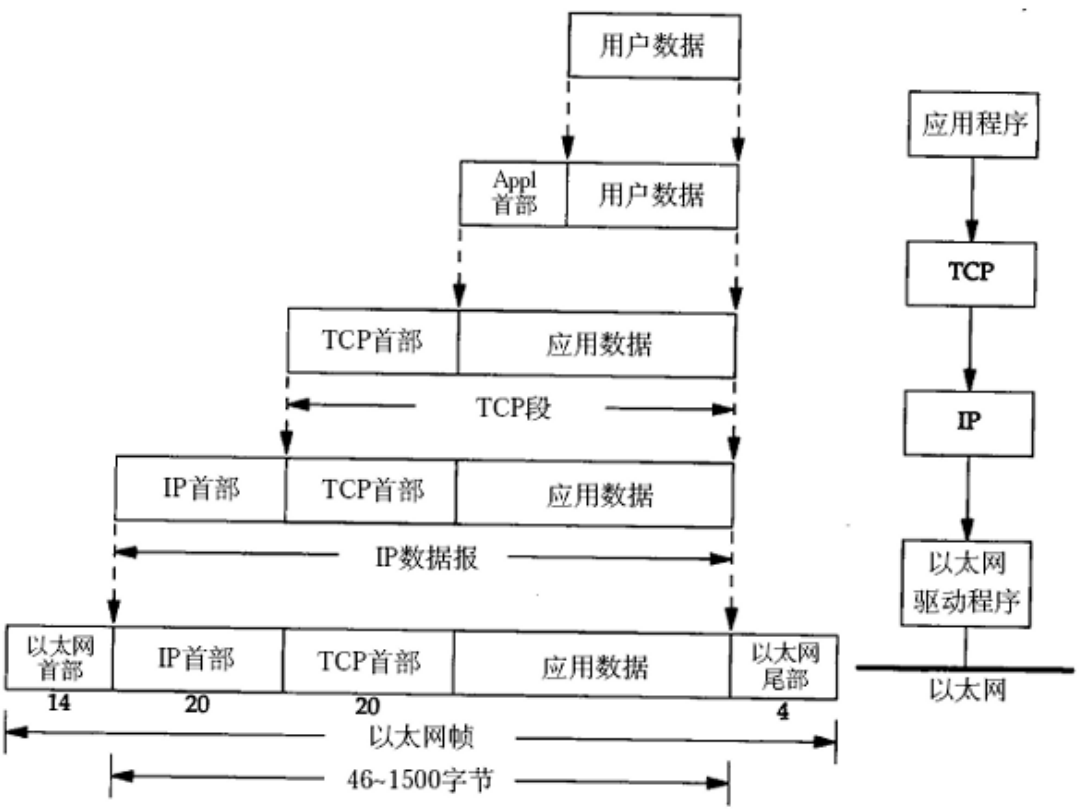

图 3：数据包的格式

每个数据层的数据结构定义如下：

- Ethernet 首部

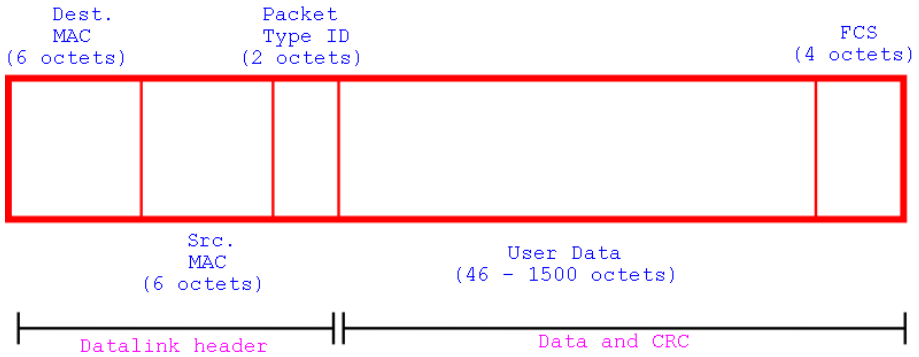

图 4：以太网首部数据格式

以太网首部的数据结构如下所示：

```
1.  /*Ethernet 首部的数据结构*/
2.  /* /include/ether.h */
3.  struct ether_header {
4.      u8 ether_dhost[ETH_ALEN];  // 目的 mac 地址
5.      u8 ether_shost[ETH_ALEN];  // 源 mac 地址
```

```
    u16 ether_type;             // protocol format
};
```

