



南开大学
Nankai University

计算机学院
计算机网络

基于 **UDP** 服务
设计可靠传输协议
并编程实现 **3-1**

专业：计算机科学与技术

学号：**2212032**

姓名：徐亚民

日期：**2024 年 11 月 30 日**

目录

一、 实验要求	1
二、 协议设计	1
2.1 数据包格式 (Packet 结构体)	1
2.2 校验和计算	2
2.3 发送端与接收端交互	2
2.4 连接管理 (SYN, ACK, FIN)	3
2.5 错误处理与重传	3
2.6 文件传输流程	4
2.7 总结	4
三、 代码分析	4
3.1 共享: 数据包结构体	4
3.2 共享: 计算校验和	5
3.3 共享: 发送数据包	8
3.4 共享: 接收数据包	8
3.5 发送文件	10
3.6 发送端主函数	13
3.7 处理数据包	16
3.8 接收端主函数	18
四、 编译指令	19
4.1 指令代码	19
4.2 指令含义	20
五、 运行截图	21
5.1 三次握手 (Connection Establishment)	21
5.2 文件传输 (File Transmission)	21
5.3 超时重传 (Timeout Retransmission)	22
5.4 四次挥手 (Connection Termination)	22
六、 结果分析	24
七、 实验总结	25

一、 实验要求

1. 实现单向传输。
2. 对于每个任务要求给出详细的协议设计。
3. 给出实现的拥塞控制算法的原理说明。
4. 完成给定测试文件的传输，显示传输时间和平均吞吐率。
5. 性能测试指标：吞吐率、文件传输时延，给出图形结果并进行分析。
6. 完成详细的实验报告。
7. 编写的程序应该结构清晰，具有较好的可读性。
8. 现场演示。
9. 提交程序源码、可执行文件和实验报告。

二、 协议设计

2.1 数据包格式 (Packet 结构体)

协议中的数据包由以下字段组成：

- `seq_num` (int): 发送序号，表示数据包的序列号，用于确保数据包按顺序传输。
- `ack_num` (int): 确认序号，表示接收端期望接收到的下一个序列号。发送端通过此字段确认接收到的包。
- `name` [`MAX_FILENAME_SIZE`] (char): 文件名称，发送数据包时携带文件的名称。
- `data` [`MAX_DATA_SIZE`] (char): 发送的数据内容，最大字节数由 `MAX_DATA_SIZE` 决定。
- `data_len` (int): 数据内容的长度，表示 `data` 字段中有效数据的字节数。
- `check_sum` (int): 校验和，使用 16 位加法校验机制确保数据的完整性。
- `ACK` (bool): 确认标志位，表示是否是一个 ACK（确认）包。
- `SYN` (bool): 同步标志位，表示是否是一个 SYN（同步）包，用于建立连接。

- FIN (bool): 终止标志位，表示是否是一个 FIN（终止）包，用于断开连接。

2.2 校验和计算

校验和是对数据的完整性检查，采用 16 位加法的方式：

`calculate_checksum` 函数计算传输数据的校验和，确保数据在传输过程中没有被篡改。

校验和计算基于数据内容的 16 位加法，若数据长度为奇数，还会额外处理最后一个字节。最终，返回的是校验和的反码。

2.3 发送端与接收端交互

2.3.1 数据包传输（`send_packet` 和 `receive_packet`）

- `send_packet`: 用于将数据包通过 UDP 协议发送到接收端。在发送之前，会计算数据的校验和，并填充数据包的相应字段。发送后，打印相关日志信息。
- `receive_packet`: 接收数据包并进行校验和检查。如果数据包的校验和不正确，则丢弃该数据包。校验成功后，提取数据包中的信息并打印日志。

2.3.2 发送端（`send_file`）

- `send_file`: 用于文件传输的主函数，逐块读取文件并构造数据包，然后调用 `send_packet` 进行发送。每发送一个数据包后，会等待接收端的 ACK 确认包。如果接收确认失败（如超时），会进行重传。
 - 打开文件并获取文件大小。
 - 按块读取文件数据，并构造数据包。
 - 发送数据包后，等待接收 ACK 包。如果 ACK 包没有收到，进行超时重传。
 - 记录传输时间，计算吞吐率，并输出传输的统计信息。

2.3.3 接收端（`handle_packet`）

- `handle_packet`: 接收数据包并根据不同的标志位（SYN、ACK、FIN）进行处理：

- SYN 包：用于三次握手的初始化，接收到 SYN 包后，发送 SYN-ACK 包表示响应。
- ACK 包：用于连接的确认和数据传输的确认，接收到 ACK 包后，更新确认序号并准备接收下一条数据。
- FIN 包：用于连接终止的四次挥手，接收到 FIN 包后，发送 ACK 包并等待 FIN-ACK 包。

处理接收到的数据时，接收端会将数据写入文件并发送 ACK 确认数据包。接收端按顺序处理数据包，确保每个数据包都得到确认。

2.4 连接管理 (SYN, ACK, FIN)

2.4.1 三次握手 (Connection Establishment)

发送端和接收端通过 SYN 和 ACK 标志位进行三次握手以建立连接：

- 发送端发送一个 SYN 包，表示请求建立连接。
- 接收端收到 SYN 包后，回复 SYN-ACK 包，表示同意建立连接。
- 发送端收到 SYN-ACK 包后，发送 ACK 包，确认连接已建立。

2.4.2 四次挥手 (Connection Termination)

在文件传输完成后，发送端和接收端通过 FIN 标志位进行四次挥手以终止连接：

- 发送端发送 FIN 包，表示请求关闭连接。
- 接收端收到 FIN 包后，发送 ACK 包，确认关闭请求。
- 接收端发送 FIN 包，表示同意关闭连接。
- 发送端收到 FIN-ACK 包后，发送 ACK 包，确认关闭连接。

2.5 错误处理与重传

- 超时重传：发送端在发送数据包后等待接收端的 ACK 确认包。如果在指定时间内未收到 ACK 包，则会进行重传。重传次数通过 `timeout` 变量进行统计。

- 校验和验证：接收端接收到的数据包会首先进行校验和验证，若验证失败，则丢弃该数据包并不做任何处理。

2.6 文件传输流程

- 发送端首先通过三次握手建立连接。
- 然后，发送端逐块读取文件内容，每块数据通过 `send_packet` 发送给接收端。
- 每发送一个数据包后，发送端会等待接收端的 ACK 确认。如果没有收到确认，发送端会重发数据包。
- 文件传输完成后，发送端通过四次挥手过程断开连接。

2.7 总结

该协议基于 UDP 实现可靠的文件传输，借助 SYN、ACK、FIN 等标志位实现了可靠的数据传输和连接管理。通过校验和确保数据完整性，使用超时重传机制确保可靠性，且通过三次握手和四次挥手管理连接的建立和关闭。

三、 代码分析

3.1 共享：数据包结构体

3.1.1 核心代码

```
1 struct Packet {  
2     int seq_num = 0;           // 发送序号  
3     int ack_num = 0;           // 确认序号  
4     char name[MAX_FILENAME_SIZE]; // 文件名称  
5     char data[MAX_DATA_SIZE];    // 发送数据  
6     int data_len = 0;            // 数据长度  
7     int check_sum = 0;          // 校验和  
8     bool ACK = false;           // 标志位  
9     bool SYN = false;           // 标志位  
10    bool FIN = false;           // 标志位  
11 };
```

