**题目：网络协议栈设计与实现**

**姓名：王怡恒**

**学号：2022300385**

**班号：10012201**

**时间：2024/12/14**

**计算机学院**

**目 录**

[1、实验目的 3](#_Toc20169)

[2、实验要求 3](#_Toc2025)

[3、实现原理及实现流程 3](#_Toc8205)

[3.1 ICMP 协议 3](#_Toc24838)

[3.2 UDP 协议 4](#_Toc17954)

[4、算法设计 5](#_Toc4651)

[4.1 ICMP 协议 5](#_Toc8613)

[4.2 UDP 协议 7](#_Toc10082)

[5、运行结果与分析 13](#_Toc3291)

[5.1 ICMP 协议 13](#_Toc32468)

[5.2 UDP 协议 14](#_Toc28269)

[6、实验过程发现的新技术问题及改进方向 15](#_Toc2307)

[6.1 技术问题 15](#_Toc3925)

[6.2 改进方向 16](#_Toc916)

[7、参考文献 16](#_Toc13543)

**1、实验目的**

1. 深入理解 ICMP 和 UDP 协议的工作原理，包括协议格式、数据传输过程和通信机制。
2. 掌握在网络协议栈中实现 ICMP 和 UDP 协议模块的方法，能够编写相应的代码来处理协议相关操作。
3. 通过实验实现，增强对网络分层结构的理解，明确 ICMP 和 UDP 协议在网络通信中的作用和地位。

**2、实验要求**

1. 实现 ICMP 协议的部分功能，主要是接收端对 ICMP ECHO 请求报文的响应功能，能够正确构造并发送 ICMP ECHO 应答报文。
2. 完整实现 UDP 协议模块，包括创建套接字、绑定端口、发送数据、接收数据以及释放资源等功能。
3. 确保实现的协议模块能够正确处理各种边界情况，如数据校验、端口不可达等错误情况的处理。

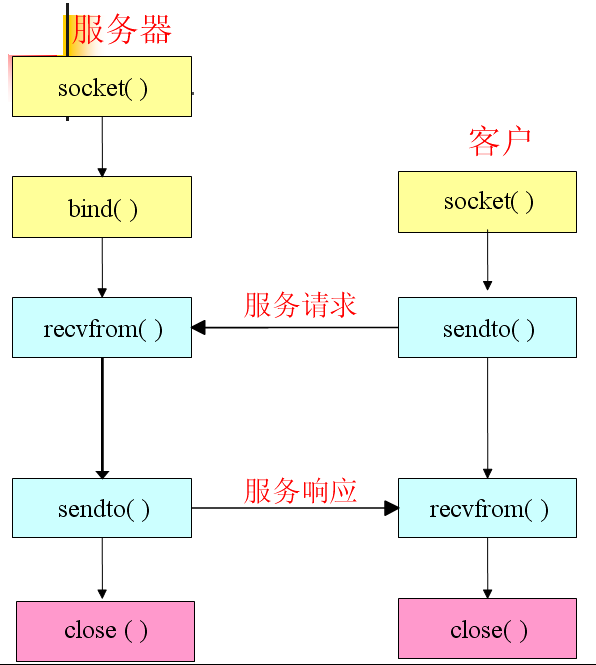
**3、实现原理及实现流程**

**3.1 ICMP 协议**

1. **实现原理**
   * 当 IP 协议接收到 IP 分组后，会检查分组的协议字段。如果协议字段为 ICMP 协议类型，则将该分组交给 ICMP 接收函数 Icmp\_recv 进行处理。
   * 在 Icmp\_recv 函数中，首先判断接收到的 ICMP 报文是否为 ICMP ECHO 请求报文（类型为 8）。如果是，则构造一个 ICMP ECHO 应答报文（类型为 0），并设置应答报文的标识符、序列号与请求报文相同，计算校验和后，将应答报文交给 IP 协议进行封装和发送。如果不是 ECHO 请求报文，则不做处理。
2. **实现流程**
   * 开始接收 IP 分组。
   * 判断协议字段是否为 ICMP 协议。如果否，结束处理；如果是，进入下一步。
   * 调用 Icmp\_recv 函数处理 ICMP 报文。
   * 在 Icmp\_recv 函数中，判断 ICMP 类型是否为 ECHO 请求。如果否，返回；如果是，构造 ICMP ECHO 应答报文。
   * 计算应答报文的校验和，将应答报文交给 IP 协议封装为 IP 分组并发送。