- IP 首部

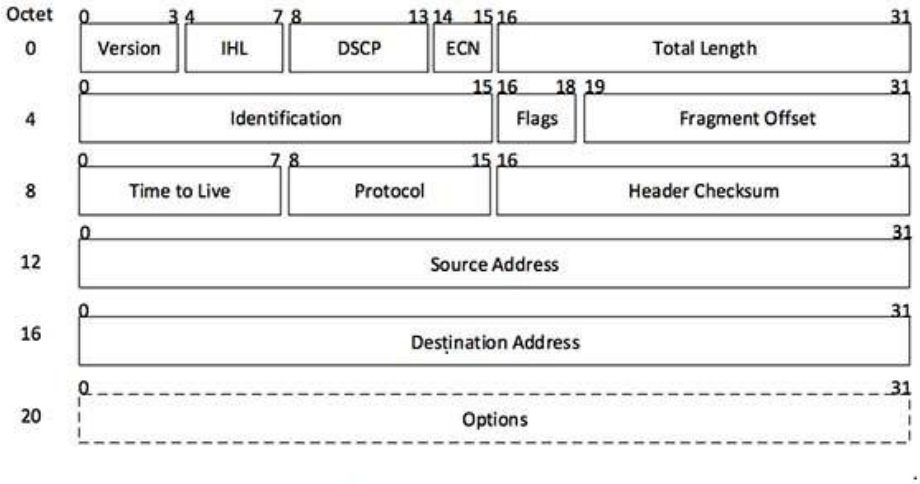

图 6：IP 首部数据格式

IP 首部的数据结构如下图所示：

```
/*IP 首部的数据结构*/
/* /include/ip.h */
struct iphdr {
#if __BYTE_ORDER == __LITTLE_ENDIAN      //小尾端
    unsigned int ihl:4;                  // length of ip header
    unsigned int version:4;              // ipv4
#elif __BYTE_ORDER == __BIG_ENDIAN       //大尾端
    unsigned int version:4;              // ip version
    unsigned int ihl:4;                  // length of ip header
#endif
    u8 tos;                              // type of service (usually set to 0)

    u16 tot_len;                         // total length of ip data
    u16 id;                              // ip identifier
    u16 frag_off;                        // the offset of ip fragment
    u8 ttl;                              // ttl of ip packet
    u8 protocol;                         // upper layer protocol
    u16 checksum;                        // checksum of ip header
    u32 saddr;                           // source ip address
    u32 daddr;                           // destination ip address
};
```

- ICMP 首部

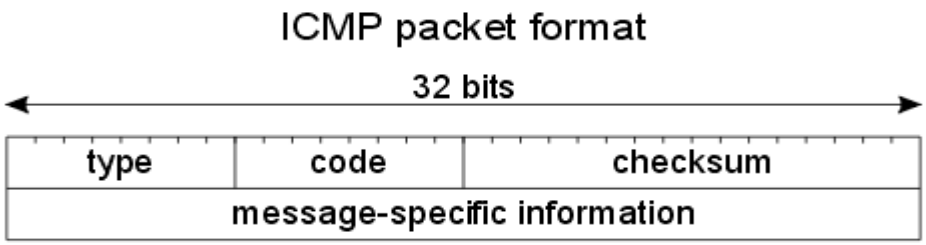

图 7：ICMP 首部数据格式

ICMP 首部数据的数据结构如下所示：

```
/*ICMP 首部的数据结构*/
/* /include/icmp.h */
struct icmphdr {
    u8  type;               // type of icmp message
    u8  code;               // icmp code
    u16 checksum;
    u16 icmp_identifier;    // icmp identifier, used in icmp echo request
    u16 icmp_sequence;      // icmp sequence, used in icmp echo request
}__attribute__((packed));
```

- ARP 报文的数据格式

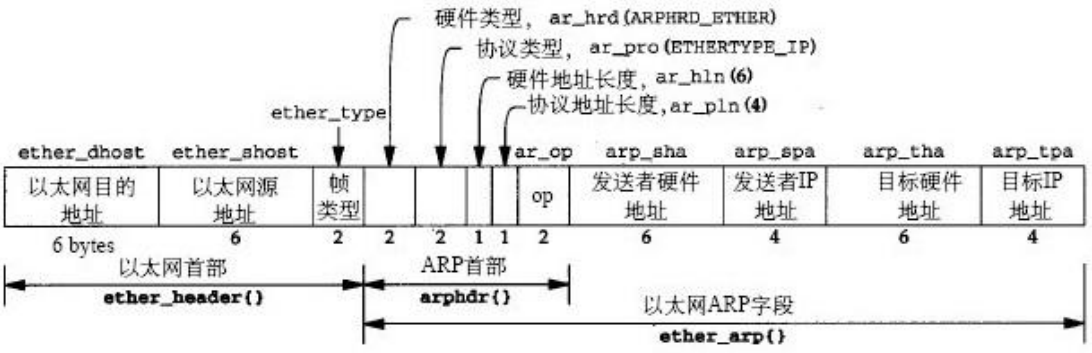

图 8：用于以太网的 ARP 请求或应答的数据格式

以太网 ARP 字段的数据结构如下所示：

```
/*ARP 报文的数据结构*/
/* /include/arp.h */
struct ether_arp {
    u16 arp_hrd;            // format of hardware address, should be 0x01
```

```
    u16 arp_pro;            // format of protocol address, should be 0x0800
    u8  arp_hln;            // length of hardware address, should be 6
    u8  arp_pln;            // length of protocol address, should be 4
    u16 arp_op;             // ARP opcode (command)
    u8  arp_sha[ETH_ALEN];  // sender hardware address
    u32 arp_spa;            // sender protocol address
    u8  arp_tha[ETH_ALEN];  // target hardware address
    u32 arp_tpa;            // target protocol address
} __attribute__ ((packed));
```