3.1.2 代码分析

- **seq_num**: 发送序号, 表示数据包的顺序编号。用于保证数据包按顺序传输, 接收方可通过序号确认接收到的包是否按顺序排列。
- **ack_num**: 确认序号, 表示接收到的数据包的确认编号。在可靠协议中, 接收方会返回一个确认序号, 告知发送方已成功接收到哪个序号的数据包。
- **name**: 文件名称, 用于存储文件名。发送方将文件名通过数据包传输给接收方。
- **data**: 发送的数据, 存储实际传输的内容。它是一个字符数组, 用于存储文件或消息数据的内容。
- **data_len**: 数据长度, 表示数据字段中存储的实际数据的大小, 通常与 `MAX_DATA_SIZE` 相关, 指明当前数据包有效数据的大小。
- **check_sum**: 校验和, 用于确保数据在传输过程中的完整性。发送方计算数据包的校验和并附加在包中, 接收方在接收数据包后会进行校验, 确保数据未被损坏。
- **ACK**: 确认标志位, 指示该数据包是否为确认包。如果为 `true`, 表示这是一个确认包, 用于告知发送方接收方已成功接收数据。
- **SYN**: 同步标志位, 通常用于连接的初始化。在 TCP 协议的三次握手中, `SYN` 位为 `true` 时, 表示这是一个连接请求包。
- **FIN**: 终止标志位, 表示连接的终止。如果 `FIN` 位为 `true`, 则表示这是一个连接终止请求包, 告知接收方发送方准备断开连接。

3.2 共享：计算校验和

3.2.1 核心代码

```
1 unsigned short calculate_checksum(char* data) {
2     unsigned short len = strlen(data);
3     unsigned short checksum = 0;
4     unsigned short* ptr = (unsigned short*)data; // 将数据视为16位单位的数组
5
6     // 对数据进行16位加法求和
7     for (int i = 0; i < len / 2; i++) {
8         checksum += ptr[i];
9
10        // 如果有进位, 进行回卷
11        if (checksum > 0xFFFF) {
12            checksum = (checksum & 0xFFFF) + 1;
13        }
14    }
```

```
15
16 // 如果数据长度是奇数，处理剩余的最后一个字节
17 if (len % 2 != 0) {
18     checksum += (unsigned short)(data[len - 1] << 8);
19
20     // 如果有进位，进行回卷
21     if (checksum > 0xFFFF) {
22         checksum = (checksum & 0xFFFF) + 1;
23     }
24 }
25
26 // 返回反码
27 return ~checksum;
28 }
```

3.2.2 代码分析

该代码段实现了一个计算数据校验和的功能。具体来说，它使用的是 16 位的校验和算法，通常用于网络协议（如 TCP/IP）和一些数据传输协议中的错误检测。该校验和算法的核心思想是将数据按 16 位拆分，并对每个 16 位的数据进行加法求和，同时处理进位，并最终返回校验和的反码。以下是对代码的详细分析：

1. 函数声明和参数说明：

```
1 unsigned short calculate_checksum(char* data)
```

- 该函数接受一个指向字符数组 data 的指针，返回一个无符号的 16 位整数 checksum。
- data 是待计算校验和的字节数据。

2. 初始化：

```
1 unsigned short len = strlen(data);
2 unsigned short checksum = 0;
3 unsigned short* ptr = (unsigned short*)data;
```

- len 变量存储了 data 字符串的长度。strlen(data) 返回字符串的字符数，不包括末尾的空字符。
- checksum 变量用于保存计算后的校验和，初始值为 0。
- ptr 是一个 unsigned short* 指针，它将 data 指针转换为指向 16 位无符号整数的指针。这是为了将数据视为由 16 位数据单元组成。

3. 16 位加法求和：

```
1 for (int i = 0; i < len / 2; i++) {
2     checksum += ptr[i];
```



```
3
4 // 如果有进位, 进行回卷
5 if (checksum > 0xFFFF) {
6     checksum = (checksum & 0xFFFF) + 1;
7 }
8 }
```

- 这部分循环处理数据中的每个 16 位单元。由于每个 `unsigned short` 类型占 2 字节, `len / 2` 确定了数据可以被分成多少个 16 位单元。

- `ptr[i]` 指向当前 16 位数据单元, `checksum` 累加这些 16 位数据的值。

- 如果 `checksum` 超过了 16 位 (即大于 `0xFFFF`), 则通过回卷操作 (`checksum & 0xFFFF`) + 1 来处理进位。这个操作确保了校验和始终保持在 16 位内。

4. 处理奇数长度数据的剩余字节:

```
1 if (len % 2 != 0) {
2     checksum += (unsigned short)(data[len - 1] << 8);
3
4     // 如果有进位, 进行回卷
5     if (checksum > 0xFFFF) {
6         checksum = (checksum & 0xFFFF) + 1;
7     }
8 }
```

- 如果数据长度是奇数 (`len % 2 != 0`), 说明最后一个字节没有配对的 16 位数据。该字节需要以 16 位的形式参与计算。

- `data[len - 1]` 取最后一个字节, 左移 8 位使其成为 16 位数据 (高字节), 并将其加入 `checksum`。

- 如果在加法后出现进位, 进行回卷操作, 保证 `checksum` 仍然是 16 位。

5. 返回反码:

```
1 return ~checksum;
```

- 校验和的最终值是 `checksum` 的反码。取反操作 `checksum` 是为了满足一些协议的要求, 如 TCP/IP 协议中的反码校验。

- 反码校验是一种将所有位反转的操作, 通常用于确保接收方能更容易地检测到数据是否被修改。

3.3 共享：发送数据包

3.3.1 核心代码

```
1 int send_packet(SOCKET& sock, struct sockaddr_in& receiver_addr, Packet& pkt) {
2     // 计算校验和
3     pkt.check_sum = calculate_checksum(pkt.data);
4
5     // 发送数据包
6     int sent_len = sendto(sock, (char*)&pkt, sizeof(pkt), 0, (struct sockaddr*)&
7         receiver_addr, sizeof(receiver_addr));
8
9     if (sent_len == SOCKET_ERROR) {
10         cerr << "发送数据包失败" << endl;
11         return -1;
12     }
13     else {
14         cout << "发送数据包, 发送序号: " << pkt.seq_num << ", 确认序号: " << pkt.
15             ack_num
16             << ", 数据大小: " << pkt.data_len << ", 校验和: " << pkt.check_sum
17             << ", ACK: " << pkt.ACK << ", SYN: " << pkt.SYN << ", FIN: " << pkt.FIN <<
18             endl;
19         return 0;
20     }
21 }
```

3.3.2 代码分析

- 在发送数据包的函数‘send_packet’中，首先计算数据包的校验和，通过调用‘calculate_checksum’函数对数据部分进行校验和的计算，确保数据在传输过程中的完整性。接着，使用‘sendto’函数将数据包发送到指定的目标地址。若发送失败，则输出错误信息。
- 如果数据包成功发送，则输出包的各项信息，如发送序号、确认序号、数据长度、校验和以及是否设置了ACK、SYN、FIN等标志位，便于调试和数据包追踪。
- 该函数通过检查发送返回值，判断数据包是否成功发送，并进行日志打印。

