

**计算机网络**

**课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | IP数据报的转发及收发 | | | | | |
| 姓名 | 许坤彬 | | 院系 | 计算学部 | | |
| 班级 | 2203103 | | 学号 | 2022113586 | | |
| 任课教师 | 聂兰顺 | | 指导教师 | 聂兰顺 | | |
| 实验地点 | G001 | | 实验时间 | 2024/10/31 | | |
| 实验课表现 | 出勤、表现得分(10) |  | 实验报告  得分(40) |  | 实验总分 |  |
| 操作结果得分(50) |  |
| 教师评语 | | | | | | |
|  | | | | | | |

****

|  |
| --- |
| 实验目的： |
| - 了解原始套接字的基本概念和使用方法。  - 掌握路由器进行 IP 数据报转发的基本原理。  - 实现于原始套接字的 IP 数据报的发送和接收  - 实现基于原始套接字的 IP 数据报转发，包括 AF\_INET 和 AF\_PACKET 原始套接字的  应用。 |
| 实验内容： |
| (1) 使用虚拟机实现多主机间的 UDP 数据报收发及转发  利用虚拟机搭建实验环境，掌握 Linux 下的 Socket 网络编程。  选做 1：改进程序，示例程序只实现了一个数据包（携带 1 条消息）的发、转、收过程，要求实现每条消息由控制台输入，并且不限制发送消息的数目。  (2) 基于单网口主机的 IP 数据转发及收发  在局域网中，模拟 IP 数据报的路由转发过程。通过原始套接字实现了完整的数据封装过程，实现了 UDP 头部、IP 头部、MAC 帧头部的构造。  选做 2：扩展实验的网络规模，由原始方案中 3 台主机增加到不少于 5 台主机，共同完成 IP 数据报转发及收发过程，要求采用转发表改进示例程序，增加程序通用性。  (3) 基于双网口主机的路由转发  构造了静态路由表，并实现了不同子网间的 IP 数据报查表转发过程。  选做 3：通过完善路由表，改进示例程序实现双向传输 |
| 实验过程： |
| 1. 实验环境  本次实验环境采用Ubuntu 22.04-server作为操作系统，使用vmware17创建虚拟机完成实验。使用C语言进行代码编写，gcc版本为11.4.0，使用vscode插件remote-ssh连接虚拟机ip进行代码编写。  2. 使用虚拟机实现多主机间的UDP数据报收发及转发  首先使用创建虚拟机，安装ubuntu22.04系统，在网络适配器上选择NAT模式，完成后利用vmware的虚拟机克隆方式，克隆出三台虚拟机，通过vmware的设置界面，为不同虚拟机分配不同的mac地址。  随后开始编写这一部分的代码，首先为了能够实现重复多次发送，需要为send\_ip程序增加一个while循环，并且每次通过询问用户是否需要再次发送，同时为了通过控制台输入的方式来完成信息的输入，通过fgets的方法从控制台缓冲中读取字符流。每次发送完都需要将meesage的内存空间释放，防止出现叠加的情况；接下来在forward\_ip程序中，首先需要将原本的逻辑修改为无限循环，即每次接收数据报检验并发送之后都要循环，从而实现多次接收转发；最后一部分是recv\_ip程序，同样需要为程序加上循环逻辑，同时注意每次使用缓冲区前通过memset(buffer, 0, sizeof(buffer))完成缓冲区的清零，防止上次接收到的字符串和这次接收到的字符串出现叠加。  3. 基于单网口主机的IP数据转发及收发  （1）路由转发表的实现  首先创建结构体router\_entry，其包含三个属性，分别是dest\_ip，next\_hop和iface，随后创建一个router\_entry的数组，作为路由表。在每次router接收到数据报时，首先根据数据中的dest\_ip在路由表中进行匹配寻找，找到之后返回一个指向对应router\_entry的指针。  （2）通过ARP协议获取对应下一跳ip的MAC地址  通过ioctl(sockfd, SIOCGARP, &req)可以从arp缓存中获得下一跳IP地址对应的MAC地址，这里MAC地址的获得分为两种情况，一种是IP地址的MAC地址已经被缓存在ARP的cache中，则通过这个函数可以直接获得MAC地址；一种则是IP地址对应的MAC地址没有被缓存到ARP的cache中，则此时需要先发送一个ARP广播请求，然后监听返回的ARP响应，从响应中获取MAC地址，并将其存储在ARP的缓存中。利用源IP、源mac、目的IP和网卡接口类型来构造ARP广播请求，进而使用socket的sendto方法发送，随后利用recvfrom进行接收。在这里sockfd需要同时处理UDP协议和ARP协议，所以需要通过sockfd = socket(AF\_PACKET, SOCK\_RAW, htons(ETH\_P\_ALL));来创建套接字。    如图所示可以看到两种获得MAC地址方式的区别，我们使用arp -n指令来查询arp缓存，第一次查询时arp缓存中是没有192.168.48.130的，之后我们用ping的方式对其发送icmp数据报，ping通之后我们再次调用arp -n指令查询arp缓存，发现192.168.48.130出现在了缓存中。  （3）主机拓展  在这一个过程中只需要将原本作为路由器的虚拟机克隆两份，从而得到一共五台虚拟机，修改router.c中的路由表，使得数据报能够依次通过三个路由器到达接收方。需要注意的是每次克隆虚拟机之后要在VMware中更新MAC地址，避免虚拟机之间因为占用同一MAC地址而引起运行失效。通过如图的方式可以在vmware中点击生成而产生新的MAC地址。    4. 基于双网口主机的路由转发  （1）搭建双网口主机及路由配置  首先在vmware虚拟网络编辑器中创建两张虚拟网卡，全部选择仅主机模式，并手动为他们分配子网IP为192.168.1.0和192.168.2.0。    随后克隆三份虚拟机，先在NAT模式下为他们分配三个不同的MAC地址，随后在设置中增加一个网络适配器，让他们拥有两张网卡。分别将两张网卡的网络对应设定为VMnet1和VMnet2。    完成这些配置后打开虚拟机，首先进行网络配置。首先进行路由器的配置，先在/etc/netplan/XXXX.yaml文件中增加如下的配置：    同时为了防止重启后我们的配置丢失，我们需要按照注释在/etc/cloud/cloud.d/99-disable-network-config.cfg中进行配置：    配置完成后，我们通过ifconfig能够看到对应的配置信息。    之后我们为另外两台主机进行网络配置，同样完成上面两个文件的修改，在路由的配置上则只需要处理到达本地，和去往另一子网的路由即可。    完成配置后，我们分别使用主机对路由器对应的子网进行ping，发现同一子网的接口之间是可以ping通的，则网络配置正确。    接下来开始这一部分的代码编写，首先是为了实现双向传输，需要对路由表进行扩展，将原本的单向路由修改为双向路由，增加接收方向发送方的路由。    同时在设置MAC地址时也需要改为双向：    之后分别在发送端和接收端进行修改，在接收端接收到信息之后，向发送端发送一条信息，而发送方在发送之后，会进入接收状态等待接收下一条信息。  5. 实验源码  （1）4.1  forward\_ip.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <time.h>  int main()  {    // 变量申请    int sockfd;    struct sockaddr\_in src\_addr, dest\_addr, my\_addr;    char\* buffer = NULL;    socklen\_t addr\_len;    int src\_port = 12345;  // 原始端口号    int dest\_port = 54321; // 目标端口号（接收程序的端口号）    int len = 1024;    // 创建 UDP 套接字    if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) < 0)    {      perror("socket");      return 1;    }    // 本地地址    my\_addr.sin\_family = AF\_INET;    my\_addr.sin\_port = htons(src\_port);    my\_addr.sin\_addr.s\_addr = INADDR\_ANY;    // 修改目标地址为接收程序主机的 IP 地址    dest\_addr.sin\_family = AF\_INET;    dest\_addr.sin\_port = htons(dest\_port);    dest\_addr.sin\_addr.s\_addr = inet\_addr("192.168.48.133"); // 替换为接收程序主机的实际 IP 地址    // 绑定套接字到本地地址    if (bind(sockfd, (struct sockaddr \*)&my\_addr, sizeof(my\_addr)) < 0)    {      perror("bind");      return 1;    }    while(1)    {      buffer = (char\*)malloc(len \* sizeof(char));      