3.1.2 基于路由器组件的一系列数据结构

在 IP 路由转发实验中，路由器主要完成以下功能：

- 提取数据包信息
- 转发数据包
- IP 地址查找
- 处理 ARP 请求和应答
- ARP 缓存管理

完成这些功能依靠路由器涉及到的一系列数据结构：

- **路由表**

该路由表的每一个表项都都记录着该路由器可以转发的 ip 路由，路由表的数据结构如下所示：

```
/*路由表的数据结构*/
/* /include/rtable.h */
typedef struct {
    struct list_head list;  //链表
    u32 dest;               // destination ip address
    u32 gw;                 // ip address of next hop
    u32 mask;               // network mask of dest
    int flags;              // flags
    char if_name[16];       // name of the interface
    iface_info_t *iface;    // pointer to the interface structure
} rt_entry_t;
```

- **以太网接口表**

一个路由器可以连接两个或者多个子网，每一个子网，接在路由器的一个网卡上，路由器内部有一个专门的以太网接口链表记录着每个接口的基本信息，指向该表的指针也是路由表结构的成员之一，接口信息的数据结构如下所示：

```c
/*以太网接口的数据结构*/
/* /include/base.h */
typedef struct {
    struct list_head list;      // 链表（记录所有以太网接口信息）
    int fd;                     // 文件描述符（收或发）
    int index;                  // 以太网接口的 id
    u8  mac[ETH_ALEN];          // 该接口的 mac 地址
    u32 ip;                     // 该接口的 ip 地址
    u32 mask;                   // 该接口的子网掩码
    char name[16];              // 该接口的名字
    char ip_str[16];            // ip 地址的字符串
} iface_info_t;
```

- ARP 缓存管理数据格式

```c
/*ARP 缓存管理的数据结构*/
/* /include/arpcache.h */
typedef struct {
    struct arp_cache_entry entries[MAX_ARP_SIZE];// 最多存储的项数
    struct list_head req_list;                   // 挂起的 arp 请求包，指向
    pthread_mutex_t lock;                        // 缓存表查询，更新操作锁
    pthread_t thread;                            // 老化操作对应的线程
 } arpcache_t;
```

- ARP 缓存表的数据格式

```c
/*ARP 缓存表的数据结构*/
/* /include/arpcache.h */
struct arp_cache_entry {
    u32 ip4;            // 目的 IP 地址，本地字节序
    u8 mac[ETH_ALEN];   // IP 地址对应的 MAC 地址
    time_t added;       // 该条目添加的时间
    int valid;          // 该条目是否有效
};
```

- arp_req

```c
/*ARP 缓存表的数据结构*/
/* /include/arpcache.h */
struct arp_req {
    struct list_head list;
```

```
    iface_info_t *iface;    // the interface that will send the pending packet
s
    u32 ip4;                // destination ip addressÄ¿µÄipµØÖ·
    time_t sent;            // last time when arp request is sent
    int retries;            // number of retries
    struct list_head cached_packets;    // pending packets
};
```

- cached_pkt

```
/*ARP 缓存表的数据结构*/
/* /include/arpcache.h */
struct cached_pkt {
    struct list_head list;
    char *packet;           // packet
    int len;                // the length of packet
};
```

3.2 实现逻辑

在理解清楚项目的整个数据结构的基础上，需要完善几个关键的函数功能，例如，在/ip.c 文件中，查找大前缀长度匹配的函数，给定目的 ip 地址，需要在已有的路由表中，查找出与给定 ip 地址最匹配的表项。我们在工作就是在“rt_entry_t *longest_prefix_match(u32 dst)”函数中编写实现上述算法的代码。

下面将从函数的角度，将我们所完善的所有功能函数都列举出来，并且加以注释和分析它们的逻辑功能：

3.2.1 ip.c 文件

- rt_entry_t *longest_prefix_match(u32 dst)

  最长前缀匹配（Longest prefix match）是指在 IP 协议中被路由器用于在路由表中进行选择的一个算法。因为路由表中的每个表项都指定了一个网络，所以一个目的地址可能与多个表项匹配，此时需要找出最明确的一个表项，即子网掩码最长的一个，就叫做最长前缀匹配。之所以这样称呼它，是因为这个表项也是路由表中，与目的地址的高位匹配得最多的表项。代码如下所示：

```
1.  //lookup in the routing table, to find
2.  //the entry with the same and longest prefix
3.  rt_entry_t *longest_prefix_match(u32 dst)     //根据最大前缀匹配算法
4.  {                                             //返回一个表项指针
5.      rt_entry_t *selected = NULL;
6.      rt_entry_t *entry = NULL;
7.      u32 max_mask = 0;
8.      /*查找路由表的一个循环算法*/
9.      list_for_each_entry(entry, &rtable, list)//头文件中被定义为循环
10.     {
11.         //路由表目的地址和子网掩码与运算以确定网段
12.         u32 net = entry->dest &entry->mask;
13.         //将要 ping 的目的地址和子网掩码与运算以确定目的网段
14.         u32 temp = dst & entry->mask;
15.         if ((dst & entry->mask) == net)
16.         //如果目的地址和路由器处于同一网段
17.         {
18.             if (entry->mask > max_mask)
19.             {
20.                 selected = entry;
21.                 max_mask= entry->mask;        //选择更大长度的子网掩码
22.             }
23.         }
24.     }
25.     /*查找路由表后，返回被选中的路由表表项，它是一个结构体指针，
26.       指向路由表格式 rt_entry_t 的指针*/
27.     return selected;
28. }
```