3.4 共享：接收数据包

3.4.1 核心代码

```
1 int receive_packet(SOCKET& sock, struct sockaddr_in& sender_addr, Packet& pkt_received)
2 {
3     int len = sizeof(sender_addr);
4     char buffer[MAX_PACKET_SIZE];
```

```
4      int recv_len = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr*)&
5          sender_addr, &len);
6
7      if (recv_len == SOCKET_ERROR) {
8          cerr << "接收数据包失败" << endl;
9          return -1;
10     }
11
12     // 提取数据包内容
13     memcpy(&pkt_received, buffer, sizeof(pkt_received));
14
15     // 计算校验和
16     unsigned short calculated_checksum = calculate_checksum(pkt_received.data);
17
18     // 校验和出错
19     if (pkt_received.check_sum != calculated_checksum) {
20         cerr << "接收校验和: " << pkt_received.check_sum << ", 计算校验和: " <<
21             calculated_checksum << ", 校验和错误, 丢弃数据包";
22         return -1;
23     }
24
25     // 检验ACK
26     if (pkt_received.ACK && pkt_received.ack_num != seq_num_share) {
27         cout << "确认序号不正确, 丢弃数据包" << endl;
28         return -1;
29     }
30
31     cout << "接收数据包, 发送序号: " << pkt_received.seq_num << ", 确认序号: " <<
32         pkt_received.ack_num
33         << ", 数据大小: " << pkt_received.data_len << ", 校验和: " << pkt_received.
34         check_sum
35         << ", ACK: " << pkt_received.ACK << ", SYN: " << pkt_received.SYN << ", FIN: "
36         << pkt_received.FIN << endl;
37
38     return 0;
39 }
```

3.4.2 代码分析

- 接收数据包的函数‘receive_packet’通过调用‘recvfrom’从网络接收数据。如果接收过程中出现错误，返回‘-1’并输出错误信息。
- 接收到的数据存储在‘buffer’中，然后使用‘memcpy’将其转换为‘Packet’类型的数据结构。之后，函数计算接收到的数据的校验和，并与数据包中包含的校验和进行比较。如果校验和不匹配，则说明数据包在传输过程中出现了错误，丢弃该数据包并返回‘-1’。
- 如果校验和正确，则输出接收到的数据包的详细信息，包括发送序号、确认序号、数据大小、校验和、以及ACK、SYN、FIN等标志位。

- 函数返回 0 表示数据包成功接收并校验通过。

3.5 发送文件

3.5.1 核心代码

```
1  int send_file(SOCKET& sock, struct sockaddr_in& receiver_addr, string filename) {
2      // 超时重传次数
3      int timeout = 0;
4
5      // 记录文件的开始时间
6      auto start_time = chrono::high_resolution_clock::now();
7
8      // 打开文件
9      ifstream file("send/" + filename, ios::binary);
10     if (!file.is_open()) {
11         cerr << "打开文件失败" << endl;
12         return -1;
13     }
14
15     // 获取文件大小
16     file.seekg(0, ios::end); // 移动到文件末尾
17     long file_size = file.tellg(); // 获取文件大小
18     file.seekg(0, ios::beg); // 将文件指针重置到文件开头
19
20     // 设置接收超时时间，单位为毫秒
21     int timeout_ms = 1000; // 1秒超时
22     setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout_ms, sizeof(
        timeout_ms));
23
24     while (!file.eof()) {
25         Packet pkt;
26         // 设置文件名
27         strncpy_s(pkt.name, sizeof(pkt.name), filename.c_str(), sizeof(pkt.name) - 1);
28         pkt.name[sizeof(pkt.name) - 1] = '\0'; // 确保文件名以'\0'结尾
29
30         // 读取数据到数据包中
31         file.read(pkt.data, sizeof(pkt.data));
32         int read_len = file.gcount();
33         pkt.data_len = read_len;
34         pkt.seq_num = seq_num_share++;
35
36         // 发送数据包
37         send_packet(sock, receiver_addr, pkt);
38
39         // 等待ACK并处理超时重传
40         struct sockaddr_in sender_addr;
41         int sender_len = sizeof(sender_addr);
42         char buffer[MAX_PACKET_SIZE];
43         int ack_len = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr*)&
            sender_addr, &sender_len);
44
45         if (ack_len == SOCKET_ERROR) {
```

```

46     int err = WSAGetLastError();
47     if (err == WSAETIMEDOUT) {
48         // 如果超时, 打印重传信息
49         cerr << "接收ACK超时, 重新发送数据包: " << pkt.seq_num << endl;
50         timeout++;
51         seq_num_share--; // 回退序号, 重新发送当前数据包
52         file.seekg(-read_len, ios::cur); // 回退文件指针, 重新读取当前数据块
53         continue; // 超时重传
54     }
55     else {
56         cerr << "recvfrom 错误, 错误码: " << err << endl;
57         break;
58     }
59 }
60
61 // 解析收到的ACK包
62 Packet pkt_received;
63 memcpy(&pkt_received, buffer, sizeof(pkt_received));
64
65 // 输出收到的ACK信息
66 if (pkt_received.ACK && pkt_received.data_len == 0) {
67     cout << "接收数据包, 发送序号: " << pkt_received.seq_num << ", 确认序号: "
68         << pkt_received.ack_num
69         << ", 数据大小: " << pkt_received.data_len << ", 校验和: " <<
70         << pkt_received.check_sum
71         << ", ACK: " << pkt_received.ACK << ", SYN: " << pkt_received.SYN << "
72         << ", FIN: " << pkt_received.FIN << endl;
73 }
74
75 // 记录文件传输结束时间
76 auto end_time = chrono::high_resolution_clock::now();
77 chrono::duration<double> duration = end_time - start_time;
78
79 // 输出超时重传次数
80 cout << "超时重传次数为: " << timeout << " 次" << endl;
81
82 // 计算吞吐率 (文件大小 / 传输时间)
83 double throughput = (double)file_size / duration.count(); // 吞吐率, 单位字节/秒
84 cout << "文件传输时间: " << duration.count() << " 秒" << endl;
85 cout << "吞吐率: " << throughput / 1024 << " KB/s" << endl; // 吞吐率单位: KB/s
86
87 file.close();
88
89 return 0;
90 }

```

3.5.2 代码分析

- 函数参数:

- SOCKET& sock: 套接字, 用于与接收端进行通信。

- `struct sockaddr_in& receiver_addr`: 接收端的地址信息。
- `string filename`: 需要发送的文件名。

- **函数功能:**

- 打开文件并读取文件内容，以数据包的形式发送到接收端。
- 实现了超时重传机制，确保数据的可靠传输。
- 统计并输出超时重传次数、文件传输时间和吞吐率。

- **主要步骤分析:**

- * **文件打开和读取:** 打开指定路径的文件 (`send/ + filename`), 并使用二进制模式读取文件内容。文件大小通过 `seekg` 和 `tellg` 获取。
- * **数据包发送:** 文件内容被分割成多个数据包 (`Packet`), 每个数据包最多包含 `sizeof(pkt.data)` 字节的数据。发送过程中, 会不断读取文件数据, 并将其封装进数据包。每发送一个数据包, 都会调用 `send_packet` 函数向接收端发送数据。
- * **超时重传:** 数据包发送后, 会等待接收端的确认包 (`ACK`)。如果超时未收到 `ACK`, 则进行超时重传, 直到接收到确认包或达到重传次数的限制。
- * **ACK 处理:** 每次接收到的 `ACK` 数据包都会检查其确认号和其他字段, 以确认数据是否正确接收。收到的 `ACK` 数据包会打印相关的接收信息, 如确认序号、数据大小、校验和等。
- * **吞吐率计算:** 在文件发送完成后, 记录文件传输的起始和结束时间, 通过计算文件大小与传输时间, 得出吞吐率, 并以 `KB/s` 为单位输出。