memset(buffer, 0, sizeof(buffer));      // 接收数据报      addr\_len = sizeof(src\_addr);      if (recvfrom(sockfd, buffer, len, 0, (struct sockaddr \*)&src\_addr, &addr\_len) < 0)      {        perror("recvfrom");        return 1;      }      time\_t now;      time(&now);      struct tm \*local = localtime(&now);      local->tm\_hour += 8;      mktime(local);  //标准化时间，处理溢出情况（例如可能加8小时后超过24小时）      printf("Time: %02d:%02d:%02d ,Datagram received: %s\n", local->tm\_hour, local->tm\_min, local->tm\_sec, buffer);      // 发送数据报      if (sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr \*)&dest\_addr,sizeof(dest\_addr)) < 0)      {        perror("sendto");        return 1;      }      printf("Datagram forwarded.\n");      free(buffer);      buffer = NULL;    }    return 0;  }  recv.c  #include <stdio.h>  #include <stdlib.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <string.h>  #include <time.h>  int main()  {    int sockfd;    struct sockaddr\_in src\_addr, my\_addr;    char\* buffer = NULL;    socklen\_t addr\_len;    int port = 54321; // 修改后的接收端口号    int len = 1024;    // 创建 UDP 套接字    if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) < 0)    {      perror("socket");      return 1;    }    // 本地地址    my\_addr.sin\_family = AF\_INET;    my\_addr.sin\_port = htons(port);    my\_addr.sin\_addr.s\_addr = INADDR\_ANY;    // 绑定套接字到本地地址    if (bind(sockfd, (struct sockaddr \*)&my\_addr, sizeof(my\_addr)) < 0)    {      perror("bind");      return 1;    }    // 接收数据报    while(1)    {      addr\_len = sizeof(src\_addr);      buffer = (char\*)malloc(len\*sizeof(char));      memset(buffer, 0, sizeof(buffer));      if (recvfrom(sockfd, buffer, len, 0, (struct sockaddr \*)&src\_addr, &addr\_len) < 0)      {        perror("recvfrom");        return 1;      }      time\_t now;      time(&now);      struct tm \*local = localtime(&now);      local->tm\_hour += 8;      mktime(local);  //标准化时间，处理溢出情况（例如可能加8小时后超过24小时）      printf("Time: %02d:%02d:%02d ,Datagram received: %s\n", local->tm\_hour, local->tm\_min, local->tm\_sec, buffer);      free(buffer);      buffer = NULL;    }    return 0;  }  send\_ip.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <unistd.h> // 为了使用 close 函数  int main()  {        int sockfd;      struct sockaddr\_in dest\_addr;      char \*message = NULL;      char answer = '\0';      int port = 12345; // 目标端口号      int len  = 1024;        // 创建 UDP 套接字      if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) < 0)      {          perror("socket");          return 1;      }      // 目标地址      dest\_addr.sin\_family = AF\_INET;      dest\_addr.sin\_port = htons(port);      dest\_addr.sin\_addr.s\_addr = inet\_addr("192.168.48.132");//目标 IP 地址");      //询问是否发送并且接收信息      do      {          printf("Continue to send?(Y or N)\n");          answer = getchar();          getchar(); // 读取并丢弃换行符          if(answer == 'Y'){              message = (char \*)malloc(len \* sizeof(char));              if (message == NULL) {                  perror("malloc");                  return 1;              }              printf("Please input the information:\n");              fgets(message, len, stdin);              message[strcspn(message, "\n")] = 0;              message[strcspn(message, "\n")] = 0;              // getchar(); // 读取并丢弃换行符              if (sendto(sockfd, message, strlen(message), 0, (struct sockaddr \*)&dest\_addr, sizeof(dest\_addr)) < 0)              {                  perror("sendto");                  return 1;              }              free(message);              message = NULL;              printf("Datagram sent.\n");          }      }while(answer == 'Y');      close(sockfd);      return 0;  }  (2) 4.2  send.