- void ip_forward_packet(u32 ip_dst, char *packet, int len)
  从指定的接口转发 IP 数据包，当转发数据包时，您应该检查 TTL，转发一次 TTL 减一，当 TTL 小于零时，发送 ICMP 包，回复错误类型，丢弃该包；T 当 ip 头部数据发生变化时，更新校验和，确定转发包的下一跳，然后通过 iface_send_packet_by_arp()函数转发。代码如下所示：

```
1.  void ip_forward_packet(u32 ip_dst, char *packet, int len)
2.  {
3.      struct iphdr *ip_header = packet_to_ip_hdr(packet);
4.      ip_header->ttl -=1;
5.      if(ip_header->ttl <= 0)
6.      {
7.         icmp_send_packet(packet, len, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL);
8.         free(packet);//将该数据包丢弃
```

```
    }
    else
    {   //重新设置 checksum
        ip_header->checksum = ip_checksum(ip_header);
        //查询下一跳 IP 地址和端口
        rt_entry_t *entry = longest_prefix_match(ip_dst);
        u32 next_hop = get_next_hop(entry, ip_dst);
        /*有了对应的最佳 interface 和目的 ip 地址，计算下一跳的 ip*/
        iface_info_t *iface = entry->iface;
        //转发数据包
        iface_send_packet_by_arp(iface, next_hop, packet, len);
    }
}
```

3.2.2 icmp.c 文件

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

  icmp 函数，首先从以太网包中，拆解出 icmp 包，初始化重新给 icmp 包赋值，一下是几种路由查找失败回复原因：

  - type=3 code=0 路由表查找失败
  - type=3 code=1 ARP 查询失败
  - type=11 code=1 TTL 为 0
  - type=0 code=0 ping 本端口

```
// send icmp packet: construct icmp packet and send the packet by ip_send_packet
void icmp_send_packet(const char *in_pkt, int len, u8 type, u8 code)
{
    char* icmp_packet;//当前用于做 icmp 处理的包，调用下一个函数时要使用该参数
    struct iphdr *temp_ip_header = (struct iphdr *)(in_pkt + ETHER_HDR_SIZE);
    icmp_packet = malloc(len);
    /*以下三行用于定义 icmp_packet 包自身的指针*/
    struct ether_header *eth_header = (struct ether_header *)icmp_packet;
    struct iphdr *ip_header=(struct iphdr *)(icmp_packet+ETHER_HDR_SIZE);
    struct icmphdr *icmpheader =(struct icmphdr *)(icmp_packet+ETHER_HDR_SIZE+IP_BASE_HDR_SIZE);
    //设置 ethernet header
    struct ether_header *packet_header = (struct ether_header *)in_pkt;
    eth_header->ether_type=htons(ETH_P_IP);
    /*以下两句调用 memcpy 用于重置 MAC 地址*/
```

```
15.     memcpy(eth_header->ether_dhost,packet_header->ether_shost,ETH_ALEN);
16.     memcpy(eth_header->ether_shost,packet_header->ether_dhost,ETH_ALEN);
17.     //设置 ip header
18.     struct iphdr *packet_ip_header =(struct iphdr *)(in_pkt+ETHER_HDR_SIZE);
19.     u32 temp_ntohl_daddr=ntohl(packet_ip_header->daddr);
20.     u32 temp_ntohl_saddr=ntohl(packet_ip_header->saddr);
21.     ip_init_hdr(ip_header,temp_ntohl_daddr,temp_ntohl_saddr,len-
    ETHER_HDR_SIZE,IPPROTO_ICMP);
22.
23.     if(type == 3&&code==0)//路由表查找失败
24.     {
25.         //设置 icmp 头部
26.         icmpheader->type=3;
27.         icmpheader->code=0;
28.         icmpheader->checksum=icmp_checksum(icmpheader,ICMP_HDR_SIZE+ICMP_COPIE
    D_DATA_LEN);
29.         icmpheader->icmp_identifier=0;//前四位设为 0
30.         icmpheader->icmp_sequence=0;
31.         //设置 Rest of Header
32.         memcpy(icmpheader+ICMP_HDR_SIZE,packet_ip_header,IP_HDR_SIZE(temp_ip_h
    eader)+ICMP_COPIED_DATA_LEN);
33.     }
34.     if(type == 3&&code==1)//ARP 查询失败
35.     {
36.         //设置 icmp 头部
37.         icmpheader->type=3;
38.         icmpheader->code=1;
39.         icmpheader->checksum=icmp_checksum(icmpheader,ICMP_HDR_SIZE+ICMP_COPIE
    D_DATA_LEN);
40.         icmpheader->icmp_identifier=0;
41.         icmpheader->icmp_sequence=0;
42.         //设置 Rest of Header
43.         memcpy(icmpheader+ICMP_HDR_SIZE,packet_ip_header,IP_HDR_SIZE(temp_ip_h
    eader)+ICMP_COPIED_DATA_LEN);
44.     }
45.     if(type ==11&&code==0)//TTL 值为 0
46.     {
47.         //设置 icmp 头部
48.         icmpheader->type=11;
49.         icmpheader->code=0;
50.         icmpheader->checksum=icmp_checksum(icmpheader,ICMP_HDR_SIZE+ICMP_COPIE
    D_DATA_LEN);
51.         icmpheader->icmp_identifier=0;
52.         icmpheader->icmp_sequence=0;
```

```
53.         //设置 Rest of Header
54.         memcpy(icmpheader+ICMP_HDR_SIZE,packet_ip_header,IP_HDR_SIZE(temp_ip_h
    eader)+ICMP_COPIED_DATA_LEN);
55.     }
56.     if(type == 0&&code==0)//Ping 本端口
57.     {
58.         //设置 icmp 头部
59.         struct icmphdr *temp_icmp_header = (struct icmphdr *)(in_pkt + ETHER_H
    DR_SIZE+IP_BASE_HDR_SIZE);
60.         icmpheader->type=0;
61.         icmpheader->code=0;
62.         icmpheader->icmp_identifier=temp_icmp_header->icmp_identifier;
63.         icmpheader->icmp_sequence=temp_icmp_header->icmp_sequence;
64.
65.         memcpy(icmpheader+ICMP_COPY_SIZE,temp_icmp_header+ICMP_COPY_SIZE,len-
    ETHER_HDR_SIZE-IP_BASE_HDR_SIZE-8);
66.         //设置 Rest of Header
67.         icmpheader->checksum=icmp_checksum(icmpheader,len-ETHER_HDR_SIZE-
    IP_BASE_HDR_SIZE);
68.     }
69.     /*再调用 ip 层函数，用以封装 ip 层*/
70.     ip_send_packet(icmp_packet, len);
71.     return NULL;
72. }
```