**3.2 UDP 协议**

1. **实现原理**
   * UDP 协议采用无连接的数据报通信方式。通过创建套接字 socket 函数来初始化通信五元组，绑定 bind 函数将服务器的 IP 地址和端口号与套接字关联，发送数据使用 sendto 函数明确指定接收方地址，接收数据使用 recvfrom 函数，并在接收过程中进行目的端口检查和校验和验证，最后使用 closesocket 函数释放资源。
2. **实现流程**



* + 服务器端和客户端创建套接字。
  + 服务器端绑定 IP 地址和端口号。
  + 客户端或服务器端发送数据，调用 sendto 函数构造 UDP 用户数据报并交付给 IP 协议发送。
  + 接收方接收数据，调用 recvfrom 函数处理接收到的 IP 分组，检查目的端口和校验和，将数据复制到缓冲区并返回接收数据大小。
  + 通信结束后，双方使用 closesocket 函数关闭套接字释放资源。

**4、算法设计**

**4.1 ICMP 协议**

1. **主要函数及功能**
   * icmp\_recv函数：
     + 接收 IP 缓冲区中的数据，解析出 ICMP 报文头。
     + 打印 ICMP 报文的类型、代码、校验和、标识符和序列号等信息。
     + 判断 ICMP 类型是否为 ECHO 请求（类型为 8），若是则调用 send\_icmp\_echo\_reply 函数发送应答报文。
   * send\_icmp\_echo\_reply函数：
     + 创建 ICMP Echo 应答报文，设置类型为 0，代码为 0，标识符和序列号与请求报文相同。
     + 计算应答报文的校验和。
     + 封装应答报文到 IP 数据报中，交换源 IP 和目的 IP 地址，设置 IP 标识符、TTL 和上层协议类型为 ICMP。
     + 调用 ethernet\_send\_packet 函数发送 IP 数据报。
2. **代码解析**
3. 在 icmp\_recv 函数中：

void icmp\_recv(u\_int8\_t \*ip\_buffer) {
  
 **struct** ip\_header \*ip\_hdr = (**struct** ip\_header \*)ip\_buffer;
  
 **struct** icmp\_header \*icmp\_hdr = (**struct** icmp\_header \*)(ip\_buffer + **sizeof**(**struct** ip\_header));
  
 *// 打印ICMP报文信息*  
 printf("--------------ICMP Protocol-------------------\n");
  
 printf("ICMP Type: %d\n", icmp\_hdr->type);
  
 printf("ICMP Code: %d\n", icmp\_hdr->code);
  
 printf("ICMP Checksum: %04x\n", icmp\_hdr->checksum);
  
 printf("ICMP ID: %d\n", ntohs(icmp\_hdr->id));
  
 printf("ICMP Sequence: %d\n", ntohs(icmp\_hdr->sequence));
  
 *// 检查ICMP类型是否为Echo请求*  
 **if** (icmp\_hdr->type == ICMP\_ECHO\_REQUEST) {
  
 printf("Received ICMP Echo Request\n");
  
 *// 发送ICMP Echo应答*  
 send\_icmp\_echo\_reply(ip\_buffer, ntohs(ip\_hdr->total\_length), icmp\_hdr);
  
 }
  
 printf("-----------------End of ICMP Protocol---------------\n");
  
}

* 首先从 IP 缓冲区中获取 IP 头和 ICMP 头指针，然后打印 ICMP 报文的各项信息。接着判断 ICMP 类型，如果是 ECHO 请求，则调用 send\_icmp\_echo\_reply 函数构造并发送应答报文。