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <netinet/ip.h>  #include <netinet/udp.h>  #include <netinet/ether.h>  #include <net/if.h>  #include <sys/ioctl.h>  #include <linux/if\_packet.h>  // 00:50:56:24:78:7E  #define DEST\_MAC0 0x00  #define DEST\_MAC1 0x50  #define DEST\_MAC2 0x56  #define DEST\_MAC3 0x24  #define DEST\_MAC4 0x78  #define DEST\_MAC5 0x7e  #define ETHER\_TYPE 0x0800  #define BUFFER\_SIZE 1518  #define UDP\_SRC\_PORT 12345  #define UDP\_DST\_PORT 12345  // 校验和  unsigned short checksum(void \*b, int len)  {    unsigned short \*buf = b;    unsigned int sum = 0;    unsigned short result;    for (sum = 0; len > 1; len -= 2)      sum += \*buf++;    if (len == 1)      sum += \*(unsigned char \*)buf;    sum = (sum >> 16) + (sum & 0xFFFF);    sum += (sum >> 16);    result = ~sum;    return result;  }  int main()  {    int sockfd;    struct ifreq if\_idx, if\_mac;    struct sockaddr\_ll socket\_address;    char buffer[BUFFER\_SIZE];    char msg[] = "Hello, this is a test message.111111111";    // 创建原始套接字    if ((sockfd = socket(AF\_PACKET, SOCK\_RAW, htons(ETH\_P\_ALL))) == -1)    {      perror("socket");      return 1;    }    // 获取接口索引    memset(&if\_idx, 0, sizeof(struct ifreq));    strncpy(if\_idx.ifr\_name, "ens33", IFNAMSIZ - 1);    if (ioctl(sockfd, SIOCGIFINDEX, &if\_idx) < 0)    {      perror("SIOCGIFINDEX");      return 1;    }    // 获取接口 MAC 地址    memset(&if\_mac, 0, sizeof(struct ifreq));    strncpy(if\_mac.ifr\_name, "ens33", IFNAMSIZ - 1);    if (ioctl(sockfd, SIOCGIFHWADDR, &if\_mac) < 0)    {      perror("SIOCGIFHWADDR");      return 1;    }    // 构造以太网头    struct ether\_header \*eh = (struct ether\_header \*)buffer;    eh->ether\_shost[0] = ((uint8\_t \*)&if\_mac.ifr\_hwaddr.sa\_data)[0];    eh->ether\_shost[1] = ((uint8\_t \*)&if\_mac.ifr\_hwaddr.sa\_data)[1];    eh->ether\_shost[2] = ((uint8\_t \*)&if\_mac.ifr\_hwaddr.sa\_data)[2];    eh->ether\_shost[3] = ((uint8\_t \*)&if\_mac.ifr\_hwaddr.sa\_data)[3];    eh->ether\_shost[4] = ((uint8\_t \*)&if\_mac.ifr\_hwaddr.sa\_data)[4];    eh->ether\_shost[5] = ((uint8\_t \*)&if\_mac.ifr\_hwaddr.sa\_data)[5];    for (int i = 0; i < 6; i++)    {      printf("%02x ", eh->ether\_shost[i]);    }    printf("\n");    eh->ether\_dhost[0] = DEST\_MAC0;    eh->ether\_dhost[1] = DEST\_MAC1;    eh->ether\_dhost[2] = DEST\_MAC2;    eh->ether\_dhost[3] = DEST\_MAC3;    eh->ether\_dhost[4] = DEST\_MAC4;    eh->ether\_dhost[5] = DEST\_MAC5;    eh->ether\_type = htons(ETHER\_TYPE);    // 构造 IP 头    struct iphdr \*iph = (struct iphdr \*)(buffer + sizeof(struct ether\_header));    iph->ihl = 5;    iph->version = 4;    iph->tos = 0;    iph->tot\_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr) + strlen(msg));    iph->id = htonl(54321);    iph->frag\_off = 0;    iph->ttl = 255;    iph->protocol = IPPROTO\_UDP;    iph->check = 0;    iph->saddr = inet\_addr("192.168.48.135");    iph->daddr = inet\_addr("192.168.48.130");    iph->check = checksum((unsigned short \*)iph, sizeof(struct iphdr));    // 构造 UDP 头    struct udphdr \*udph = (struct udphdr \*)(buffer + sizeof(struct ether\_header) + sizeof(struct iphdr));    udph->source = htons(UDP\_SRC\_PORT);    udph->dest = htons(UDP\_DST\_PORT);    udph->len = htons(sizeof(struct udphdr) + strlen(msg));    udph->check = 0; // UDP 校验和可选    // 填充数据    char \*data = (char \*)(buffer + sizeof(struct ether\_header) + sizeof(struct iphdr) + sizeof(struct udphdr));    strcpy(data, msg);    // 设置 socket 地址结    socket\_address.sll\_ifindex = if\_idx.ifr\_ifindex;    socket\_address.sll\_halen = ETH\_ALEN;    socket\_address.sll\_addr[0] = DEST\_MAC0;    socket\_address.sll\_addr[1] = DEST\_MAC1;    socket\_address.sll\_addr[2] = DEST\_MAC2;    socket\_address.sll\_addr[3] = DEST\_MAC3;    socket\_address.sll\_addr[4] = DEST\_MAC4;    socket\_address.sll\_addr[5] = DEST\_MAC5;    // 发送数据包    int len = sizeof(struct ether\_header) + sizeof(struct iphdr) + sizeof(struct udphdr) + strlen(msg);    printf("len=%d\n", len);    if (sendto(sockfd, buffer, len, 0, (struct sockaddr \*)&socket\_address, sizeof(struct sockaddr\_ll)) < 0)    {      perror("sendto");      return 1;    }    close(sockfd);    return 0;  }  rec.