3.2.3 arp.c 文件

- void arp_send_request(iface_info_t *iface, u32 dst_ip)

  arp发送请求包，初始化以太网包，重新给arp协议赋值，将以太网头部的源MAC地址设置为转发端口的MAC地址，将目的MAC地址设置为对应的MAC地址，如果目的MAC不可知是用FF：FF：FF：FF：FF：FF，为广播包，通过iface_send_packet()函数转发

```
1.  // send an arp request: encapsulate an arp request packet,
2.  //send it out through iface_send_packet
3.  void arp_send_request(iface_info_t *iface, u32 dst_ip)
4.  {
5.      //定义包的大小
6.      size_t packet_size = ETHER_HDR_SIZE+sizeof(struct ether_arp);
7.      char *packet= malloc(packet_size);//给包空间
8.      //转化成 mac 头包
```

```
    struct ether_header *header = (struct ether_header *)packet;
    //arp 包的大小
    struct ether_arp *arp= (struct ether_arp*)(packet+ETHER_HDR_SIZE);
    header->ether_type=htons(ETH_P_ARP);//类型定义
    memset(header->ether_dhost,0xff,ETH_ALEN);//将后面的字节用 ff 代替
    memcpy(header->ether_shost,iface->mac,ETH_ALEN);//接口地址给目的 mac 地址
    arp->arp_hrd=htons(0x01);
    arp->arp_pro=htons(0x0800);
    arp->arp_hln=6;
    arp->arp_pln=4;
    arp->arp_op=htons(ARPOP_REQUEST);
    arp->arp_spa=htonl(iface->ip);
    arp->arp_tpa=htonl(dst_ip);
    memset(arp->arp_tha,0,ETH_ALEN);
    memcpy(arp->arp_sha,iface->mac,ETH_ALEN);

    iface_send_packet(iface,packet,packet_size);
}
```

- void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)

  arp发送应答包，初始化以太网包，重新给arp协议赋值，将以太网头部的源MAC地址设置为转发端口的MAC地址，将目的MAC地址设置为对应的MAC地址，再对ARP报文信息进行设置。设置完毕，通过调用端口发送函数转发。

```
/*arp 请求应答函数*/
/*先对 arp 包进行封装，再通过 iface_send_packet()发送*/
void arp_send_reply(iface_info_t *iface, struct ether_arp *req_hdr)
{
    /*提取包的大小*/
    size_t packet_size = ETHER_HDR_SIZE+sizeof(struct ether_arp);
    char *packet= malloc(packet_size);   //像系统申请一定内存空间
    /*以下两行用于定义 arp 包自身的指针*/
    struct ether_header *header = (struct ether_header *)packet;
    struct ether_arp *arp= (struct ether_arp*)(packet+sizeof(struct ether_head
er));
    /*在以太网首部将 type 设置为 ARP*/
    header->ether_type=htons(ETH_P_ARP);
    /*调用 mencpy 函数设置以太网首部的 mac 地址*/
    memcpy(header->ether_dhost,req_hdr->arp_sha,ETH_ALEN);
    memcpy(header->ether_shost,iface->mac,ETH_ALEN);
```

```
    /*以下部分是设置 arp 首部信息*/
    arp->arp_hrd=htons(0x01);       //设置硬件类型
    arp->arp_pro=htons(0x0800);     //设置协议类型
    arp->arp_hln=6;                 //硬件地址长度
    arp->arp_pln=4;                 //协议地址长度
    arp->arp_op=htons(ARPOP_REPLY);//设置 arp_op 位
    /*以下部分设置 arp 报文信息*/
    arp->arp_spa=htonl(iface->ip);                  //发送者硬件地址
    arp->arp_tpa=htonl(req_hdr->arp_spa);           //目的硬件地址
    memcpy(arp->arp_sha,iface->mac,ETH_ALEN);       //发送者 IP 地址
    memcpy(arp->arp_tha,req_hdr->arp_sha,ETH_ALEN);//目标 IP 地址
    /*将封装完毕的 arp 应答包通过 interface 转发*/
    iface_send_packet(iface,packet, packet_size);
}
```