1. 在send\_icmp\_echo\_reply 函数中：

void send\_icmp\_echo\_reply(u\_int8\_t \*ip\_buffer, int ip\_len, **struct** icmp\_header \*icmp\_request) {  
 *// 创建ICMP Echo应答报文*  
 **struct** icmp\_header icmp\_response;  
 icmp\_response.type = ICMP\_ECHO\_REPLY;  
 icmp\_response.code = 0;  
 icmp\_response.id = icmp\_request->id;  
 icmp\_response.sequence = icmp\_request->sequence;  
 *// 计算校验和*  
 icmp\_response.checksum = 0;  
 u\_int16\_t checksum = calculate\_checksum(&icmp\_response, **sizeof**(**struct** icmp\_header));  
 icmp\_response.checksum = checksum;  
 *// 将ICMP响应封装到IP数据报中*  
 **struct** ip\_header \*ip\_hdr = (**struct** ip\_header \*)ip\_buffer;  
 ip\_hdr->id = htons(ip\_id++);  
 ip\_hdr->time\_to\_live = 64;  
 ip\_hdr->upper\_protocol\_type = IPPROTO\_ICMP;  
 *// 交换源IP和目的IP*  
 u\_int8\_t temp\_ip[4];  
 memcpy(temp\_ip, ip\_hdr->source\_ip, 4);  
 memcpy(ip\_hdr->source\_ip, ip\_hdr->destination\_ip, 4);  
 memcpy(ip\_hdr->destination\_ip, temp\_ip, 4);  
 *// 发送IP数据报*  
 ethernet\_send\_packet((u\_int8\_t \*)&icmp\_response, ip\_buffer + **sizeof**(**struct** ip\_header), **sizeof**(**struct** icmp\_header));  
}

* 该函数首先构造 ICMP Echo 应答报文，设置类型、代码、标识符和序列号。然后计算校验和，将应答报文封装到 IP 数据报中，交换源 IP 和目的 IP 地址，设置 IP 标识符、TTL 和上层协议类型，最后调用 ethernet\_send\_packet 函数发送 IP 数据报。

**4.2 UDP 协议**

1. **主要函数及功能**
   * create\_socket函数：
     + 创建 UDP 套接字，指定地址族为 AF\_INET，通信协议类型为 SOCK\_DGRAM，网络通信协议为 IPPROTO\_UDP。
     + 如果创建失败，打印错误信息并返回 -1；成功则返回套接字描述符。
   * bind\_socket函数：
     + 将服务器的 IP 地址和端口号与套接字绑定。
     + 填充 sockaddr\_in 结构体，设置地址族、端口号和 IP 地址。
     + 调用 bind 函数进行绑定，如果失败，打印错误信息并返回 -1；成功则返回 0。
   * sendto\_socket函数：
     + 发送 UDP 数据报，指定套接字描述符、数据缓冲区、数据长度、目的地址和地址长度。
     + 打印发送相关信息，如套接字描述符和数据长度等。
     + 如果发送失败，打印错误信息并返回 -1；成功则返回发送的数据字节数。
   * recvfrom\_socket函数：
     + 接收 UDP 数据报，指定套接字描述符、数据缓冲区、缓冲区长度、源地址和地址长度指针。
     + 如果接收失败，打印错误信息并返回 -1；成功则返回接收的数据字节数。
   * close\_socket函数：
     + 关闭套接字，释放资源。
     + 调用 closesocket 函数关闭套接字，如果成功返回 1，失败返回 -1。
   * transport\_udp\_send函数：
     + 从文件中读取数据，创建 UDP 套接字，组装目的地址，发送数据并接收回应。
     + 打开文件，获取文件大小，分配缓冲区并读取文件内容。
     + 创建套接字，设置目的地址信息，调用 sendto\_socket 函数发送数据。
     + 接收回应数据，打印接收到的数据和 UDP 头部信息，最后清理资源，关闭套接字。
   * transport\_udp\_recv函数：
     + 接收 UDP 数据，创建套接字，绑定端口，接收数据并处理。
     + 创建套接字，绑定到指定端口（如 12345）。
     + 接收数据，将数据写入文件，打印接收到的数据和 UDP 头部信息，发送响应数据，最后关闭套接字。
2. **代码解析**
3. 在 create\_socket 函数中：

int create\_socket() {  
 int sockid = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);  
 **if** (sockid == INVALID\_SOCKET) {  
 printf("Failed to create socket. Error: %d\n", WSAGetLastError());  
 **return** -1;  
 }  
 **return** sockid;  
}