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <unistd.h>  #define PORT 12345  int main()  {    int sockfd;    struct sockaddr\_in server\_addr, client\_addr;    socklen\_t addr\_len = sizeof(client\_addr);    char buffer[1024];    // 创建 UDP 套接字    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);    if (sockfd < 0)    {      perror("Socket creation failed");      return 1;    }    // 绑定套接字到端口    memset(&server\_addr, 0, sizeof(server\_addr));    server\_addr.sin\_family = AF\_INET;    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;    server\_addr.sin\_port = htons(PORT);    if (bind(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)    {      perror("Bind failed");      return 1;    }    // 接收数据包    int recv\_len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr \*)&client\_addr, &addr\_len);    if (recv\_len < 0)    {      perror("Recvfrom failed");      return 1;    }    buffer[recv\_len] = '\0';    printf("Received message: %s\n", buffer);    close(sockfd);    return 0;  }  router.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <netinet/ip.h>  #include <netinet/if\_ether.h>  #include <netinet/ether.h>  #include <sys/socket.h>  #include <unistd.h>  #include <linux/if\_packet.h>  #include <net/if.h>  #include <sys/ioctl.h>  #include <time.h>  //#include <linux/if\_arp.h>  #include <net/if\_arp.h>  #define BUFFER\_SIZE 65536  #define MAX\_ROUTES 10  // 路由器条目  struct route\_entry {      char dest\_ip[INET\_ADDRSTRLEN];      char next\_hop[INET\_ADDRSTRLEN];      char iface[IFNAMSIZ];  };  // 路由表  struct route\_entry routing\_table[MAX\_ROUTES] = {      {"192.168.48.130", "192.168.48.130", "ens33"},      {"10.0.0.2", "10.0.0.1", "ens38"}  };  // 寻找目的ip对应的路由表  const struct route\_entry\* lookup\_next\_hop(const char \*dest\_ip) {      for (int i = 0; i < MAX\_ROUTES; i++) {          if (strcmp(routing\_table[i].dest\_ip, dest\_ip) == 0) {              return &routing\_table[i];          }      }      return NULL;  }    // 发送 ARP 广播请求  // 发送ARP请求  int send\_arp\_request(int sockfd, const char \*src\_ip\_str, unsigned char \*src\_mac, const char \*dest\_ip\_str, unsigned char \*dest\_mac, const char\* iface)  {      unsigned char buffer[42]; // ARP请求的长度为42字节      struct in\_addr src\_ip, dest\_ip;      // 将字符串形式的IP地址转换为in\_addr结构体      if (inet\_pton(AF\_INET, src\_ip\_str, &src\_ip) <= 0) {          perror("Invalid source IP address");          return -1;      }      if (inet\_pton(AF\_INET, dest\_ip\_str, &dest\_ip) <= 0) {          perror("Invalid destination IP address");          return -1;      }      // 构造以太网头      struct ether\_header \*eh = (struct ether\_header \*)buffer;      memcpy(eh->ether\_shost, src\_mac, ETH\_ALEN);      memset(eh->ether\_dhost, 0xFF, ETH\_ALEN); // 广播地址      eh->ether\_type = htons(ETH\_P\_ARP);      // 构造 ARP 请求      struct ether\_arp \*arp\_req = (struct ether\_arp \*)(buffer + sizeof(struct ether\_header));      arp\_req->arp\_hrd = htons(ARPHRD\_ETHER);      arp\_req->arp\_pro = htons(ETH\_P\_IP);      arp\_req->arp\_hln = ETH\_ALEN;      arp\_req->arp\_pln = 4;      arp\_req->arp\_op = htons(ARPOP\_REQUEST);      memcpy(arp\_req->arp\_sha, src\_mac, ETH\_ALEN);      memcpy(arp\_req->arp\_spa, &src\_ip.s\_addr, 4);      memset(arp\_req->arp\_tha, 0x00, ETH\_ALEN);      memcpy(arp\_req->arp\_tpa, &dest\_ip.s\_addr, 4);      int if\_index = if\_nametoindex(iface);      if (if\_index == 0) {          perror("Failed to get interface index");          return -1;      }      // 设置 socket 地址结构      struct sockaddr\_ll socket\_address;      memset(&socket\_address, 0, sizeof(socket\_address));      socket\_address.sll\_ifindex = if\_index;      socket\_address.sll\_halen = ETH\_ALEN;      memset(socket\_address.sll\_addr, 0xFF, ETH\_ALEN); // 广播地址      printf("ARP Request:\n");      for (int i = 0; i < sizeof(buffer); i++) {          printf("%02x ", buffer[i]);          if ((i + 1) % 16 == 0) printf("\n");      }      // 发送 ARP 请求      if (sendto(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr \*)&socket\_address, sizeof(socket\_address)) < 0)      {          perror("sendto ARP request");          return -1;      }      printf("Sent ARP request for %s\n", dest\_ip\_str);      return 0;  }  // 接收ARP回复  int receive\_arp\_response(int sockfd, const char \*target\_ip\_str, unsigned char \*target\_mac)  {      unsigned char buffer[BUFFER\_SIZE];      struct in\_addr target\_ip;      // 将字符串形式的IP地址转换为in\_addr结构体      if (inet\_pton(AF\_INET, target\_ip\_str, &target\_ip) <= 0) {          perror("Invalid target IP address");          return -1;      }      while (1)      {          int length = recvfrom(sockfd, buffer, BUFFER\_SIZE, 0, NULL, NULL);          if (length < 0)          {              perror("recvfrom ARP reply");              return -1;          }          // printf("Received packet of length %d\n", length);          struct ether\_header \*eh = (struct ether\_header \*)buffer;          if (ntohs(eh->ether\_type) == ETH\_P\_ARP)          {              struct ether\_arp \*arp\_resp = (struct ether\_arp \*)(buffer + sizeof(struct ether\_header));              struct in\_addr sender\_ip;              memcpy(&sender\_ip.s\_addr, arp\_resp->arp\_spa, 4);              if (ntohs(arp\_resp->arp\_op) == ARPOP\_REPLY && sender\_ip.s\_addr == target\_ip.s\_addr) {                  memcpy(target\_mac, arp\_resp->arp\_sha, ETH\_ALEN);                  printf("Received ARP reply from %s with MAC: ", inet\_ntoa(sender\_ip));                  for (int i = 0; i < ETH\_ALEN; i++) {                      printf("%02x ", target\_mac[i]);                  }                  printf("\n");                  return 0; // 收到目标IP的ARP回复              }          }        }      return -1;  }  // 从arp协议获得mac地址  int get\_mac\_from\_arp(int sockfd, const char \*iface, const char\* src\_ip,unsigned char \*src\_mac, const char \*dest\_ip, unsigned char \*dest\_mac) {      struct arpreq req;      struct sockaddr\_in \*sin = (struct sockaddr\_in \*)&req.arp\_pa;      memset(&req, 0, sizeof(struct arpreq));      sin->sin\_family = AF\_INET;      sin->sin\_addr.s\_addr = inet\_addr(dest\_ip);      strncpy(req.arp\_dev, iface, IFNAMSIZ - 1);      printf("ARP:\n");      if (ioctl(sockfd, SIOCGARP, &req) == 0) {            memcpy(dest\_mac, req.arp\_ha.sa\_data, ETH\_ALEN);          printf("%s", dest\_mac);      }      else      {          printf("Not found in ARP cahce, send broadcast request.\n");              // int send\_arp\_request(int sockfd, const char \*src\_ip\_str, unsigned char \*src\_mac, const char \*dest\_ip\_str, unsigned char \*dest\_mac, int if\_index)          if(send\_arp\_request(sockfd, src\_ip, src\_mac, dest\_ip, dest\_mac, iface) < 0 )          {              printf("Arp broadcast request sent failed.\n");            }          if(receive\_arp\_response(sockfd,  dest\_ip, dest\_mac)<0)          {              printf("Arp broadcast request received failed.\n");            }      }        return 0;  }    unsigned short checksum(void \*b, int len)  {    unsigned short \*buf = b;    unsigned int sum = 0;    unsigned short result;    for (sum = 0; len > 1; len -= 2)      sum += \*buf++;    if (len == 1)      sum += \*(unsigned char \*)buf;    sum = (sum >> 16) + (sum & 0xFFFF);    sum += (sum >> 16);    result = ~sum;    return result;  }    int main()  {      // 创建socket      int sockfd;      struct sockaddr saddr;      unsigned char \*buffer = (unsigned char \*)malloc(BUFFER\_SIZE);      sockfd = socket(AF\_PACKET, SOCK\_RAW, htons(ETH\_P\_ALL));      // sockfd = socket(AF\_PACKET, SOCK\_RAW, htons(ETH\_P\_ARP));        if (sockfd < 0)      {          perror("Socket creation failed");          return 1;      }      // 循环的等待接收      while (1)      {          // 接收到报文          int saddr\_len = sizeof(saddr);          int data\_size = recvfrom(sockfd, buffer, BUFFER\_SIZE, 0, &saddr, (socklen\_t \*)&saddr\_len);          if (data\_size < 0)          {              perror("Recvfrom error");              return 1;          }          // 解析报文          struct ethhdr \*eth\_header = (struct ethhdr \*)buffer;          struct iphdr \*ip\_header = (struct iphdr \*)(buffer + sizeof(struct ethhdr));          char src\_ip[INET\_ADDRSTRLEN];          char dest\_ip[INET\_ADDRSTRLEN];          inet\_ntop(AF\_INET, &(ip\_header->saddr), src\_ip, INET\_ADDRSTRLEN);          inet\_ntop(AF\_INET, &(ip\_header->daddr), dest\_ip, INET\_ADDRSTRLEN);            // 在路由表中寻找下一跳的信息          const struct route\_entry \*route = lookup\_next\_hop(dest\_ip);          if (route == NULL)          {              // printf("No route to host %s\n", dest\_ip);              continue; // 或丢弃该数据包          }          struct ifreq ifr;          strncpy(ifr.ifr\_name, route->iface, IFNAMSIZ - 1); // 使用路由表中的接口名          