3.2.4 arpcache.c 文件

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

```
//查找 IP-> mac 映射
//遍历散列表来查找是否存在具有相同 IP 的条目
//和 MAC 地址与给定的参数
int arpcache_lookup(u32 ip4, u8 mac[ETH_ALEN])
{
    int found = 0;
    int i=0;
    struct arp_cache_entry find_arp;
    pthread_mutex_lock(&(arpcache.lock)); //加锁
    for (i=0;i<32;i++)
    {
        find_arp =arpcache.entries[i]; //结构体复制
        if(find_arp.ip4 == ip4&&find_arp.valid==1)//请注意 valid 是否初始化过？
        {
            found=1;
            //缓存中找到匹配映射，更新目的 MAC 地址
            memcpy(mac,find_arp.mac,ETH_ALEN);
        }
    }
    pthread_mutex_unlock(&(arpcache.lock)); //解锁
    return found;
}
```

- void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int len)

```
1.  //将数据包附加到 arpcache
2.  //在散列表中查找存储挂起的数据包，如果已经存在的话
3.  //使用相同的 IP 地址和 iface（表示相应的 arp 请求已发出）
4.  //只需在该条目的尾部添加此数据包即可
5.  //（该条目可能包含多个数据包）；否则，malloc 一个新的条目
6.  //与给定的 IP 地址和 iface，追加数据包，并发送 ARP 请求
7.  void arpcache_append_packet(iface_info_t *iface, u32 ip4, char *packet, int le
    n)
8.  {
9.      fprintf(stdout, "arpcache_append_packet(..) called \n");
10.     //fprintf(stdout, "TODO: implement this function please.\n");
11.     struct arp_req *pos=NULL,*q;
12.     if (list_empty(&(arpcache.req_list))) {
13.         //缓存请求列表为空
14.         //新建 ARP 请求映射
15.         struct arp_req *req_ip = malloc(sizeof(struct arp_req));
16.         req_ip->iface = iface;
17.         req_ip->ip4 = ip4;
18.         time(&req_ip->sent);
19.         req_ip->retries = 1;//sweep 修改
20.         //ARP 请求映射对应数据包
21.         struct cached_pkt *req_ip_packet = malloc(sizeof(struct cached_pkt));

22.         req_ip_packet->packet = packet;
23.         req_ip_packet->len = len;
24.         init_list_head(&req_ip->cached_packets);
25.
26.         //数据包添加到映射等待发送数据包列表尾部
27.         list_add_tail(&req_ip_packet->list,&req_ip->cached_packets);
28.         pthread_mutex_lock(&(arpcache.lock));//加锁
29.
30.         //将等待数据包添加 ARP 请求列表尾部
31.         list_add_tail(&req_ip->list,&(arpcache.req_list));
32.         pthread_mutex_unlock(&(arpcache.lock));//解锁
33.         arp_send_request(iface,ip4);
34.     }
35.     else
36.     {
37.         list_for_each_entry_safe(pos,q,&(arpcache.req_list),list)
38.         {
```

```
39.             if (pos->ip4 == ip4)
40.             {
41.                 if (pos->iface->ip == iface->ip)//          比较 iface 和 ip
42.                 {
43.                     //等待发送请求列表中有匹配地址映射，将数据包数据添加到相应请求映射数据包列表尾部
44.                     struct cached_pkt *req_ip_packet = malloc(sizeof(struct cached_pkt));
45.                     req_ip_packet->packet = packet;
46.                     req_ip_packet->len = len;
47.                     pthread_mutex_lock(&(arpcache.lock));//加锁
48.                     list_add_tail(&req_ip_packet->list, &pos->cached_packets);
49.                     pthread_mutex_unlock(&(arpcache.lock));//解锁
50.                     return NULL;
51.                 }
52.             }
53.         }
54.         //没有匹配地址映射，新建请求结构体，将数据包数据添加到相应请求映射数据包列表尾部
55.         struct arp_req *req_ip = malloc(sizeof(struct arp_req));
56.         req_ip->iface = iface;
57.         req_ip->ip4 = ip4;
58.         time(&req_ip->sent);
59.         req_ip->retries = 1;//sweep 修改
60.         struct cached_pkt *req_ip_packet = malloc(sizeof(struct cached_pkt));
61.         req_ip_packet->packet = packet;
62.         req_ip_packet->len = len;
63.         init_list_head(&req_ip->cached_packets);
64.         pthread_mutex_lock(&(arpcache.lock));//加锁
65.         list_add_tail(&req_ip_packet->list,&req_ip->cached_packets);
66.         list_add_tail(&req_ip->list,&(arpcache.req_list));
67.         pthread_mutex_unlock(&(arpcache.lock));//解锁
68.         arp_send_request(iface,ip4);
69.     }
70.     return NULL;
71. }
```

