|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验（4） | 实验名称 | | 隧道协议仿真程序设计与实现 | |
| 学时数 | 4时数 | | 实验类型 | V验证性 V设计性 □综合性 |
| 每组人数 | 3人 | | 难度 | 较高 |
| 覆盖知识点 | | 网络安全，IP协议安全，NAT地址转化 | | |
| 实验内容：   1. 编写IP隧道协议仿真程序 2. IP in IP报文传输仿真   实验开展方式：   1. 以组为单位进行实验，3人1组 2. 课内外结合的方式完成实验，课内完成实验内容、实验要求等讲解，以及实验结果、实验报告检查；课内外结合完成程序设计、编制与调试   实验软硬件环境要求：  要求在四台机器上实现  实验讲授内容：   1. IP隧道协议原理讲解 2. IP in IP隧道协议仿真实现技术讲解   学生实践内容：   1. 编制IP隧道仿真程序，包括四个程序，IP请求、隧道封装、隧道解封、IP接收等四个仿真程序 2. 进行IP隧道仿真程序部署 3. 进行IP in IP隧道协议演示 4. 编写实验报告 | | | | |

**《信息安全概论》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | |  | | | **姓名** |  |
| **实验题目** | 隧道协议仿真程序设计与实现 | | | | | |
| **实验时间** | **2021.5.21** | | **实验地点** | **DS3 305** | | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 □√综合性** | | |
| 教师评价：  □算法/实验过程正确； □源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | | |
| 一、实验目的   1. 理解IP隧道协议原理 2. 学会IP in IP隧道协议仿真实现 | | | | | | |
| 二、实验项目内容   1. 编制IP隧道仿真程序，包括四个程序，IP请求、隧道封装、隧道解封、IP接收等四个仿真程序 2. 进行IP隧道仿真程序部署 3. 进行IP in IP隧道协议演示 4. 编写实验报告 | | | | | | |
| 三、实验过程或算法（源程序）  本次实验实现 IP in IP Tunnel 仿真，首先应实现基本 IP 协议，在此基础上再进行封装。  如下： （一）基本IP通信功能实现 在Linux上使用C语言在数据链路层上通信，实现IPv4和UDP协议以及应用层封装。 1. 数据链路层（Ethernet Frame） 使用Ethernet帧结构  dst\_mac(6) | src\_mac(6) | protocal(2) | payload(46-1500) | fcs(4)  fcs使用crc32算法进行计算，接收端如果校验失败抛出异常。  payload长度不足46时，使用bzero进行0比特填充，长于1500则抛出异常。 2. 网络层（IP packet） 使用自定义的固定的IPv4头（20字节）。  头检验和check使用16位半字相加取结果补码，接收端如果校验失败抛出异常。  **IP分片**  发送时：   * 偏移值offset使用8个字节为偏移单位，每一片的长度为8的整数倍   接收时：   * 如果MF（more frag) = 1或者frag 0ff 不为0表示分片 * MF=1并且 offset偏移0，为第一片 * MF=0，为最后一片 * 其余情况 ，为中间片   根据IP的id进行分片重组。 3. 运输层（UDP datagram） 使用了自定义的UDP头（8字节）。  UDP伪首部 、UDP首部 、UDP的数据一起计算校验和，接收端如果校验失败抛出异常。 4. 应用层（Chat） 使用多线程库，分别创建读线程和发线程，用于双向的聊天。  输出解析后的MAC帧结构信息、IP Packet结构信息，UDP datagram结构信息。 其他 对于非unsigned char类型，使用htons和ntohs进行网络字节序的转化。  使用raw socket进行数据链路层的通信。 （二）基本数据结构和算法 在 myhdr.h 中，定义了本次程序使用的基本数据结构以及常量参数定义。  1. 常量参数  #define IPHDR\_IHL 5                     // IP header 长  #define UDP\_HDR\_LEN 8                   // UDP header 长  #define IP\_HDR\_LEN 20                   // IP header 长度  #define MAC\_HDR\_LEN 14                  // MAC 头长度  #define MAC\_FCS\_LEN 4                   // MAC 尾长度  #define MIN\_FRAME\_PAYLOAD\_LEN 46        // 最小frame payload长度  const uchar UDP\_PROTOCAL = (uchar)17U;  // UDP 协议号  #define MAX\_FRAME\_LEN 1518              // max frame length  typedef uchar mac\_addr[6];  typedef uchar checknum[4];  #define ETH\_PROTOCAL 0x0800     // IPv4 协议号  #define ERROR\_TAKEN 0           // 错误发生类型（调试）  #define MY\_ETH\_PROTOCAL 0x4376  // 自定义协议号  #define MY\_IP\_PROTOCAL 0x0001   // 自定义协议号  // #define MTU 1460  #define MTU 100                  // 为了测试，使用较小的MTU  #define MAX\_IP\_PACKET\_LEN 65536  // 最大的IP packet长度  2. IP header  struct myiphdr {      uchar ihl : 4, version : 4;      // 版本 0.5个字节 // IHL 0.5字节      uchar tos;        // 区分服务 1      ushort