



南開大學

Nankai University

计算机学院

计算机网络实验报告

实验 3.1：基于 UDP 服务设计可靠传输协议并编程实现

姓名：刘浩泽

学号：2212478

专业：计算机科学与技术

班级：计算机卓越班

2024 年 11 月 30 日

目录

1 实验要求	2
2 UDP 报文设计	2
3 三次握手建立连接	3
3.1 三次握手的基本流程	3
3.2 报文传输前数据包的构建	4
3.3 接收到数据包后进行差错检测和校验	4
3.4 三次握手传输过程中的超时重传和快速重传机制	5
4 文件传输：采用停等机制进行流量控制	10
4.1 文件传输的基本流程	10
4.2 报文传输前数据包的构建	13
4.3 接收到数据包后进行差错检测和校验	14
4.4 文件传输过程中的超时重传和快速重传机制	16
4.5 客户端采取停等机制进行流量控制	16
5 三次挥手释放连接	19
5.1 三次挥手的基本流程	19
5.2 接收到数据包后进行差错检测和校验	20
6 文件传输效果和性能测试指标	21
7 实验内容总结	22

1 实验要求

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传等。流量控制采用停等机制，完成给定测试文件的传输。

- (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) 完成给定测试文件的传输，显示传输时间和平均吞吐率。
- (4) 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。
- (5) 现场演示，完成详细的实验报告，提交程序源码、可执行文件和实验报告。

2 UDP 报文设计

基于 UDP 服务设计可靠的传输协议很大程度上依赖于报文的设计。由于 UDP 本身是无连接的，即客户端和服务端之间不需要建立连接即可进行数据传输，并且也为了减少报文首部的长度，因此传统的 UDP 报文字段不包含序列号和确认号等字段，也没有 SYN、ACK、FIN 等为了确保数据可靠传输而设计的标志位。而在本实验中，由于我们需要实现可靠的数据传输，因此我们需要重新设计 UDP 的报文格式。具体来说，**首先我设置了一个伪首部，在数据传输的过程中不进行传送，只用于服务端和客户端进行数据校验**，包括 32 位源 IP 地址，32 位目的 IP 地址，8 位全 0 填充位，8 位协议号，16 位存储首部和数据部分的长度，如下图所示。

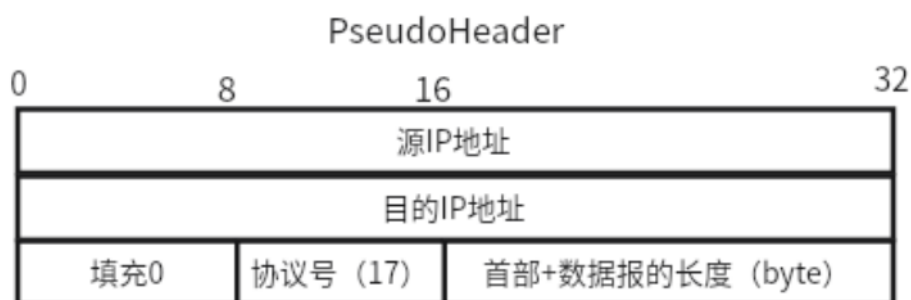


图 2.1: 伪首部报文格式

首部和数据部分在数据传输的过程中都需要进行传送，**对于首部的报文格式**，我设置了 16 位源端口号，16 位目的端口号，32 位序列号，32 位确认号，8 位标志位包括 ACK，SYN 标志位等，8 位全 0 填充位，16 位存储首部和数据部分的长度，16 位校验和用于服务端和客户端进行数据校验，16 位窗口大小（在本实验中由于采用停等机制，窗口大小设置为 1），**如下图所示**。IP 地址和端口号的加入是为了确保程序的可扩展性，确保更换 IP 地址后程序依然可以正常运行。窗口大小的加入是为了在后续实验中实现滑动窗口和动态调整窗口大小做准备。同时在标志位中，我设置了 FILE_END 标志位作为一个文件传输结束的标志，用于通知服务端当前文件传输完毕，服务端可以继续准备接收下一个文件。

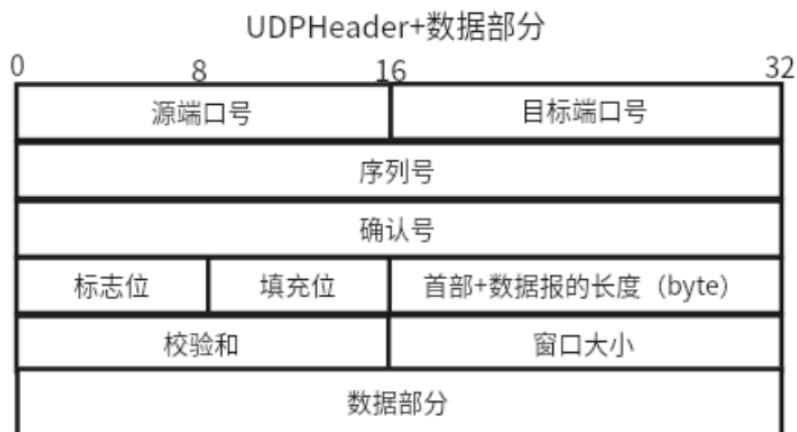


图 2.2: 首部报文格式

3 三次握手建立连接

3.1 三次握手的基本流程

在本实验中，由于我们希望基于 UDP 实现可靠的数据传输，因此在数据传输前需要首先像 TCP 协议一样建立可靠的连接，确保数据传输的可靠性。本实验中我采用了三次握手的方式来建立连接，即当客户端希望与服务端建立连接时，会尝试与服务端进行三次握手，确保双方能够成功建立连接。客户端会首先向服务端发送带有 SYN 标志位的数据报，表示希望建立连接，服务端在接收到客户端发来的数据报后会回复带有 SYN 和 ACK 标志位的数据报，表示同意建立连接，同时也希望与客户端建立连接，最后客户端回复 ACK 确认信号，完成连接的建立。具体实现效果如下图所示：