- **关键点:**

- 通过使用 `chrono::high_resolution_clock` 来计算文件传输时间, 并且使用 `file.gcount()` 来获取每次读取的字节数, 确保数据包大小的正确性。
- `seq_num` 是每个数据包的序列号, 用于标识数据包的顺序, 确保接收端能正确组装文件。
- 通过超时计数器 `timeout` 来记录因超时而重新发送数据包的次数。

3.6 发送端主函数

3.6.1 核心代码

```
1 #pragma comment(lib, "ws2_32.lib")
2 int main() {
3     // 初始化WinSock
4     WSADATA wsaData;
5     if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
6         cerr << "WinSock 初始化失败" << endl;
7         return -1;
8     }
9
10    // 创建UDP套接字
11    SOCKET sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
12    if (sock == INVALID_SOCKET) {
13        cerr << "创建套接字失败" << endl;
14        return -1;
15    }
16
17    // 创建接收方地址
18    sockaddr_in receiver_addr;
19    receiver_addr.sin_family = AF_INET; // 设置地址族为IPv4
20    receiver_addr.sin_port = htons(SERVER_PORT); // 设置服务器端口号为20000
21    receiver_addr.sin_addr.s_addr = inet_addr(IPADDR); // 设置IP地址为127.0.0.1
22
23    // 创建本地客户端地址
24    sockaddr_in client_addr;
25    client_addr.sin_family = AF_INET; // 设置地址族为IPv4
26    client_addr.sin_port = htons(CLIENT_PORT); // 设置客户端端口号为10000
27    client_addr.sin_addr.s_addr = inet_addr(IPADDR); // 设置客户端IP地址为127.0.0.1
28
29    // 绑定客户端套接字到本地端口
30    if (bind(sock, (struct sockaddr*)&client_addr, sizeof(client_addr)) == SOCKET_ERROR) {
31        cerr << "绑定套接字失败" << endl;
32        return -1;
33    }
34
35    // 三次握手第一步：发送SYN包
36    Packet pkt0;
37    pkt0.SYN = true;
38    pkt0.seq_num = seq_num_share++;
39    // cout << "发送SYN包，请求建立连接" << endl;
40    send_packet(sock, receiver_addr, pkt0);
41
42    // 三次握手第二步：接收SYN-ACK包
43    Packet pkt_received0;
44    receive_packet(sock, receiver_addr, pkt_received0);
45
46    // 三次握手第三步：发送ACK包
47    if (pkt_received0.SYN && pkt_received0.ACK && pkt_received0.ack_num ==
48        seq_num_share) {
49        Packet pkt1;
50        pkt1.ACK = true;
```

```
50     pkt1.seq_num = seq_num_share++;
51     pkt1.ack_num = pkt_received0.seq_num + 1;
52     send_packet(sock, receiver_addr, pkt1);
53 }
54
55 // 创建文件名称
56 string filename;
57
58 // 循环发送文件
59 while (true) {
60     // 持续读取用户输入的文件名
61     cout << "请输入要发送的文件名（输入exit退出）： ";
62     getline(cin, filename);
63
64     if (filename == "exit")
65         break; // 用户输入exit时退出
66     else
67         send_file(sock, receiver_addr, filename); // 调用send_file发送文件
68 }
69
70 // 发送完文件后，开始四次挥手过程
71
72 // 四次挥手第一步：发送FIN包
73 Packet pkt2;
74 pkt2.FIN = true;
75 pkt2.seq_num = seq_num_share++;
76 send_packet(sock, receiver_addr, pkt2);
77
78 // 四次挥手第二步：接收ACK包
79 Packet pkt_received1;
80 receive_packet(sock, receiver_addr, pkt_received1);
81
82 // 四次挥手第三步：接收FIN-ACK包
83 Packet pkt_received2;
84 receive_packet(sock, receiver_addr, pkt_received2);
85
86 // 四次挥手第四步：发送ACK包
87 Packet pkt3;
88 pkt3.ACK = true;
89 pkt3.seq_num = seq_num_share++;
90 pkt3.ack_num = pkt_received2.seq_num + 1;
91 send_packet(sock, receiver_addr, pkt3);
92
93 closesocket(sock);
94 WSACleanup();
95
96 return 0;
97 }
```

3.6.2 代码分析

- 函数功能:

- 本函数负责初始化网络环境、创建套接字、并进行文件发送。
- 包括三次握手过程建立连接、文件传输、以及四次挥手过程断开连接。

• **步骤分析:**

- **WinSock 初始化:** 使用 `WSAStartup` 初始化 WinSock 库, 确保可以使用网络相关的 API。如果初始化失败, 程序会输出错误信息并返回。
- **创建套接字:** 创建一个 UDP 套接字用于数据的发送。通过 `socket` 函数创建套接字, 指定协议族为 IPv4 (`AF_INET`), 协议类型为 UDP (`SOCK_DGRAM`)。如果创建失败, 则输出错误并退出。
- **设置接收端地址:** 设置目标接收端的 IP 地址和端口。通过 `inet_addr` 设置 IP 地址为 `127.0.0.1`, 并通过 `htons` 设置端口号。
- **三次握手过程:**
 - * 发送 SYN 包 (第一步): 构建 SYN 包, 设置 SYN 标志位为 `true`, 并发送给接收端。
 - * 接收 SYN-ACK 包 (第二步): 等待接收端响应的 SYN-ACK 包, 并验证其确认号。
 - * 发送 ACK 包 (第三步): 根据接收到的 SYN-ACK 包, 发送确认的 ACK 包, 完成三次握手, 建立连接。
- **文件发送:** 在三次握手完成后, 进入文件传输循环, 用户输入文件名后调用 `send_file` 函数发送文件。如果用户输入 `exit`, 则跳出循环, 结束文件传输。
- **四次挥手过程:**
 - * 发送 FIN 包 (第一步): 发送 FIN 包, 表示文件发送完毕。
 - * 接收 ACK 包 (第二步): 等待接收端确认。
 - * 接收 FIN-ACK 包 (第三步): 接收端响应 FIN-ACK 包, 表示连接可以关闭。
 - * 发送 ACK 包 (第四步): 确认收到 FIN-ACK 包, 并最终断开连接。
- **清理资源:** 最后调用 `closesocket` 关闭套接字, 使用 `WSACleanup` 清理 WinSock 环境。