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

```
1. //将 IP-> mac 映射插入到 arpcache 中，如果有挂起的数据包
2. //等待这个映射，为每个映射填充以太网头，然后发送出去
3. void arpcache_insert(u32 ip4, u8 mac[ETH_ALEN])
```

```
4.  {
5.      //fprintf(stdout, "TODO: implement this function please.\n");
6.      fprintf(stdout, "arpcache_insert(..) called \n");
7.      int find=0;
8.      int i=0;
9.      while(find==0)
10.     {
11.         if((arpcache.entries[i]).ip4 == 0)
12.         {
13.             pthread_mutex_lock(&(arpcache.lock));//加锁
14.             (arpcache.entries[i]).ip4 = ip4;
15.             memcpy((arpcache.entries[i]).mac,mac,ETH_ALEN);
16.             time((arpcache.entries[i]).added);
17.             (arpcache.entries[i]).valid = 1;
18.             pthread_mutex_unlock(&(arpcache.lock));//解锁
19.             find=1;
20.         }
21.         i++;
22.         if(i==32) find=-1;
23.     }
24.     if(find==-1)
25.     {
26.         //缓存列表已满 32，生成随机下标，替换成传入 ip->mac 映射
27.         int index =rand()%32;
28.         pthread_mutex_lock(&(arpcache.lock));//加锁
29.         (arpcache.entries[index]).ip4 = ip4;
30.         memcpy((arpcache.entries[index]).mac,mac,ETH_ALEN);
31.         time((arpcache.entries[index]).added);
32.         (arpcache.entries[index]).valid = 1;
33.         pthread_mutex_unlock(&(arpcache.lock));//解锁
34.     }
35.     struct arp_req *pos,*q;
36.     list_for_each_entry_safe(pos,q,&(arpcache.req_list),list)//将在缓存中等待该
    映射的数据包，依次填写目的 MAC 地址，转发出去，并删除掉相应缓存数据包
37.     {
38.         if(pos->ip4==ip4)
39.         {
40.             if(!list_empty(&(pos->cached_packets)))
41.             {
42.                 struct cached_pkt *req_ip_packet,*p;
43.                 pthread_mutex_lock(&(arpcache.lock));//加锁
44.                 list_for_each_entry_safe(req_ip_packet,p,&(pos->cached_packets
    ),list)//发送该 ip 下队列中数据包
45.                 {
```

```
46.
47.                     struct ether_header *header =(struct ether_header *)req_ip
    _packet->packet;
48.                     //设置 mac 地址
49.                     memcpy(header->ether_dhost,mac,ETH_ALEN);
50.                     //转发此包
51.                     iface_send_packet(pos->iface,req_ip_packet->packet,req_ip_
    packet->len);
52.                     //删除此包
53.                     list_delete_entry(&(req_ip_packet->list));
54.                     free(req_ip_packet);
55.                 }
56.                 pthread_mutex_unlock(&(arpcache.lock));//解锁
57.             }
58.             pthread_mutex_lock(&(arpcache.lock));//加锁
59.             list_delete_entry(&(pos->list));
60.             free(pos);
61.             pthread_mutex_unlock(&(arpcache.lock));//解锁
62.         }
63.
64.     }
65.     return NULL;
66. }
```

- void *arpcache_sweep(void *arg)

```
1.  /*
2.  每 1 秒钟，运行 arpcache_sweep 操作
3.          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
4.          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
5.          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
6.  */
7.  void *arpcache_sweep(void *arg)
8.  {
9.      fprintf(stdout, "TODO: implement this function please.\n");
10.     fprintf(stdout, "arpcache_sweep(..) called \n");
11.     while(1)
12.     {
13.         int i;
14.         pthread_mutex_lock(&(arpcache.lock));//加锁
15.         time_t nowtime = time((time_t*)NULL);
16.         for (i=0;i<32;i++)
17.         {
```

```
18.             if(nowtime-(arpcache.entries[i]).added>15)//大于 15 秒则清除记录
19.             {
20.                 memset(&(arpcache.entries[i]),0, sizeof(struct arp_cache_entry
    ));
21.             }
22.         }
23.         struct arp_req *pos,*q;
24.         nowtime = time((time_t*)NULL);
25.         list_for_each_entry_safe(pos,q,&(arpcache.req_list),list)
26.         {
27.             if(nowtime-pos->sent>1&&pos->retries<6)//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
28.             {
29.                 iface_info_t *iface = pos->iface;
30.                 u32 ip4 = pos->ip4;
31.                 arp_send_request(iface,ip4);
32.                 pos->sent= nowtime;
33.                 pos->retries +=1;
34.             }
35.             else if(pos->retries>5)//对该队列下的数据包依次回复 ICMP
    （Destination Host Unreachable）消息
36.             {
37.                 struct cached_pkt *req_ip_packet,*n;
38.                 list_for_each_entry_safe(req_ip_packet,n,&(pos->cached_packets
    ),list)
39.                 {
40.                     icmp_send_packet(req_ip_packet->packet,req_ip_packet->len,
    ICMP_DEST_UNREACH,ICMP_HOST_UNREACH);
41.                 }
42.
43.             }
44.         }
45.         pthread_mutex_unlock(&(arpcache.lock));//解锁
46.         sleep(1);
47.     }
48. }
```