tot\_len;   // 总长度 2      ushort id;        // 标识 2      ushort frag\_off;  // 分段偏移量 2      uchar ttl;        // 生存周期 1      uchar protocol;   // 协议 1      ushort check;     // 头校验和 2      uint saddr;       // 源地址 4      uint daddr;       // 目标地址 4  };  3. UDP header  struct myudphdr {      ushort sport;      ushort dport;      ushort len;      ushort check;  };  在 utils.c 中实现了基本的算法，如int转二进制输出：char \*int\_to\_binary(int n);  如 crc32 校验算法：  unsigned int crc32(char \*data, int len);  如 ip 地址点分法转32位整数：  unsigned short csum(unsigned short \*src, int count);  如 ip 整数转点分法  void ipaddr2str(unsigned int in, char result[]); （三）场景介绍 实现了一个简单的 ipip tunnel 模拟，基于 UDP，IPv4（包括分片），RAW\_SOCKET 二次通信。  为了方便实现，实现的是 IPv4 in IPv4，并且为了区分，将 inner IP-version 设置为 5，下称为IPv5，以进行区分。  作为很粗糙的实现，这里假设了很多条件，给出场景：   * PC1 位于 net-A 中，R1 是 net-A 的路由器，PC2 位于 net-B 中，R2 是 net-B 的路由器。 * 假设 PC1 和 PC2 进行通信，实际场景中，如：PC1 ping PC2，涉及 ARP 协议的实现。这里简单处理，指定 MAC 地址进行通信。 * 程序在 linux 上，基于 raw\_socket 进行数据传输。指定 PC1 向 MAC\_R1 发送帧 frame，如果是需要转发到 net-B 中的数据， R1 将数据转发到 MAC\_R2，R2 再进一步将数据转发给 PC2.   这里只模拟 PC1 发送到 PC2 的数据包，即只模拟跨子网的数据传输，且规定子网内部全部使用 IPv5 进行通信，但是两个子网之间只能通过 IPv4 进行通信。  需要进一步封装IPIP，并通过 tunnel 传输到另一个子网的场景如下：   * mac 地址和本机 mac 地址匹配 * 源 ip 位于自己所在的子网 * 目的 ip 位于对端所在的子网 * ip 的 version 为 5 （需要封装为IPv4）   需要将接受的数据解封装，在内部子网中转发的场景如下：   * mac 地址和本机 mac 地址匹配 * 源 ip 位于对端所在的子网 * 目的 ip 位于自己所在的子网 * ip 的 version 为 4 （需要封装为IPv5）  （四）整体流程  1. **PC1 中输入要发送的数据**    * 数据将会被封装为 UDP    * 封装为 IPv5    * 封装为 Ethernet    * 通过 RAW\_SOCKET 发送给 R1（省略 ARP） 2. **R1 接受数据**    * 查看 Ethernet 的 目的 MAC 地址，与本地 MAC 地址匹配    * 解析为 IPv5    * 发现版本号为 5 并且源 IP 位于子网 net-A，目的 IP 位于子网 net-B 中，封装为 IPv4，并且IP的protocol设置为 IPIP 协议号    * 封装为 Ethernet    * 通过 RAW\_SOCKET 发送给 R2（省略ARP） 3. **R2 接受数据**    * 查看 Ethernet 的 目的 MAC 地址，与本地 MAC 地址匹配    * 解析为 IPv4    * 发现IP的protocol为 IPIP 协议号，并且源 IP 位于子网 net-A，目的 IP 位于子网 net-B 中，解析为 IPv5    * 封装为 Ethernet    * 通过 RAW\_SOCKET 发送给 PC2（省略ARP） 4. **PC2 接受数据**    * 查看 Ethernet 的 目的 MAC 地址，与本地 MAC 地址匹配    * 解析为 IPv5    * 解析为 UDP    * 打印数据部分 payload  （五）发送端 PC1 实现 **1. 创建 raw\_socket**  /\*\*   \* create the raw socket   \* type is PF\_PACKET, representing the data link layer   \* use sock\_raw to send   \*/  sock\_raw\_fd = socket(PF\_PACKET, SOCK\_RAW, htons(ETH\_P\_ALL));  **2. 设置绑定各种网卡参数**  **3. clinet1 启动写线程**      // 启动多线程      pthread\_t send\_id, recv\_id;      // 发数据线程      if (pthread\_create(&send\_id, NULL, (void \*)send\_thread, NULL) != 0) {          printf("Create pthread error!\n");          exit(1);      }      pthread\_join(send\_id, NULL);  **4. 应用层接收数据并发送：**      // port      uint sport = 5000;      uint dport = 8080;      printf("send thread start...\n");      // init long data      // char long\_data[5000];      // for (int i = 0; i < 4000; i++) long\_data[i] = 'a';      // send\_data(sport, dport, long\_data);      while (1) {          char data[MAX\_IP\_PACKET\_LEN];          fgets(data, MAX\_IP\_PACKET\_LEN, stdin);          send\_data(sport, dport, data);      }  **5. 