• **关键点:**

- 使用 Packet 结构体来封装数据和控制信息,如序列号、确认号、标志位(SYN、ACK、FIN)。
- 程序通过接收和发送数据包的过程实现可靠的数据传输,并且确保连接的建立和断开都遵循 TCP 协议的三次握手和四次挥手过程。

3.7 处理数据包

3.7.1 核心代码

```
1  int handle_packet(SOCKET& sock, struct sockaddr_in& sender_addr) {
2
3      // 接收数据包
4      Packet pkt_received;
5      receive_packet(sock, sender_addr, pkt_received);
6
7      // 处理三次握手
8      if (pkt_received.SYN) {
9          my_connect = true;
10         // 收到SYN包, 发送SYN-ACK包
11         Packet pkt;
12         pkt.SYN = true;
13         pkt.ACK = true;
14         pkt.seq_num = seq_num_share++;
15         pkt.ack_num = pkt_received.seq_num + 1;
16         send_packet(sock, sender_addr, pkt);
17         return 0;
18     }
19
20     // 处理三次握手和处理四次挥手
21     if (pkt_received.ACK && pkt_received.ack_num == seq_num_share) {
22         if (my_connect) {
23             my_connect = false;
24             return 0;
25         }
26         if (my_disconnect) {
27             my_disconnect = false;
28             return -1;
29         }
30         return 0;
31     }
32
33     // 处理四次挥手
34     if (pkt_received.FIN) {
35         my_disconnect = true;
36         // 收到FIN包, 发送ACK包
37         Packet pkt1;
38         pkt1.ACK = true;
39         pkt1.seq_num = seq_num_share++;
40         pkt1.ack_num = pkt_received.seq_num + 1;
41         send_packet(sock, sender_addr, pkt1);
42         // 收到FIN包, 发送FIN-ACK包
```

```
43     Packet pkt2;  
44     pkt2.FIN = true;  
45     pkt2.ACK = true;  
46     pkt2.seq_num = seq_num_share++;  
47     pkt2.ack_num = pkt_received.seq_num + 1;  
48     send_packet(sock, sender_addr, pkt2);  
49     return 0;  
50 }  
51  
52 // 将数据写入文件  
53 ofstream file("receive/" + string(pkt_received.name), ios::binary | ios::app);  
54 if (file.is_open()) {  
55     file.write(pkt_received.data, pkt_received.data_len);  
56     file.close();  
57     cout << "数据已写入文件。路径: receive/" + string(pkt_received.name) << endl;  
58 }  
59 else {  
60     cerr << "写入文件失败" << endl;  
61 }  
62  
63 // 发送ACK包  
64 Packet pkt;  
65 pkt.ACK = true;  
66 pkt.seq_num = seq_num_share++;  
67 pkt.ack_num = pkt_received.seq_num + 1;  
68 send_packet(sock, sender_addr, pkt);  
69 return 0;  
70 }
```

3.7.2 代码分析

‘handle_packet’ 函数主要负责处理接收到的数据包。其核心功能包括接收数据、处理三次握手和四次挥手，以及保存文件数据。具体流程如下：

- **接收数据包：**使用 ‘receive_packet’ 函数接收数据包，并将其存储在 ‘pkt_received’ 中，获取数据包内容后进一步处理。
- **三次握手处理：**如果接收到的包含有 SYN 标志位（即 ‘pkt_received.SYN’ 为真），说明是一个连接请求包，表示开始三次握手。此时，设置 ‘my_connect’ 为真，并构造一个包含 SYN 和 ACK 标志位的响应包，发送回客户端以确认连接请求。此时，‘seq_num_share’ 作为序列号递增，保证序列号的正确性。
- **处理连接确认：**若收到带有 ACK 标志且确认号为当前 ‘seq_num_share’ 的数据包，判断是否是连接确认的包。如果 ‘my_connect’ 为真，说明此时正在进行三次握手的最后一步，完成握手后设置 ‘my_connect’ 为假，返回。若是四次挥手的确认包，则将 ‘my_disconnect’ 设置为真，返回 -1 表示连接关闭。
- **四次挥手处理：**如果收到的数据包包含 FIN 标志位，表示客户端请求关闭连接，此

时设置 ‘my_disconnect’ 为真，并发送一个 ACK 包确认收到 FIN 包。接着发送一个带有 FIN 和 ACK 标志的包，以完成四次挥手过程，结束连接。

- **数据保存：**如果数据包既不包含 SYN、ACK 或 FIN 标志，说明是一个数据包。此时将数据包中的内容写入文件，并输出文件保存路径。使用二进制模式打开文件，避免字符数据丢失。
- **发送 ACK 包：**不管数据包是否有效，都会发送一个带 ACK 标志的包，以告知发送方数据包已收到并处理。

3.8 接收端主函数

3.8.1 核心代码

```
1 #pragma comment(lib, "ws2_32.lib")
2 int main() {
3     // 初始化WinSock
4     WSADATA wsaData;
5     if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
6         cerr << "WinSock 初始化失败" << endl;
7         return -1;
8     }
9
10    // 创建UDP套接字
11    SOCKET sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
12    if (sock == INVALID_SOCKET) {
13        cerr << "创建套接字失败" << endl;
14        return -1;
15    }
16
17    // 创建服务器地址
18    sockaddr_in server_addr;
19    server_addr.sin_family = AF_INET; // 设置地址族为IPv4
20    server_addr.sin_port = htons(SERVER_PORT); // 设置端口号为30000
21    server_addr.sin_addr.s_addr = inet_addr(IPADDR); // 设置IP地址为127.0.0.1
22
23    // 绑定套接字
24    if (bind(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR) {
25        cerr << "绑定套接字失败" << endl;
26        return -1;
27    }
28
29    // 创建客户端地址
30    sockaddr_in client_addr;
31    client_addr.sin_family = AF_INET; // 设置地址族为IPv4
32    client_addr.sin_port = htons(CLIENT_PORT); // 设置客户端端口号为20000
33    client_addr.sin_addr.s_addr = inet_addr(IPADDR); // 设置客户端IP地址为127.0.0.1
34
35    // 循环处理
36    while (true) {
```

```
37     int end = handle_packet(sock, client_addr); // 接收数据包并处理
38     if (end == -1)
39         break;
40 }
41
42 closesocket(sock);
43 WSACleanup();
44
45 return 0;
46 }
```

3.8.2 代码分析

‘main’ 函数是接收端程序的入口函数，负责初始化网络环境、创建套接字、绑定本地地址，并在循环中处理接收到的数据包。其主要流程如下：

- **初始化 WinSock**：调用 ‘WSAStartup’ 函数初始化 WinSock 库，以便进行网络通信。若初始化失败，打印错误信息并退出程序。
- **创建 UDP 套接字**：使用 ‘socket’ 函数创建一个 UDP 套接字。如果创建失败，打印错误信息并退出。
- **设置对方地址**：设置 ‘sender_addr’ 变量，配置对方的 IP 地址和端口。此处设置为 ‘127.0.0.1’，即本地回环地址，通常用于调试时的本地通信。
- **绑定套接字**：调用 ‘bind’ 函数将创建的套接字绑定到指定的地址和端口上。如果绑定失败，打印错误信息并退出程序。
- **接收和处理数据包**：进入一个无限循环，调用 ‘handle_packet’ 函数处理接收到的数据包。循环会持续直到 ‘handle_packet’ 返回 -1，表示连接已关闭，此时退出循环并结束程序。
- **关闭套接字并清理资源**：循环结束后，调用 ‘closesocket’ 关闭套接字，最后调用 ‘WSACleanup’ 清理 WinSock 资源，确保程序正常退出。