* 此函数使用 socket 系统调用创建 UDP 套接字，根据返回值判断创建是否成功，若失败则打印错误信息并返回 -1，成功则返回套接字描述符。

1. 在 bind\_socket 函数中：

int bind\_socket(int sockid, const char\* ip, int port) {  
 **struct** sockaddr\_in servaddr;  
 servaddr.sin\_family = AF\_INET;  
 servaddr.sin\_port = htons(port);  
 servaddr.sin\_addr.s\_addr = inet\_addr(ip);  
 int nResult = bind(sockid, (**struct** sockaddr\*)&servaddr, **sizeof**(servaddr));  
 **if** (nResult == SOCKET\_ERROR) {  
 printf("Failed to bind socket. Error: %d\n", WSAGetLastError());  
 **return** -1;  
 }  
 **return** 0;  
}

* 该函数用于将服务器的 IP 地址和端口号与套接字绑定。首先填充 sockaddr\_in 结构体，然后调用 bind 函数进行绑定操作，根据返回值判断绑定是否成功，失败则打印错误信息并返回 -1，成功返回 0。

1. 在 sendto\_socket 函数中：

int sendto\_socket(int sockid, const char\* buf, int buflen, const **struct** sockaddr\* destaddr, int addrlen) {  
 int sent = sendto(sockid, buf, buflen, 0, destaddr, addrlen);  
 **if** (sent == SOCKET\_ERROR) {  
 printf("Failed to send data. Error: %d\n", WSAGetLastError());  
 **return** -1;  
 }  
 **return** sent;  
}

* 此函数用于发送 UDP 数据报，调用 sendto 系统调用发送数据，根据返回值判断发送是否成功，失败则打印错误信息并返回 -1，成功返回发送的数据字节数。

1. 在 recvfrom\_socket 函数中：

int recvfrom\_socket(int sockid, char\* buf, int buflen, **struct** sockaddr\* srcaddr, int\* addrlen) {  
 int received = recvfrom(sockid, buf, buflen, 0, srcaddr, addrlen);  
 **if** (received == SOCKET\_ERROR) {  
 printf("Failed to receive data. Error: %d\n", WSAGetLastError());  
 **return** -1;  
 }  
 **return** received;  
}

* 该函数用于接收 UDP 数据报，调用 recvfrom 系统调用接收数据，根据返回值判断接收是否成功，失败则打印错误信息并返回 -1，成功返回接收的数据字节数。