从应用层向下，发至运输层，进行封装和UDP的校验**  /\*\*   \* 从应用层发往运输层   \* @param sport 源端口   \* @param dport 目标端口   \* @param payload payload结构   \* @param payload\_len payload长度   \*/  void send\_to\_tansport\_layer(ushort sport, ushort dport, char \*payload,                              uint payload\_len) {      struct myudphdr hdr;      char udp\_datagram[MAX\_IP\_PACKET\_LEN];      make\_udphdr(sport, dport, payload\_len, &hdr);      // check\_udp();      // printf("%d\n", sizeof(hdr));      memcpy(udp\_datagram, &hdr, sizeof(hdr));      memcpy(udp\_datagram + sizeof(hdr), payload, payload\_len);      // memcpy(udp\_datagram + sizeof(hdr), payload, payload\_len);      uint udplen = payload\_len + sizeof(hdr);  **// 伪首部 + UDP首部 + 数据一起计算校验和**      hdr.check = calc\_check\_udphdr(udp\_datagram, udplen, srcip, dstip);  **// 更新udp头的check**      memcpy(udp\_datagram, &hdr, sizeof(hdr));      // printf("udphdr check: %x\n", hdr.check);      // printf("[DEBUG] send udp: %s\n", udp\_datagram + 8);      send\_to\_network\_layer(UDP\_PROTOCAL, udplen, udp\_datagram);  }  **6. 从运输层向下，发至网络层，进行封装、IP分片和校验**  /\*\*   \* 从运输层发往网络层   \* @param protocal 运输层的协议号   \* @param payload\_len payload长度   \* @param payload   \*/  void send\_to\_network\_layer(uchar protocal, uint payload\_len, uchar \*payload) {      // DEBUG UDP LEN      struct myudphdr u;      memcpy(&u, payload, 8);      struct myiphdr iphdr;      char ip\_packet[MAX\_FRAME\_LEN \* 2];      bzero(ip\_packet, sizeof(ip\_packet));      iphdr.version = 5;  // 我定义为ipv4      iphdr.ihl = IPHDR\_IHL;      iphdr.tos = 0;      iphdr.tot\_len =          htons(payload\_len + (iphdr.ihl << 2));  // 这里是20个字节的固定长度      iphdr.id = rand() % 10000;      iphdr.frag\_off = 0;      iphdr.ttl = 128;      iphdr.protocol = protocal;      iphdr.daddr = dstip;      // printf("in make iphdr: %u %u\n", saddr, daddr);      iphdr.saddr = srcip;      iphdr.check = 0;      struct myiphdr test;      // printf("check iphdr sizeof iphdr %ld\n", sizeof(\*iphdr));      uchar more\_frag;  // store  **// IP 分片**      if (payload\_len + (iphdr.ihl << 2) > MTU) {          // 首先确定需要分几片          // 第一片带有IP hdr和 UDP hdr，后面的片没有 UDP hdr          // 第一片的最大payload为1480          int nfrag = payload\_len / (MTU - IP\_HDR\_LEN);          int last\_frag\_len = payload\_len % (MTU - IP\_HDR\_LEN);          nfrag = last\_frag\_len ? nfrag + 1 : nfrag;          int each\_size = MTU - IP\_HDR\_LEN;          int left = payload\_len;          for (int i = 0; i < nfrag; i++) {              iphdr.tot\_len = 0;              left -= each\_size;              more\_frag = left > 0;              iphdr.tot\_len = htons(more\_frag ? MTU : last\_frag\_len + IP\_HDR\_LEN);              // calc frag off              // 这里使用8个字节作为偏移单位，要求每个分片的长度是8字节的整数倍              