if (ioctl(sockfd, SIOCGIFHWADDR, &ifr) < 0) {              perror("ioctl");              return -1;          }          unsigned char src\_mac[ETH\_ALEN];          memcpy(src\_mac, ifr.ifr\_hwaddr.sa\_data, ETH\_ALEN);          // 使用 ARP 获取下一跳 MAC 地址          unsigned char next\_hop\_mac[ETH\_ALEN];          if (get\_mac\_from\_arp(sockfd, route->iface, src\_ip, src\_mac, route->next\_hop, next\_hop\_mac) < 0)          {              // printf("ARP request failed for next hop %s\n", route->next\_hop);              continue; // 或丢弃该数据包          }          // 获取当前系统时间          time\_t rawtime;          struct tm \*timeinfo;          char time\_str[100];          time(&rawtime);          timeinfo = localtime(&rawtime);          // 格式化时间字符串          strftime(time\_str, sizeof(time\_str), "%Y-%m-%d %H:%M:%S", timeinfo);            // 打印信息          printf("[%s] Captured packet from %s to %s\n", time\_str, src\_ip, dest\_ip);          // 修改 TTL          ip\_header->ttl -= 1;          ip\_header->check = 0;          ip\_header->check = checksum((unsigned short \*)ip\_header, ip\_header->ihl \* 4);          // 获取网卡接口索引          struct ifreq ifr\_mac;          memset(&ifr, 0, sizeof(ifr));          snprintf(ifr.ifr\_name, sizeof(ifr.ifr\_name), route->iface);          if (ioctl(sockfd, SIOCGIFINDEX, &ifr) < 0)          {              perror("ioctl");              return 1;          }          // 获取网卡接口 MAC 地址          memset(&ifr\_mac, 0, sizeof(ifr\_mac));          snprintf(ifr\_mac.ifr\_name, sizeof(ifr\_mac.ifr\_name), route->iface);          if (ioctl(sockfd, SIOCGIFHWADDR, &ifr\_mac) < 0)          {              perror("ioctl");              return 1;          }          // 发送数据包到目的主机          // 设置 MAC 地址和接口          struct sockaddr\_ll dest = {0};          dest.sll\_ifindex = ifr.ifr\_ifindex;          dest.sll\_halen = ETH\_ALEN;          memcpy(dest.sll\_addr, next\_hop\_mac, ETH\_ALEN);          // 构造新的以太网帧头          memcpy(eth\_header->h\_dest, next\_hop\_mac, ETH\_ALEN);                   // 目标 MAC 地址          memcpy(eth\_header->h\_source, ifr\_mac.ifr\_hwaddr.sa\_data, ETH\_ALEN);  // 源 MAC   地址          eth\_header->h\_proto = htons(ETH\_P\_IP);                               // 以太网类型为 IP          printf("Interface name: %s, index: %d\n", ifr.ifr\_name, ifr.ifr\_ifindex);          printf("Sending packet to %s with MAC: ", dest\_ip);          for (int i = 0; i < ETH\_ALEN; i++) {              printf("%02x ", next\_hop\_mac[i]);          }          printf("\n");          if (sendto(sockfd, buffer, data\_size, 0, (struct sockaddr \*)&dest, sizeof(dest)) < 0)          {              perror("Sendto error");              return 1;          }          printf("Datagram forwarded.\n");      }      close(sockfd);      free(buffer);      return 0;  }  (3) 4.3  t4-3recv.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <unistd.h>  #define PORT 12345  #define MESSAGE "Hello, fuck you too!"  int main()  {    int sockfd;    struct sockaddr\_in server\_addr, client\_addr;    socklen\_t addr\_len = sizeof(client\_addr);    char buffer[1024];    // 创建 UDP 套接字    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);    if (sockfd < 0)    {      perror("Socket creation failed");      return 1;    }    // 绑定套接字到端口    memset(&server\_addr, 0, sizeof(server\_addr));    server\_addr.sin\_family = AF\_INET;    server\_addr.sin\_addr.s\_addr = INADDR\_ANY;    server\_addr.sin\_port = htons(PORT);    if (bind(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)    {      perror("Bind failed");      return 1;    }    // 接收数据包    int recv\_len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr \*)&client\_addr, &addr\_len);    if (recv\_len < 0)    {      perror("Recvfrom failed");      return 1;    }    buffer[recv\_len] = '\0';    printf("Received message: %s\n", buffer);    // 获取客户端的IP地址    char client\_ip[INET\_ADDRSTRLEN];    inet\_ntop(AF\_INET, &client\_addr.sin\_addr, client\_ip, INET\_ADDRSTRLEN);    printf("Client IP: %s, Port: %d\n", client\_ip, ntohs(client\_addr.sin\_port));    // 发送消息回客户端    if (sendto(sockfd, MESSAGE, strlen(MESSAGE), 0, (struct sockaddr \*)&client\_addr, addr\_len) < 0)    {      perror("Sendto failed");      close(sockfd);      return 1;    }    printf("Message sent to server: %s\n", MESSAGE);    close(sockfd);    return 0;  }  t4-3route.