1. 在 close\_socket 函数中：

int close\_socket(int sockid) {  
 **return** closesocket(sockid);  
}

* 此函数用于关闭套接字，调用 closesocket 函数释放资源，根据返回值判断关闭是否成功，成功返回 1，失败返回 -1。

1. 在 transport\_udp\_send 函数中：

void transport\_udp\_send(u\_int8\_t \*ip\_buffer, u\_int8\_t protocol\_type) {  
 *// 从文件读取数据*  
 FILE \*file = fopen("data\_send.txt", "r");  
 **if** (file == NULL) {  
 printf("Failed to open file.\n");  
 **return**;  
 }  
 *// 获取文件大小*  
 fseek(file, 0, SEEK\_END);  
 long file\_size = ftell(file);  
 fseek(file, 0, SEEK\_SET);  
 *// 分配缓冲区并读取文件内容*  
 char \*data = (char \*)malloc(file\_size + 1);  
 **if** (data == NULL) {  
 printf("Failed to allocate memory.\n");  
 fclose(file);  
 **return**;  
 }  
 fread(data, 1, file\_size, file);  
 data[file\_size] = '\0';  
 fclose(file);  
 *// 创建套接字*  
 int sockid = create\_socket();  
 **if** (sockid == -1) {  
 printf("Failed to create socket.\n");  
 free(data);  
 **return**;  
 }  
 *// 组装目的地地址*  
 **struct** sockaddr\_in dest\_addr;  
 dest\_addr.sin\_family = AF\_INET;  
 dest\_addr.sin\_port = htons(12345);  
 dest\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");  
 *// 调用sendto\_socket函数发送数据*  
 int sent = sendto\_socket(sockid, data, file\_size, (**struct** sockaddr\*)&dest\_addr, **sizeof**(dest\_addr));  
 **if** (sent == -1) {  
 printf("Failed to send data.\n");  
 } **else** {  
 printf("Data sent successfully. Sent %d bytes.\n", sent);  
 }  
 *// 接收回应*  
 char recv\_buf[1024];  
 **struct** sockaddr srcaddr;  
 int addrlen = **sizeof**(srcaddr);  
 *// 接收UDP数据*  
 int received = recvfrom\_socket(sockid, recv\_buf, **sizeof**(recv\_buf), (**struct** sockaddr\*)&srcaddr, &addrlen);  
 **if** (received == -1) {  
 printf("Failed to receive response.\n");  
 } **else** {  
 *// 打印接收到的数据*  
 printf("Received %d bytes: %s\n", received, recv\_buf);  
 *// 打印UDP头部信息*  
 **struct** udphdr \*udp\_hdr = (**struct** udphdr \*)(recv\_buf);  
 printf("--------------UDP Protocol-------------------\n");  
 printf("UDP Source Port: %d\n", ntohs(udp\_hdr->uh\_sport));  
 *// printf("UDP Source Port\_initial: %d\n", udp\_hdr->uh\_sport);*  
 printf("UDP Destination Port: %d\n", ntohs(udp\_hdr->uh\_dport));  
 *// printf("UDP Destination Port\_initial: %d\n", udp\_hdr->uh\_dport);*  
 printf("UDP Length: %d\n", ntohs(udp\_hdr->uh\_ulen));  
 printf("UDP Checksum: %04x\n", ntohs(udp\_hdr->uh\_sum));  
 printf("-----------------End of UDP Protocol---------------\n");  
 }  
 *// 清理*  
 free(data);  
 closesocket(sockid);  
}

* 这个函数实现了 UDP 数据的发送和接收过程。首先从文件中读取数据，创建套接字，组装目的地址，然后发送数据。发送成功后，等待接收回应数据，接收到数据后打印数据内容和 UDP 头部信息，最后释放资源，关闭套接字。

1. 在 transport\_udp\_recv 函数中：

void transport\_udp\_recv() {  
 *// 创建套接字*  
 int sockid = create\_socket();  
 **if** (sockid == -1) {  
 printf("Failed to create socket.\n");  
 **return**;  
 }  
 *// 绑定套接字*  
 **if** (bind\_socket(sockid, "0.0.0.0", 12345)!= 0) {  
 printf("Failed to bind socket.\n");  
 close\_socket(sockid);  
 **return**;  
 }  
 *// 准备接收缓冲区和地址结构*  
 char recv\_buf[1024];  
 **struct** sockaddr srcaddr;  
 int addrlen = **sizeof**(srcaddr);  
 *// 接收UDP数据*  
 printf("begin to recv\n");  
 int received = recvfrom\_socket(sockid, recv\_buf, **sizeof**(recv\_buf), &srcaddr, &addrlen);  
 **if** (received == -1) {  
 printf("Failed to receive data.\n");  
 close\_socket(sockid);  
 **return**;  
 }  
 *// 处理接收到的数据*  
 recv\_buf[received] = '\0';  
 printf("Received %d bytes: %s\n", received, recv\_buf);  
 *// 将接收到的数据写入文件*  
 FILE \*file = fopen("data\_recv.txt", "w");  
 **if** (file == NULL) {  
 printf("Failed to open file for writing.\n");  
 close\_socket(sockid);  
 **return**;  
 }  
 fwrite(recv\_buf, 1, received, file);  
 fclose(file);  
 printf("Data written to file successfully.\n");  
 *// 打印UDP头部信息*  
 **struct** udphdr \*udp\_hdr = (**struct** udphdr \*)recv\_buf;  
 printf("--------------UDP Protocol-------------------\n");  
 printf("UDP Source Port: %d\n", ntohs(udp\_hdr->uh\_sport));  
 printf("UDP Destination Port: %d\n", ntohs(udp\_hdr->uh\_dport));  
 printf("UDP Length: %d\n", ntohs(udp\_hdr->uh\_ulen));  
 printf("UDP Checksum: %04x\n", ntohs(udp\_hdr->uh\_sum));  
 printf("-----------------End of UDP Protocol---------------\n");  
 *// 发送响应数据*  
 const char\* response = "Response from server";  
 **if** (sendto\_socket(sockid, response, strlen(response) + 1, &srcaddr, addrlen) == -1) {  
 printf("Failed to send response.\n");  
 }  
 printf("Success to send udp response.\n");  
 *// 关闭套接字*  
 close\_socket(sockid);  
}