四、 编译指令

4.1 指令代码

```
1 g++ -o receiver.exe receiver.cpp -lws2_32
2 g++ -o sender.exe sender.cpp -lws2_32
```

4.2 指令含义

1. `'g++ -o receiver.exe receiver.cpp -lws2_32'`

- `'g++'`: GNU 编译器, 用于编译 C++ 源代码。
- `'-o receiver.exe'`: 指定输出文件的名称为 `'receiver.exe'`, 这是编译后的可执行文件。
- `'receiver.cpp'`: 源代码文件, 包含接收端程序的实现。
- `'-lws2_32'`: 链接 WinSock 库 `'ws2_32.lib'`, 这是 Windows 下用于网络编程的库, 用于提供 socket 编程支持。

2. `'g++ -o sender.exe sender.cpp -lws2_32'`

- `'g++'`: 同上, GNU 编译器。
- `'-o sender.exe'`: 指定输出文件的名称为 `'sender.exe'`, 这是编译后的发送端可执行文件。
- `'sender.cpp'`: 源代码文件, 包含发送端程序的实现。
- `'-lws2_32'`: 同上, 链接 `'ws2_32.lib'` 库, 用于网络编程。

这两条指令分别用于编译接收端 (`'receiver.exe'`) 和发送端 (`'sender.exe'`) 的程序, 并确保在 Windows 平台上能够正确进行网络通信。

五、 运行截图

5.1 三次握手 (Connection Establishment)

发送端和接收端通过 SYN 和 ACK 标志位进行三次握手以建立连接：

- 发送端发送一个 SYN 包，表示请求建立连接。
- 接收端收到 SYN 包后，回复 SYN-ACK 包，表示同意建立连接。
- 发送端收到 SYN-ACK 包后，发送 ACK 包，确认连接已建立。

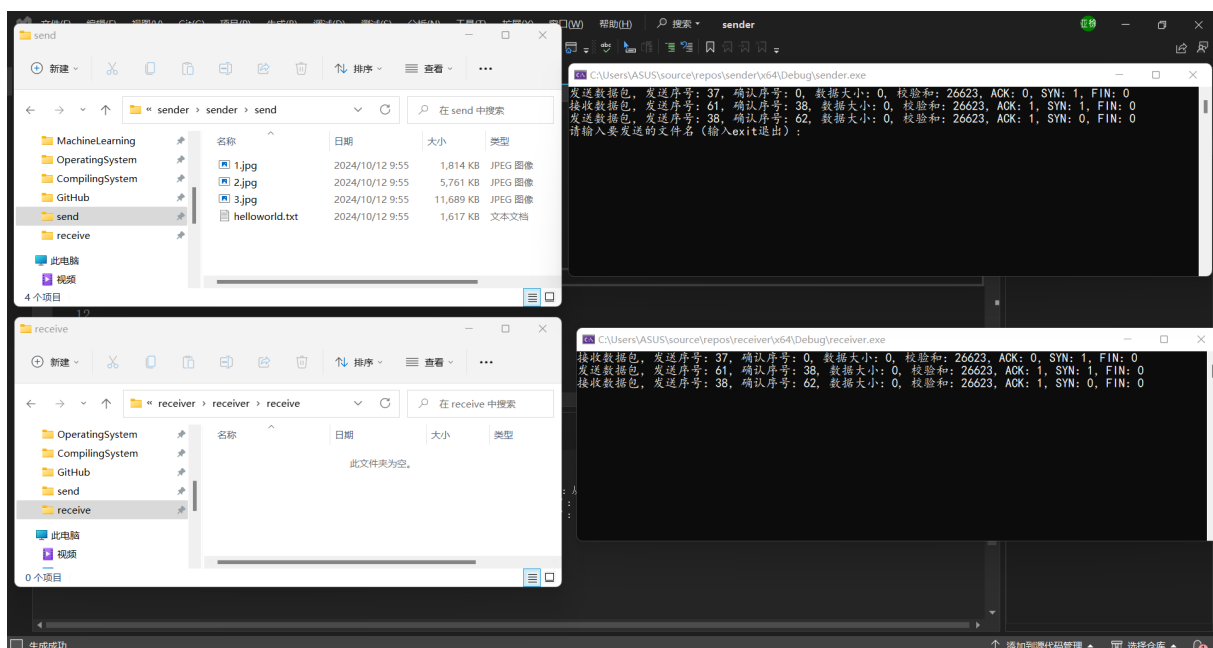


图 1: 三次握手

5.2 文件传输 (File Transmission)

- **文件打开和读取:** 打开指定路径的文件 (send/ + filename)，并使用二进制模式读取文件内容。文件大小通过 seekg 和 tellg 获取。
- **数据包发送:** 文件内容被分割成多个数据包 (Packet)，每个数据包最多包含 sizeof(pkt.data) 字节的数据。发送过程中，会不断读取文件数据，并将其封装进数据包。每发送一个数据包，都会调用 send_packet 函数向接收端发送数据。
- **ACK 处理:** 每次接收到的 ACK 数据包都会检查其确认号和其他字段，以确认数据是否正确接收。收到的 ACK 数据包会打印相关的接收信息，如确认序号、数据大小、校验和等。