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <netinet/ip.h>  #include <netinet/if\_ether.h>  #include <sys/socket.h>  #include <unistd.h>  #include <linux/if.h>  #include <linux/if\_packet.h>  #include <sys/ioctl.h>  #include <time.h>  #define BUFFER\_SIZE 65536  struct route\_entry  {    uint32\_t dest;    uint32\_t gateway;    uint32\_t netmask;    char interface[IFNAMSIZ];  };  struct route\_entry route\_table[2];  int route\_table\_size = sizeof(route\_table) / sizeof(route\_table[0]);  void convert\_to\_ip\_string(uint32\_t ip\_addr, char \*ip\_str)  {    struct in\_addr addr;    addr.s\_addr = ip\_addr; // htonl(ip\_addr); // 转换为网络字节序 inet\_ntop(AF\_INET, &addr, ip\_str, INET\_ADDRSTRLEN);    inet\_ntop(AF\_INET, &addr, ip\_str, INET\_ADDRSTRLEN);  }  unsigned short checksum(void \*b, int len)  {    unsigned short \*buf = b;    unsigned int sum = 0;    unsigned short result;    for (sum = 0; len > 1; len -= 2)      sum += \*buf++;    if (len == 1)      sum += \*(unsigned char \*)buf;    sum = (sum >> 16) + (sum & 0xFFFF);    sum += (sum >> 16);    result = ~sum;    return result;  }  struct route\_entry \*lookup\_route(uint32\_t dest\_ip)  {    char ip\_str[32];    for (int i = 0; i < route\_table\_size; i++)    {      if ((dest\_ip & route\_table[i].netmask) == (route\_table[i].dest & route\_table[i].netmask))      {        convert\_to\_ip\_string(dest\_ip, ip\_str);        convert\_to\_ip\_string(route\_table[i].dest, ip\_str);        return &route\_table[i];      }    }    return NULL;  }  void initialize\_route\_table()  {    route\_table[0].dest = inet\_addr("192.168.1.121");       // 子网1的目的地址    route\_table[0].gateway = inet\_addr("192.168.1.120");    // 子网1的网关    route\_table[0].netmask = inet\_addr("255.255.255.0");    strcpy(route\_table[0].interface, "ens33");    route\_table[1].dest = inet\_addr("192.168.2.121");       // 子网2的目的地址    route\_table[1].gateway = inet\_addr("192.168.2.120");    // 子网2的网关    route\_table[1].netmask = inet\_addr("255.255.255.0");    strcpy(route\_table[1].interface, "ens37");  }  int main()  {    int sockfd;    struct sockaddr saddr;    unsigned char \*buffer = (unsigned char \*)malloc(BUFFER\_SIZE);    initialize\_route\_table();    sockfd = socket(AF\_PACKET, SOCK\_RAW, htons(ETH\_P\_IP));    if (sockfd < 0)    {      perror("Socket creation failed");      return 1;    }    while (1)    {      int saddr\_len = sizeof(saddr);      int data\_size = recvfrom(sockfd, buffer, BUFFER\_SIZE, 0, &saddr, (socklen\_t \*)&saddr\_len);      if (data\_size < 0)      {        perror("Recvfrom error");        return 1;      }      if (data\_size == 0)      continue;      struct ethhdr \*eth\_header = (struct ethhdr \*)buffer;      struct iphdr \*ip\_header = (struct iphdr \*)(buffer + sizeof(struct ethhdr));      struct route\_entry \*route = lookup\_route(ip\_header->daddr);      if (route == NULL)      {        // fprintf(stderr, "No route to host\n");        continue;      }      char ip\_s[32],ip\_d[32];      convert\_to\_ip\_string(ip\_header->saddr, ip\_s);      convert\_to\_ip\_string(ip\_header->daddr, ip\_d);      // 提取发送端的源 MAC 地址      unsigned char \*src\_mac = eth\_header->h\_source;      unsigned char \*dest\_mac = eth\_header->h\_dest;      if(strcmp(ip\_s,"192.168.1.1")!=0 && strcmp(ip\_s,"192.168.2.1")!=0)      {          // 打印信息          printf("Captured packet from %s to %s\n", ip\_s, ip\_d);          printf("Source MAC (from sender): %02x:%02x:%02x:%02x:%02x:%02x\n",                 src\_mac[0], src\_mac[1], src\_mac[2], src\_mac[3], src\_mac[4], src\_mac[5]);      }      // 修改 TTL      ip\_header->ttl -= 1;      ip\_header->check = 0;      ip\_header->check = checksum((unsigned short \*)ip\_header, ip\_header->ihl \* 4);      // 发送数据包到目的主机      struct ifreq ifr, ifr\_mac;      struct sockaddr\_ll dest;      // 获取网卡接口索引      memset(&ifr, 0, sizeof(ifr));      snprintf(ifr.ifr\_name, sizeof(ifr.ifr\_name), route->interface);      if (ioctl(sockfd, SIOCGIFINDEX, &ifr) < 0)      {        perror("ioctl");        return 1;      }      // 获取网卡接口 MAC 地址      memset(&ifr\_mac, 0, sizeof(ifr\_mac));      snprintf(ifr\_mac.ifr\_name, sizeof(ifr\_mac.ifr\_name), route->interface);      if (ioctl(sockfd, SIOCGIFHWADDR, &ifr\_mac) < 0)      {        perror("ioctl");        return 1;      }      // 设置目标 MAC 地址（假设目标地址已知,此处做了简化处理，实际上，如果查找路由表后，存在“下      // 一跳”，应该利用 ARP 协议获得 route->gateway 的 MAC 地址，如果是“直接交付”的话，也应使用 ARP 协议获得      // 目的主机的 MAC 地址。）      