* 该函数首先调用 create\_socket 函数创建 UDP 套接字，如果创建失败则打印错误信息并返回。接着调用 bind\_socket 函数将套接字绑定到本地地址（这里使用 "0.0.0.0" 表示所有可用网络接口）和指定端口（12345），绑定失败则进行相应处理。然后准备接收缓冲区和源地址结构，调用 recvfrom\_socket 函数接收 UDP 数据。接收成功后，将数据以空字符结尾，打印接收到的数据长度和内容，并将数据写入文件。之后，从接收缓冲区中解析出 UDP 头部信息并打印，包括源端口、目的端口、长度和校验和等。最后，构造响应数据并调用 sendto\_socket 函数发送响应，发送成功后打印提示信息，最后关闭套接字释放资源。这个函数完整地实现了 UDP 服务器端接收数据、处理数据、发送响应并关闭资源的流程。

**5、运行结果与分析**

**5.1 ICMP 协议**

1. **测试环境与方法**
   * 在本地网络环境中，使用两台主机进行测试。一台主机作为发送端，发送 ICMP ECHO 请求报文；另一台主机作为接收端，运行实现的 ICMP 协议模块来接收和处理请求报文。
2. **运行结果**

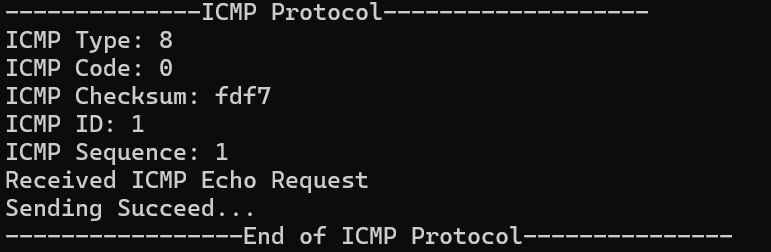


图 1 接收端

* + 发送端发送 ICMP ECHO 请求报文后，接收端成功接收到报文，并判断为 ECHO 请求。然后接收端构造并发送 ICMP ECHO 应答报文，发送端成功接收到应答报文。
  + 在接收端控制台输出了 ICMP 报文的详细信息，包括类型、代码、校验和、标识符和序列号等，验证了数据的正确性。

1. **结果分析**
   * 表明 ICMP 协议模块的接收和响应功能正常工作，能够正确识别 ECHO 请求并发送相应的应答报文，数据传输准确无误，校验和计算正确，保证了报文的完整性。

**5.2 UDP 协议**

1. **测试环境与方法**
   * 同样在本地网络环境中，分别运行 UDP 协议的服务器端和客户端程序。服务器端绑定特定端口，等待接收数据；客户端向服务器端发送数据，并接收服务器端的响应。
2. **运行结果**