iphdr.frag\_off = ceil(i \* each\_size / 8);              // set tag 1              iphdr.frag\_off = iphdr.frag\_off | 0x2000;              if (more\_frag) {                  memcpy(ip\_packet + sizeof(iphdr), payload + i \* each\_size,                         each\_size);              } else {                  iphdr.frag\_off = iphdr.frag\_off & 0xDFFF;  // more frag = 0                  // printf("last frag off %x\n", iphdr.frag\_off);                  memcpy(ip\_packet + sizeof(iphdr), payload + i \* each\_size,                         each\_size + left);              }              // iphdr.frag\_off = htons(iphdr.frag\_off);              iphdr.check = 0;              // printf("frag\_off: %x\n", iphdr.frag\_off);              iphdr.frag\_off = htons(iphdr.frag\_off);              // printf("frag\_off: %x\n", iphdr.frag\_off);              iphdr.check = csum((ushort \*)&iphdr, sizeof(iphdr));              memcpy(ip\_packet, &iphdr, sizeof(iphdr));              // printf("frag off %x\n", htons(iphdr.frag\_off));              send\_to\_datalink\_layer(ip\_packet, ntohs(iphdr.tot\_len),                                     ERROR\_TAKEN);          }      } else {          iphdr.check = csum((ushort \*)&iphdr, sizeof(iphdr));          memcpy(ip\_packet, &iphdr, sizeof(iphdr));          memcpy(ip\_packet + sizeof(iphdr), payload, payload\_len);          send\_to\_datalink\_layer(ip\_packet, ntohs(iphdr.tot\_len), ERROR\_TAKEN);      }  }  7. **从网络层向下，发至数据链路层，进行封装**  /\*\*   \* 从网络层发往数据链路层   \* @param 要发送的数据   \* @param 用于测试checksum. 0: normal, 1:crc check failed   \* @return   \*/  void send\_to\_datalink\_layer(char data[], int data\_len, int error) {      char frame[MAX\_FRAME\_LEN];      int frame\_size;      // printf("send eth datalen: %d\n", data\_len);      // 测试时候为了过滤自己协议      frame\_size = make\_frame(&dst\_mac, &my\_mac, htons(MY\_ETH\_PROTOCAL), data,                              data\_len, &frame[0]);      // 实际ip协议      if (!frame\_size) {          printf("[ERROR]:  data length must be less than 1500\n");          return;      }      // printf("[INFO]: send data size %d\n", frame\_size);      if (error == 1)          frame[14] = ~frame[14];      else if (error == 2)          frame[21] = ~frame[21];      else if (error == 3)          frame[36] = ~frame[36];      // 输出frame 测试      // output\_frame(frame, frame\_size);      // if (cnt > 0) printf("%x", \*(unsigned char \*)p);      int len = send(sock\_raw\_fd, frame, frame\_size, 0);      if (len == -1) {          perror("sendto");      } else {          printf("[SUCCESS] send %d size data\n\n", frame\_size);      }  }  8. 调用socket send 进行数据发送    int len = send(sock\_raw\_fd, frame, frame\_size, 0); （六）网关路由R1、R2 实现 **1. 启动读线程**      // 启动多线程      pthread\_t send\_id, recv\_id;      // 读数据线程      if (pthread\_create(&recv\_id, NULL, (void \*)recv\_thread, NULL) != 0) {          printf("Create pthread error!