unsigned char target\_mac[ETH\_ALEN]; // 在外部声明      if (!strcmp(ip\_d, "192.168.1.121")) {          // 这里可以设置 target\_mac 的值          memcpy(target\_mac, (unsigned char[]){0x00, 0x50, 0x56, 0x31, 0xf4, 0x92}, ETH\_ALEN);      } else if (!strcmp(ip\_d, "192.168.2.121")) {          // 这里也可以设置 target\_mac 的值         memcpy(target\_mac, (unsigned char[]){0x00, 0x50, 0x56, 0x22, 0x72, 0x32}, ETH\_ALEN); // 更改 MAC 地址      } else {          // 其他情况      }      // 替换为实际的目标 MAC 地址      memset(&dest, 0, sizeof(dest));      dest.sll\_ifindex = ifr.ifr\_ifindex;      dest.sll\_halen = ETH\_ALEN;      memcpy(dest.sll\_addr, target\_mac, ETH\_ALEN);      // 构造新的以太网帧头      memcpy(eth\_header->h\_dest, target\_mac, ETH\_ALEN);                   // 目标 MAC 地址      memcpy(eth\_header->h\_source, ifr\_mac.ifr\_hwaddr.sa\_data, ETH\_ALEN); // 源 MAC 地址      eth\_header->h\_proto = htons(ETH\_P\_IP);                              // 以太网类型为 IP      //printf("Interface name: %s, index: %d\n", ifr.ifr\_name, ifr.ifr\_ifindex);      if(strcmp(ip\_s,"192.168.1.1")!=0 && strcmp(ip\_s,"192.168.2.1")!=0)      {          printf("Destination MAC: %02x:%02x:%02x:%02x:%02x:%02x\n",                 dest\_mac[0], dest\_mac[1], dest\_mac[2], dest\_mac[3], dest\_mac[4], dest\_mac[5]);      }      if (sendto(sockfd, buffer, data\_size, 0, (struct sockaddr \*)&dest, sizeof(dest)) < 0)      {        perror("Sendto error");        return 1;      }    }    close(sockfd);    free(buffer);    return 0;  }  t4-3send.c  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #include <unistd.h>  #define DEST\_IP "192.168.2.121"  #define DEST\_PORT 12345  #define MESSAGE "Hello, fuck you!"  int main()  {    int sockfd;    struct sockaddr\_in dest\_addr;    // 创建 UDP 套接字    sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);    if (sockfd < 0)    {      perror("Socket creation failed");      return 1;    }    // 设置目的地址    memset(&dest\_addr, 0, sizeof(dest\_addr));    dest\_addr.sin\_family = AF\_INET;    dest\_addr.sin\_port = htons(DEST\_PORT);    inet\_pton(AF\_INET, DEST\_IP, &dest\_addr.sin\_addr);    // 发送数据包    if (sendto(sockfd, MESSAGE, strlen(MESSAGE), 0, (struct sockaddr \*)&dest\_addr, sizeof(dest\_addr)) < 0)    {      perror("Sendto failed");      return 1;    }    printf("Message sent to %s:%d\n", DEST\_IP, DEST\_PORT);    struct sockaddr\_in server\_addr, client\_addr;    socklen\_t addr\_len = sizeof(client\_addr);    char buffer[1024];    // 接收数据包    int recv\_len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr \*)&client\_addr, &addr\_len);    if (recv\_len < 0)    {      perror("Recvfrom failed");      return 1;    }    buffer[recv\_len] = '\0';    printf("Received message: %s\n", buffer);    close(sockfd);    return 0;  }  6. 参考文献  [1] <https://blog.csdn.net/m0_70812512/article/details/141780982>  [2] <https://blog.csdn.net/M626120212/article/details/142063268>  [3] <https://blog.csdn.net/shi450561200/article/details/135709057> |
| 实验结果： |
| 1. 使用虚拟机实现多主机间的UDP数据报收发及转发        2. 基于单网口主机的IP数据转发及收发        在这里可以看到，第一次进行信息发送时，由于没有在ARP缓存中查询到对应的MAC地址，所以发送了一个ARP广播请求，在接收到广播后得到的MAC地址之后，下次再进行发送就可以直接从arp缓存中获取。  3. 基于双网口主机的路由转发        发送方发送信息Hello, how are you!接受发接收到信息后发送Hello, I’m fine! |
| 问题讨论： |
| 1. vmware的网络适配器中的桥接模式、NAT模式和仅主机模式之间有什么区别？  在桥接模式下，虚拟机直接连接到物理网络，表现得像网络中的一个独立设备。它通过主机网络适配器直接与局域网通信，并获取和物理网络中其他设备一样的 IP 地址。这使得虚拟机可以与局域网中的其他设备进行互相访问，适用于虚拟机需要与网络中其他计算机进行直接通信的场景。  在 NAT 模式下，虚拟机通过主机的 IP 地址和外部网络进行通信，类似于路由器的网络地址转换。虚拟机没有自己独立的局域网 IP，而是通过主机进行访问。此模式适合需要连接外网，但不需要局域网其他设备访问虚拟机的情况。  在仅主机模式下，虚拟机只能与主机系统通信，而不能访问外部网络。这种模式适合用于主机和虚拟机之间进行本地测试或共享文件的场景，确保虚拟机与外部网络完全隔离。 |
| 心得体会： |
| 尝试使用了原始套接字，对原始套接字有了更多僚机，深刻了解了路由器转发机制，对路由协议、ARP协议和它们之间的联系有了更多的认识，对于ubuntu系统的网络连接配置有了更多的了解。 |