- **吞吐率计算:** 在文件发送完成后,记录文件传输的起始和结束时间,通过计算文件大小与传输时间,得出吞吐率,并以 KB/s 为单位输出。

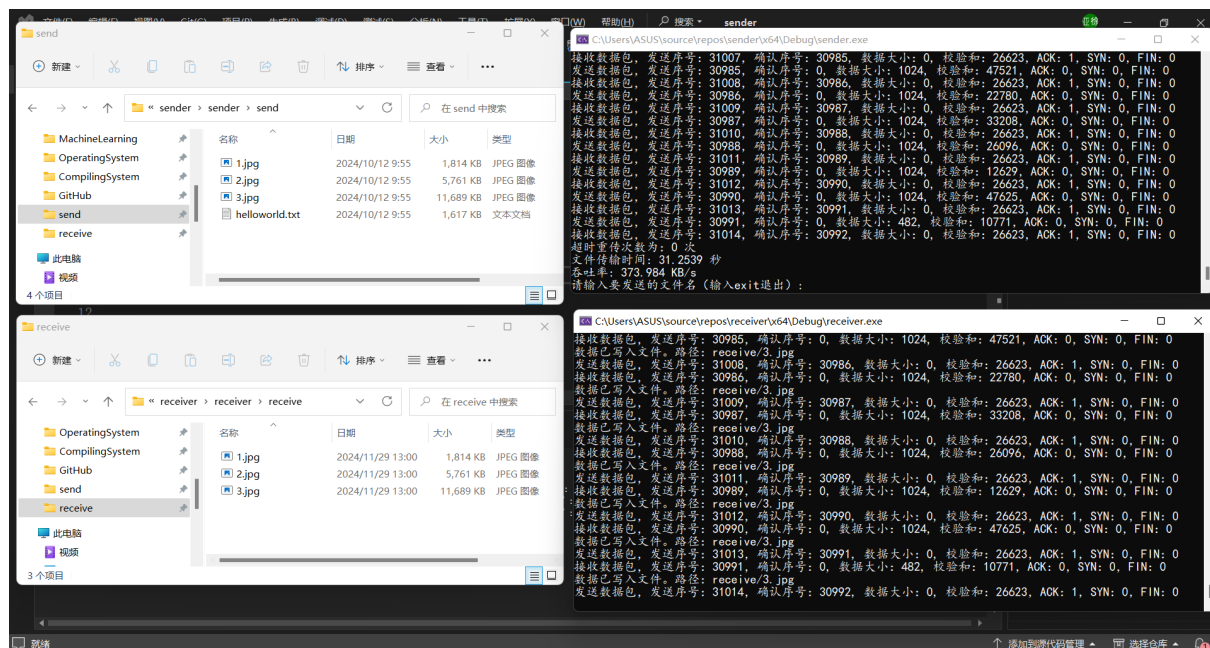


图 2: 传输文件

5.3 超时重传 (Timeout Retransmission)

- **超时重传:** 数据包发送后,会等待接收端的确认包 (ACK)。如果超时未收到 ACK,则进行超时重传,直到接收到确认包或达到重传次数的限制。

5.4 四次挥手 (Connection Termination)

在文件传输完成后,发送端和接收端通过 FIN 标志位进行四次挥手以终止连接:

- 发送端发送 FIN 包,表示请求关闭连接。
- 接收端收到 FIN 包后,发送 ACK 包,确认关闭请求。
- 接收端发送 FIN 包,表示同意关闭连接。
- 发送端收到 FIN-ACK 包后,发送 ACK 包,确认关闭连接。

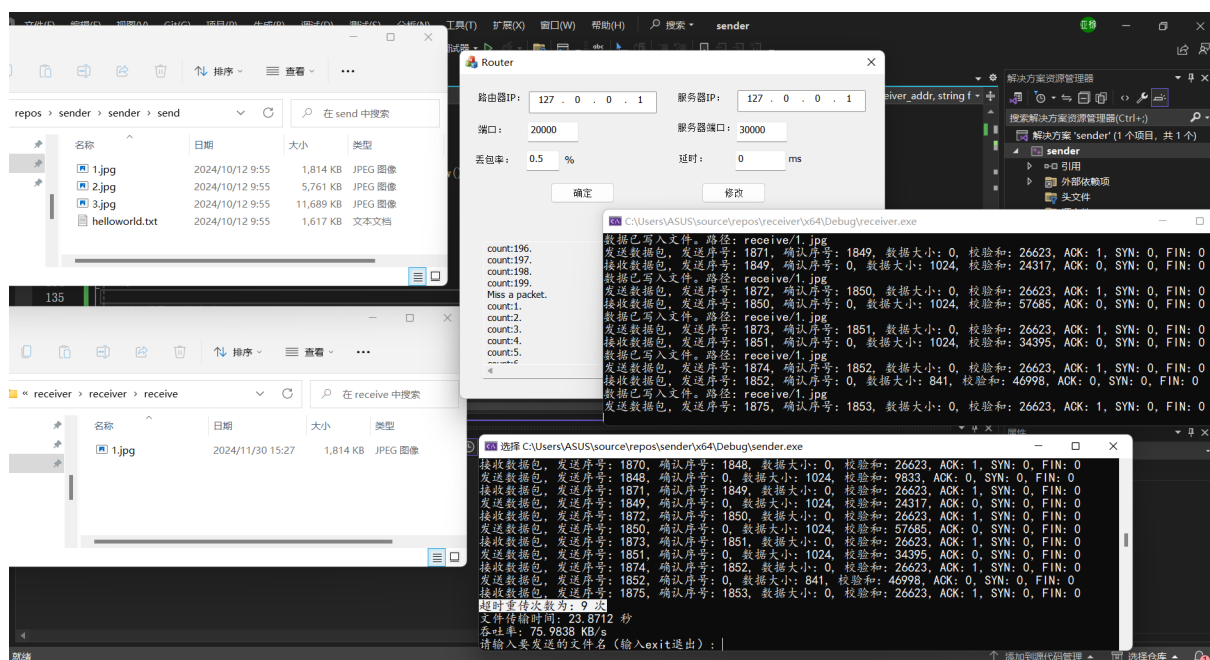


图 3: 超时重传

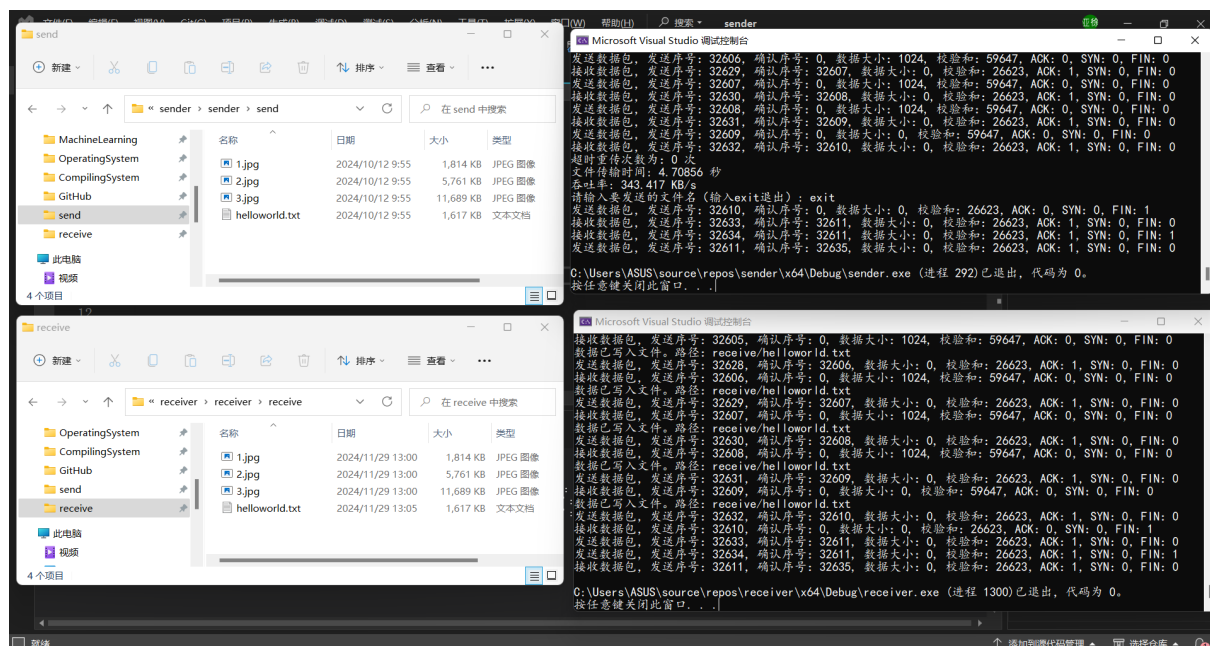


图 4: 四次挥手

六、结果分析

观察发现，传输的文件大小与原文件保持一致，而且图片也可以正常展现，而且文本内容也保持一致，因此说明传输的文件结果正确无误。

source > repos > receiver > receiver > receive





名称	日期	大小	类型	标记
 1.jpg	2024/11/29 13:00	1,814 KB	JPEG 图像	
 2.jpg	2024/11/29 13:00	5,761 KB	JPEG 图像	
 3.jpg	2024/11/29 13:00	11,689 KB	JPEG 图像	
 helloworld.txt	2024/11/29 13:05	1,617 KB	文本文档	