\n");          exit(1);      }      // pthread\_join(send\_id, NULL);      pthread\_join(recv\_id, NULL);  **2. 读取数据链路层数据，并进行匹配。只处理发往本地 MAC 的帧，且协议为自定义协议号 4376 的帧，并输出帧结构信息。**  /\*\*   \* 接收数据   \*/  void recv\_data() {      int nbyte;      char buffer\_in[1536];  // buffer to read from socket      mac\_addr src\_mac;  // src mac address      mac\_addr dst\_mac;  // dst mac address      frame\_index = 0;      while (1) {          buffer\_in[0] = '\0';          socklen\_t addr\_length = sizeof(struct sockaddr\_ll);          nbyte = recvfrom(sock\_raw\_fd, buffer\_in, 1536, 0,                           (struct sockaddr \*)&sll\_recv,                           &addr\_length);  // read from socket          if (nbyte < MAC\_HDR\_LEN + MAC\_FCS\_LEN) {              continue;          }          ushort protocal;  // protocal type          // 解析mac帧          memcpy(&dst\_mac, &buffer\_in, 6);       // get dst addr          memcpy(&src\_mac, &buffer\_in[6], 6);    // get src addr          memcpy(&protocal, &buffer\_in[12], 2);  // get protocal type  **// 根据MAC地址过滤，只读取发往本地MAC的帧**          int eq = mac\_equal(dst\_mac, my\_mac);          if (!eq) {              // 取消以下注释，输出发往非本机MAC帧的MAC地址              // printf("[NOTE]: not frame to local. To ");              // output\_mac(dst\_mac);              // printf("\n");              continue;          }  **// 这里为了测试，自定义协议protocol为4376（学号尾号），过滤，只监听此协议的MAC帧**          if (protocal != ntohs(MY\_ETH\_PROTOCAL)) {              continue;          }          // 更新帧号          frame\_index++;          printf("\n");          // 解析MAC帧的内容  **// 使用crc32进行FCS校验**          uint cas\_rec = crc32(buffer\_in, nbyte - 4);          int check\_res = frame\_crc\_check(cas\_rec, buffer\_in, nbyte);  **// 如果checksum出错，丢弃该帧**          if (check\_res == -1) continue;          // 输出MAC帧信息          printf(              "[MAC]    from MAC %02X:%02X:%02X:%02X:%02X:%02x | mac protocal %x "              "| receive %d byte data\n",              src\_mac[0], src\_mac[1], src\_mac[2], src\_mac[3], src\_mac[4],              src\_mac[5], ntohs(protocal), nbyte);          // 解析IP包          // char \*packet = (char \*)malloc(2 \* nbyte - MAC\_FCS\_LEN - MAC\_HDR\_LEN);          char packet[MAX\_FRAME\_LEN];          memcpy(packet, &buffer\_in[MAC\_HDR\_LEN],                 nbyte - MAC\_FCS\_LEN - MAC\_HDR\_LEN);          read\_from\_datalink\_layer(packet, nbyte);      }  }  **3. 从数据链路层向上传递数据，解析IP。**    /\*\*   \* 从数据链路层读取数据   \* @param packet IP包   \* @param nbyte 字节数   \* @return -1表示IP头校验失败   \*/  int read\_from\_datalink\_layer(char packet[], int nbyte, mac\_addr dmac) {      struct myiphdr iniphdr;      ipaddr inipaddr;      memcpy(&iniphdr, packet, IP\_HDR\_LEN);      char dstip\_c[16], srcip\_c[16];      ipaddr2str(iniphdr.daddr, dstip\_c);      ipaddr2str(iniphdr.saddr, srcip\_c);      // iniphdr.check = 0;      // csum      int iphdr\_error = check\_iphdr\_error(&iniphdr, sizeof(iniphdr));      if (iphdr\_error) {          printf(              "[ERROR]: frame[%d] takes error caused by failed IP "              "checksum.