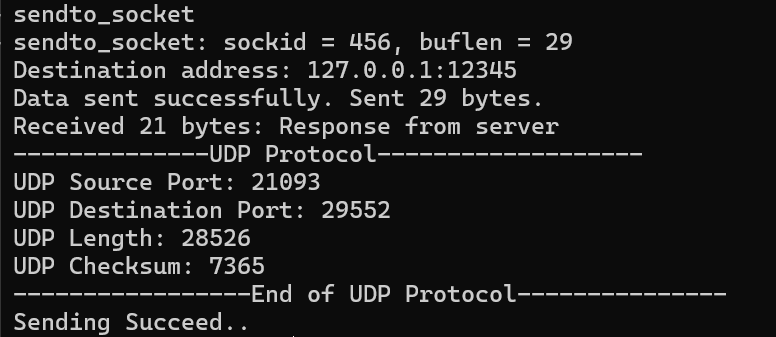


图 2 发送端接收到接收端的回应

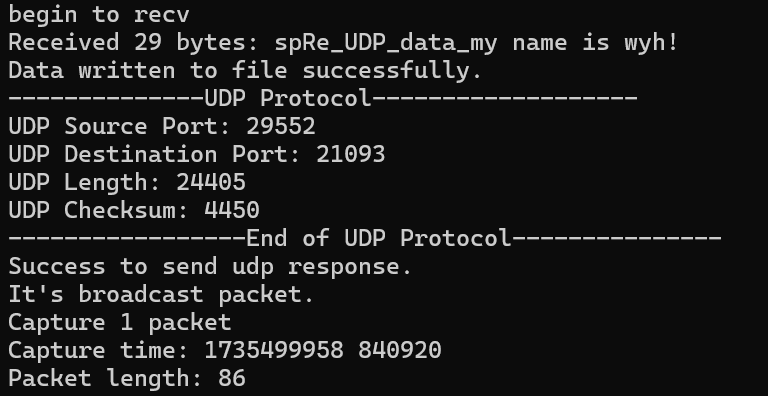


图 3 接收端接收发送端UDP，并发送UDP回应

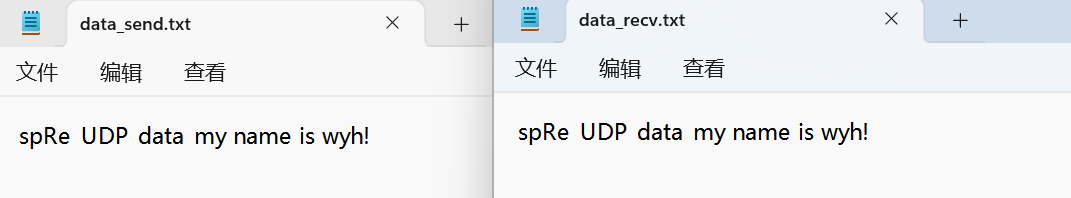


图 4 发送方和接收方txt文件一样，说明成功

* + 客户端成功发送数据，服务器端接收到数据并将其写入文件，同时打印出接收到的数据和 UDP 头部信息。服务器端发送响应数据，客户端成功接收到响应并打印相关信息。
  + 在发送和接收过程中，打印了套接字操作的详细信息，如创建套接字、绑定端口、发送和接收数据的字节数等，便于跟踪程序执行过程。

1. **结果分析**
   * UDP 协议模块的各个功能均正常运行，数据能够准确地在客户端和服务器端之间传输，端口绑定正确，数据的发送和接收符合预期，校验和验证等机制有效地保证了数据的正确性和完整性。

**6、实验过程发现的新技术问题及改进方向**

**6.1 技术问题**

1. 在计算校验和时，对于字节序的处理需要特别小心，否则可能导致校验和计算错误，影响报文的正确性和传输。
2. 处理边界情况时，如接收数据缓冲区大小的设置、文件读取和写入的边界处理等，需要仔细考虑，避免数据溢出或丢失。
3. 在网络通信中，不同主机的字节序可能不同，需要进行字节序转换，否则会导致数据解析错误。

**6.2 改进方向**

1. 在进行校验和计算等涉及字节操作的地方，增加详细的注释，解释字节序处理的逻辑，以便于代码的维护和理解。
2. 对于缓冲区大小和文件操作边界情况，可以进行更多的测试用例设计，覆盖各种可能的边界条件，确保程序的稳定性和可靠性。
3. 在代码中统一处理字节序转换问题，确保在不同主机之间通信时数据的正确解析，可以使用网络字节序转换函数（如 htons、ntohs 等）来保证数据的一致性。

**7、参考文献**

[1] 第四次课程-试点班.doc  
[2] 网络协议相关的标准文档，如 RFC 文档中关于 ICMP 和 UDP 协议的详细规范。