图 5: 文件大小

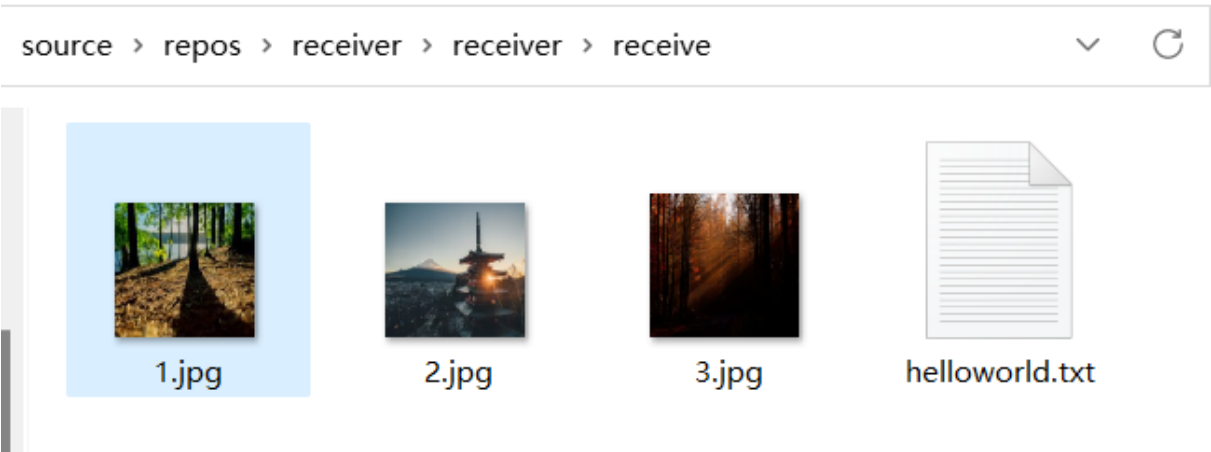


图 6: 图片展现

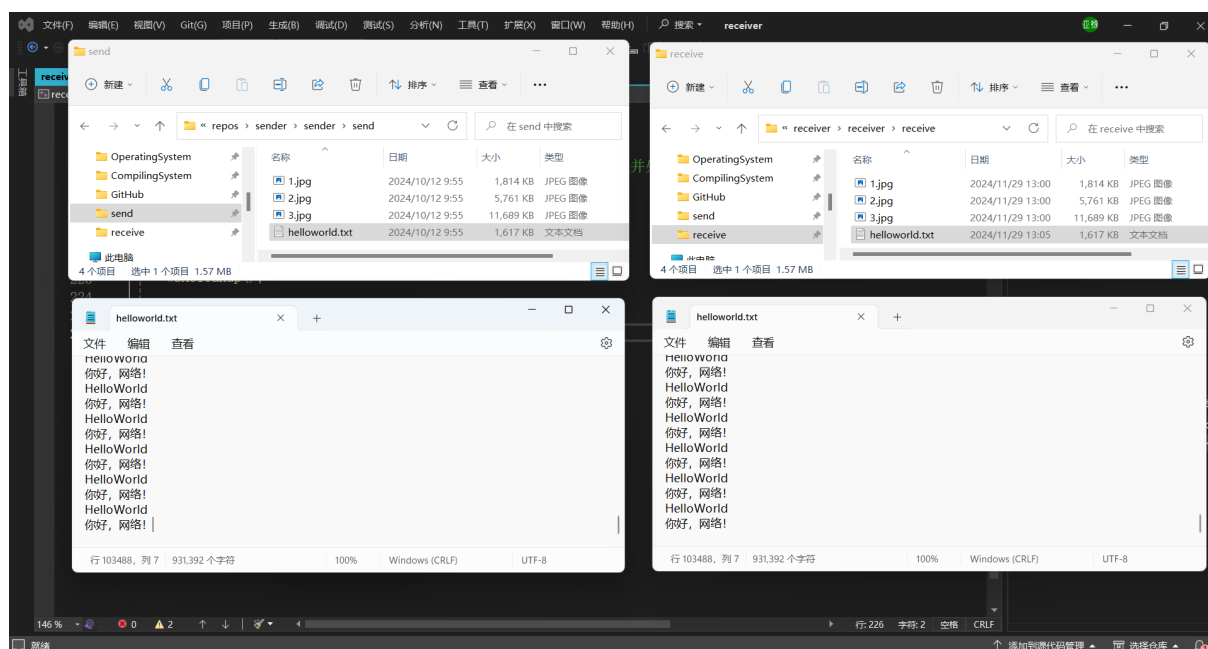


图 7: 文本展现

七、实验总结

本实验旨在实现一个基于 UDP 协议的可靠文件传输系统，并设计了相应的协议来确保数据的完整性、顺序以及连接管理。通过实现三次握手和四次挥手机制，我们成功地模拟了连接的建立和断开，同时通过校验和保证了数据在传输过程中的完整性。

在实验中，我们首先设计了一个简单的协议，采用了数据包结构，其中包含了序列号、确认号、文件名、数据内容、校验和、以及标志位（SYN、ACK、FIN）。通过这个协议，我们能够确保数据包的按序传输和正确的确认机制。此外，我们还实现了超时重传机制，用于处理数据包丢失或确认延迟的情况。

发送端在传输过程中逐块读取文件，并在每次发送数据包后等待接收端的 ACK 确认包。如果超时未收到 ACK 包，则会进行重传，确保文件能够完整地到达接收端。接收端接收到数据包后，通过验证校验和保证数据的完整性，并根据数据包中的序列号进行确认和排序，确保接收的数据包顺序正确。

在性能测试部分，我们主要关注了吞吐率和文件传输时延两个指标。通过不同文件大小的测试，计算了每个文件的传输时间，并得出了平均吞吐率。结果显示，在文件大小较小时，吞吐率较高，而随着文件大小的增加，吞吐率会有所下降，主要是由于发送端和接收端之间的通信延迟和网络拥塞所导致。

通过本次实验，我们不仅实现了一个基于 UDP 协议的可靠文件传输系统，还深入理解了文件传输协议中的一些核心概念，如数据包的构建、连接的建立与断开、数据的完整性保证以及拥塞控制等。实验中，我们成功模拟了文件传输过程中的各个环节，并

通过合理的协议设计保证了数据的正确传输。

实验的难点主要在于如何处理数据包的顺序、丢包重传和连接管理等问题。通过不断调试和优化，最终实现了一个能够稳定运行的文件传输程序。未来，若要进一步提高系统的性能，可以考虑引入更复杂的拥塞控制算法以及基于 TCP 的流量控制机制，以进一步提升系统的吞吐量和稳定性。

总之，本实验帮助我们深入理解了网络协议的设计与实现过程，并为今后在网络通信领域的深入研究和实践打下了坚实的基础。