\n",              frame\_index);          return -1;      }      ushort hs\_tot\_len = ntohs(iniphdr.tot\_len);      printf(          "[IP]     from %s to %s | version %u | ihl %2u | tos %u | total "          "len "          "%4u | id %6u | "          "ttl %4u | protocal %2u | check %x\n",          srcip\_c, dstip\_c, iniphdr.version, iniphdr.ihl, iniphdr.tos, hs\_tot\_len,          iniphdr.id, iniphdr.ttl, iniphdr.protocol, iniphdr.check);      // read\_from\_network\_layer(nbyte, packet, &iniphdr, frame\_index);      // 进行 ip in ip tunnel 封装      // read\_from\_network\_layer(nbyte, packet, &iniphdr, frame\_index);      ushort hs\_frag\_off = ntohs(iniphdr.frag\_off);      // printf("hs\_frag\_off: %x\n", hs\_frag\_off);      int dont\_frag = hs\_frag\_off & 0x4000;  **/\*\***  **\* @brief**  **\* 边界设备在收到从IPv6网络侧发来的报文后，如果报文的目的地址不是自身且下一跳出接口为Tunnel接口**  **\* 就要把收到的IPv6报文作为数据部分，加上IPv4报文头，封装成IPv4报文。**  **\***  **\* IPv6 over**  **\* IPv4手动隧道转发机制为：当隧道边界设备的IPv6侧收到一个IPv6报文后，**  **\* 根据IPv6报文的目的地址查找IPv6路由转发表，如果该报文是从此虚拟隧道接口转发出去，则根据隧道接口配置的隧道源端和目的端的IPv4地址进行封装。**  **\* 封装后的报文变成一个IPv4报文，交给IPv4协议栈处理。报文通过IPv4网络转发到隧道的终点。隧道终点收到一个隧道协议**  **\* 文后，进行隧道解封装。并将解封装后的报文交给IPv6协议栈处理。**  **\*/**      // printf("mac equal: %d\n", mac\_equal(my\_mac, dmac));      // printf("dstip %x left\_net\_ip: %x\n", iniphdr.daddr,      //        (ipstr2addr(right\_net\_ip) & ipstr2addr(right\_net\_mask)));      if (iniphdr.version == 5 ||          mac\_equal(my\_mac, dmac) &&              ipstr2addr(right\_net\_ip) ==                  (iniphdr.daddr & ipstr2addr(right\_net\_mask)) &&              ipstr2addr(left\_net\_ip) ==                  (iniphdr.saddr & ipstr2addr(left\_net\_mask))) {          // tunnel          make\_ipip\_tunnel(MY\_IP\_PROTOCAL, hs\_tot\_len, packet, dont\_frag);      } else if (iniphdr.version == 4 && mac\_equal(my\_mac, dmac) &&                 ipstr2addr(right\_net\_ip) ==                     (iniphdr.saddr & ipstr2addr(right\_net\_mask)) &&                 ipstr2addr(left\_net\_ip) ==                     (iniphdr.daddr & ipstr2addr(left\_net\_mask))) {          // 如果是ipv4，进行解包成ipv6，这里不做双协议栈，只做ipv6          hs\_tot\_len = ntohs(iniphdr.tot\_len);          if ((hs\_frag\_off & 0x2000) || hs\_frag\_off) {  **// 如果more frag 或者 frag 0ff 表示分片**  **// 1. MF=1 and no offset 第一片**  **// 2. MF=0 最后一片**  **// 3. 其余情况 中间片**              int islast = !