4 实验步骤

将代码写好并整理成文件夹\router_stack 放入 Linux 操作系统，依次操作以下步骤：

1) 在/router_stack 中打开终端，执行 make 指令，生成可执行文件；
2) 进入/scripts 文件中，分别执行

```
sh disable_arp.sh
sh disable_ip_forwarding.sh
sh disable_icmp.sh
```

用来禁止协议栈的相关功能

3) 执行命令

```
sudo python topo/router_topo.py
```

进入 mininet 控制台

4) 在 mininet 控制台下，执行命令

```
xterm h1 h2 h3 r1
```

生成 3 台仿真主机和 1 台仿真路由器

5) 此时，网络试验平台已经搭建完毕，首先在 r1 窗口执行./router 命令，用于启动路由功能。接下来，在 h1 窗口输入 ping 命令，根据窗口反馈的信息分析试验结果。（如果在执行./router 命令时报错权限不够，解决的办法可能是：先执行 chmod +x router ，再去执行./router 即可正常开启路由器功能）

5 实验结果与分析

在 h1 主机上一次输入 ping 执行，详情如下：

- ping 10.0.1.1，能够ping通，实验结果如图9所示：

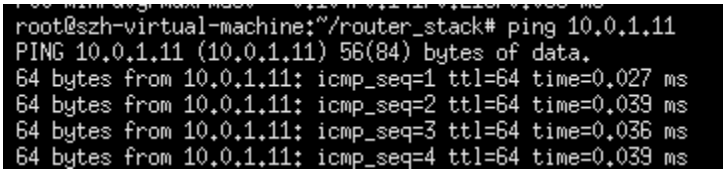

图9：ping 10.0.1.1

- ping 10.0.2.22，能够ping通，实验结果如图10所示：

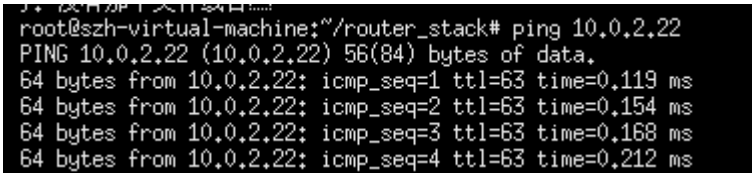

图10：ping 10.0.2.22

- ping 10.0.3.33，能够ping通，实验结果如图11所示：

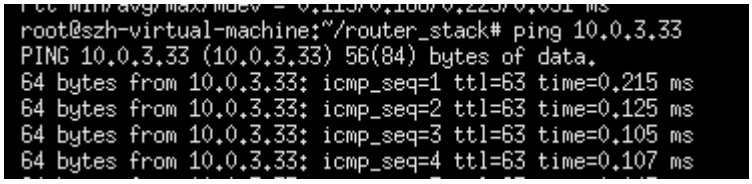

图11：ping 10.0.3.33

- ping 10.0.2.11，无返回响应，实验结果如图12所示：

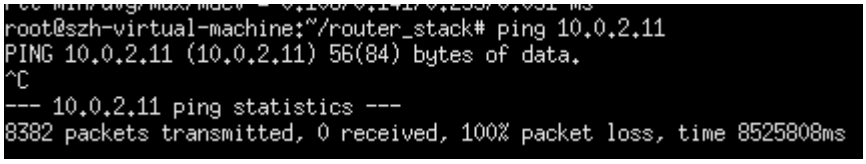

图12：ping 10.0.2.11

- ping 10.0.4.1，无返回响应，实验结果如图13所示：

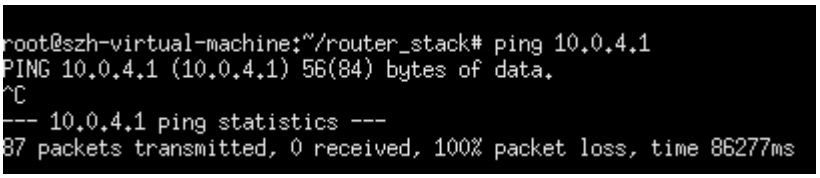

图13：ping 10.0.4.1

实验问题：当用h1 ping 10.0.4.1 和10.0.2.11时，不返回网络不可达和主机不可达，根据我们组的讨论，一致认为是我们在内存申请上出现了问题，由于时间所限还没有及时解决，之后我们会继续研究这个问题。