((hs\_frag\_off & 0x2000) >> 13);              ushort addroff = 8 \* (hs\_frag\_off & 0x1FFF);              int isfirst = !islast && !addroff;              // printf("fragoff: %x, hs\_frag\_off: %x islast %x addroff %x\n",              //        iniphdr.frag\_off, hs\_frag\_off, islast, addroff);              if (isfirst) {                  printf("\n(first fragment)\n");                  printf("----------------------------------------------\n");                  ip\_packet\_id = iniphdr.id;                  memcpy(store, packet, hs\_tot\_len);                  first\_iphdr = &iniphdr;                  total\_byte += hs\_tot\_len;                  // DEBUG UDP LEN                  struct myudphdr u;                  memcpy(&u, packet + IP\_HDR\_LEN, 8);                  // printf("udplen in first: %d\n", ntohs(u.len));              } else if (islast) {                  memcpy(store + IP\_HDR\_LEN + addroff, packet + IP\_HDR\_LEN,                         hs\_tot\_len - IP\_HDR\_LEN);                  total\_byte += hs\_tot\_len - IP\_HDR\_LEN;                  // 进行重组                  // read\_from\_network\_layer(total\_byte, store,                  // first\_iphdr,frame\_index);                  unpack\_ipip\_tunnal(total\_byte, store, first\_iphdr, frame\_index);                  // 清空                  ip\_packet\_id = -1;                  first\_iphdr = NULL;                  bzero(store, sizeof(store));                  total\_byte = MAC\_FCS\_LEN + MAC\_HDR\_LEN;                  printf("\n(last fragment)\n");                  printf("----------------------------------------------\n");              } else if (ip\_packet\_id == iniphdr.id) {                  memcpy(store + IP\_HDR\_LEN + addroff, packet + IP\_HDR\_LEN,                         hs\_tot\_len - IP\_HDR\_LEN);                  total\_byte += hs\_tot\_len - IP\_HDR\_LEN;              }          } else {              unpack\_ipip\_tunnal(nbyte, packet + IP\_HDR\_LEN, &iniphdr,                                 frame\_index);              // read\_from\_network\_layer(nbyte, packet, &iniphdr, frame\_index);          }          // send\_to\_network\_layer(IP\_PROTOCAL, ipip\_len, packet);      }  }  这里涉及到两个函数的实现：  **IPIP封装**  void make\_ipip\_tunnel(uchar protocal, uint payload\_len, uchar \*payload,int dont\_frag);  **IPIP 解封装**  void unpack\_ipip\_tunnal(int nbyte, char \*packet, struct myiphdr \*iniphdr,int frame\_index);  如果需要将数据转发到另一个子网的PC去，需要再封装一层IP Header，函数make\_ipip\_tunnel实现这一功能。实际上，实现方式和make\_ip一致（同为IPv4格式，这里为了区分设置version为5。  如果需要将接收来自另一个子网PC的数据内在自己的子网内部转发，则需要对IPIP解封装，实现方式和解封装IP一致。  对于分片，如果Inner Dont Frag为1，则Outer Don’t Frag 也为1，表示不分片。否则，如果Inner IP 分片，则Outer IP 也可能分片，具体取决于MTU大小。对于Inner IP 的MTU 和 Outer 的 MTU 可以不一致。 （七）接收端 PC2 实现 区别在于接收到IP数据包后，直接进一步解析UDP，得到数据。  /\*\*   \* 从网络层读取数据   \* 处理后直接是应用层显示了   \* @param nbyte 收到的总的字节数   \* @param packet ip packet   \* @param iniphdr ip 头   \* @param frame\_index 帧序号（仅用来打印）   \*/  void read\_from\_network\_layer(int nbyte, char \*packet, struct myiphdr \*iniphdr,                               int frame\_index) {      // 解析udp      char udp\_datagram[MAX\_IP\_PACKET\_LEN];      struct myudphdr inudphdr;      char data[MAX\_IP\_PACKET\_LEN];      ushort leftlen = nbyte - MAC\_HDR\_LEN - MAC\_FCS\_LEN - IP\_HDR\_LEN;      memcpy(udp\_datagram, packet + IP\_HDR\_LEN, leftlen);      udp\_datagram[leftlen] = '\0';      memcpy(&inudphdr, &udp\_datagram, 8);      // 先进行udp的检验      // printf("[DEBUG] recv udp: %s\n", udp\_datagram + 8);      ushort udphdr\_eror = calc\_check\_udphdr(udp\_datagram, ntohs(inudphdr.len),                                             iniphdr->saddr, iniphdr->daddr);      if (udphdr\_eror) {          printf(              "[ERROR]: frame[%d] takes error caused by failed UDP "              "checksum.\n",              frame\_index);          return;      }      printf("[UDP]    source port %u | dest port %u\n", ntohs(inudphdr.sport),             ntohs(inudphdr.dport));      // printf("UDP len: %d\n", ntohs(inudphdr.len));      strcpy(data, &udp\_datagram[UDP\_HDR\_LEN]);      printf("[DATA]   %s\n", data);  } | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程 （一）启动 这里在同一个 linux 环境下进行模拟。开启四个终端，分别执行：   |  | | --- | | root@pc1:$ make client1run  root@pc2:$ make client2run  root@pc3:$ make router1run  root@pc4:$ make router2run |  （二）测试结果 不分片的测试结果：    发生分片时的测试结果：   （三）真实环境下的修改 之前的测试基于一台机器的模拟。为了在真实环境下进行测试，需要做一定的改动。由于机器限制，我们使用一台电脑（连接校园网内网），一台云服务器（具有公网IP），进行通信。其中，PC1和Router1我们在本地电脑上模拟，PC2和Router2在云服务器上模拟。  主要需要解决两个问题。  Router1（本地）向Router2（服务器）发送数据包时，由于我们的程序是在数据链路层进行解析封装的，需要指定下一跳的mac地址。因为为了获得mac地址，我们向服务器发送 ping数据包，通过wireshark抓包看下一跳的mac地址（或者通过arp -a 查看网关的mac地址）。将router1的dst mac设置为gate 的mac地址，然后 Router1通过校园网将以太网帧frame发送Router2（服务器）。  第一个问题。发现IPIP数据包无法发送到服务器上。推测原因是由于学校校园网网关不允许IPIP类型的数据包出去（只允许TCP，UDP，ICMP？），这导致无法将数据传送出去。这种IPIP隧道不方便穿越NAT，不同NAT设备的NAT实现也各不相同。为了解决这一问题，我们采用了另外一种IPIP的实现。  原来的 [IP[UDP]] 被封装为 [outer\_IP[IP[UDP]]]，现在我们可以封装为[outer\_IP[UDP [IP[UDP]]]]]，即基于UDP 的隧道，我们将其原始的IP数据包放进一个UDP的payload中，在Router2中，需要多解析一步，将UDP的payload拿出来，即是原先的IP数据包。使用UDP封装的还有一个好处是，可以使用不同的UDP端口来区分不同的用户隧道。  不过，为了模拟“IP in IP”的过程，我们最后选择了这种方式：封装为[outer\_IP[UDP [outer\_IP[IP[UDP]]] ]]，只是将最初的ipip数据包再装载进了一个udp中（帮助我们将数据包发送到服务器）。  第二个问题。我们之前Router2会判断接收到的数据包源IP是不是来自于Router1，如果是，我们才进行处理。但是由于校园网存在NAT地址转换协议，我们本机的IP经过校园网网关发送到服务器时，源IP被替换为了网关IP，因此无法支持之前的判断逻辑（但是可以通过判断网关ip来解决）。 （四）真实环境下的测试 流程如下：   * + - 1. 在local linux 中分别启动client1，router1。在remote linux 中分别启动client2，router2.       2. client1发送数据包（目的ip为client2的ip），经过模拟arp（指定router1的mac），该数据包发送到router1。       3. router1判断数据包来自自己的网络内部，并且目的ip在remote的网络中，则将ip-udp数据包加上outer\_ip头，然后将outer\_ip-ip-udp书包再装进outer\_outer\_ip-outer\_udp中，并指定outer\_outer\_ip和outer\_ip的目的ip为router2的公网ip，outer\_ip-protocol为IPIP协议，outer\_outer\_ip-protocol为udp协议。       4. router2接收到数据包，解析为udp后，判断udp的端口号，如果我们指定的9999，表示这里的payload是一个ipip数据包。此时将进行ipip的解封装。在解析时，发现ip的protocol为IPIP协议号，将外层的ip剥去，然后根据内层的目的ip地址（模拟arp，指定mac地址），发送给client2.       5. client2接收到数据包，解析为udp，打印输出数据。   image-20210613135447238  image-20210613135457316 | | | | | | |