<pre> D:\vs2022\client\x64\Debug\client.exe ***** ***** 欢迎进入文件传输系统 ***** ***** 请输入消息或命令（'q' 退出，'s' 连接服务器）： s 客户端套接字创建成功。 =====三次握手建立连接===== 尝试第一次握手建立连接... 数据发送成功，等待SYN ACK... 发送数据包的序列号：0，ACK号：0，校验和：38720 期望收到的ACK号：1 实际收到的ACK号：1 收到数据包的序列号：0，ACK号：1，校验和：5566 收到正确的SYN ACK，第二次握手成功！ 开始尝试进行第三次握手 第三次握手发送的序列号：1，ACK号：1，校验和：33362 第三次握手成功，成功建立连接 成功建立连接 ===== 请输入要上传的文件路径（输入q释放连接）： </pre>	<pre> D:\vs2022\server\x64\Debug\server.exe 服务端套接字创建成功。 服务器已启动，等待连接... 接收到SYN，准备发送SYN-ACK... [来自客户端]收到数据包的序列号：0，ACK号：0，校验和：38720 尝试第二次握手建立连接... 数据发送成功，等待ACK... 发送数据包的序列号：0，ACK号：1，校验和：5566 期望收到的ACK号：1 实际收到的ACK号：1 [服务端]收到数据包的序列号：1，ACK号：1，校验和：33362 收到正确的ACK，第三次握手成功！ 成功建立连接 </pre>
---	---

图 3.3: 三次握手的实现效果

3.2 报文传输前数据包的构建

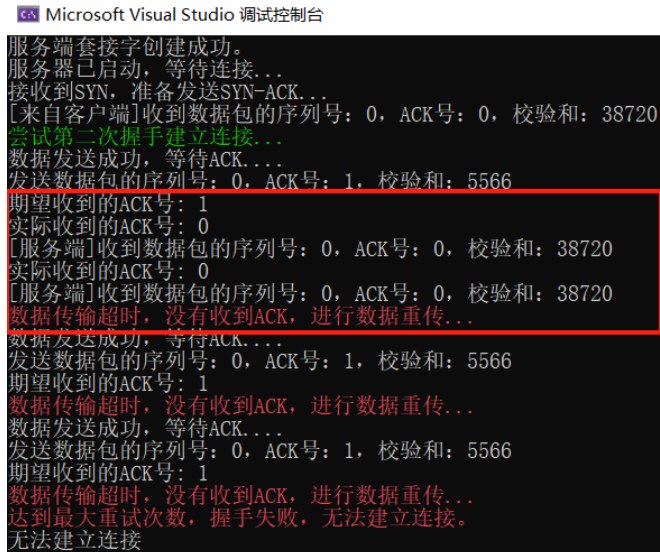
客户端在进行报文传输前，首先需要利用传入的客户端源 IP 地址和服务端目的 IP 地址构造出伪首部。然后利用 `createPacket` 函数根据传入的伪首部和首部计算出校验和并填充到首部的校验和字段，并将 UDP 首部与数据部分进行拼接，构造出数据包。然后进入到后续的传输逻辑。具体代码实现如下所示：

```
1 // 构造 UDP 数据包（包括首部和数据部分）
2 vector<char> createPacket(UDPHeader& header, const char* data, int dataLength,
3     const PseudoHeader& pseudoHeader) {
4     // 拼接 UDP 头部和数据
5     vector<char> packet(HEADER_SIZE + dataLength);
6     // 计算校验和
7     header.checksum = calculateChecksum(pseudoHeader, header, data, dataLength);
8     // 更新校验和到 UDP 头部
9     memcpy(packet.data(), &header, HEADER_SIZE);
10    memcpy(packet.data() + HEADER_SIZE, data, dataLength); // 拷贝数据部分
11    return packet;
12 }
```

3.3 接收到数据包后进行差错检测和校验

在客户端接收到来自对方的 SYN、ACK 数据包时，需先从接收到的字节流中提取出 UDP 数据报的首部，然后对数据包的各个字段进行检查和验证，以确保数据的可靠性和完整性。需要查看首部的标志位是否符合自己的要求，并且检查确认号是否匹配，匹配的话则发送 ACK 信号进行确认。而当服务端接受到客户端返回的 ACK 确认信号后，也需要进行类似的操作，检查数据包的首部字段和确认号是否匹配。并且当进行数据传输时，服务端还需要根据数据包的源 IP 地址和目的 IP 地址重新构造出伪首部来对校验和字段进行校验，通过检验校验和的正确性来确定数据在传输过程中没有被篡改或丢失。如果得到的数据包校验和错误，即数据在传输过程中可能被篡改，则服务端会将该数据包丢弃，等待客户端进行重传。

在三次握手建立连接的过程中，以服务端为例，具体的实现效果如下图所示：



```

Microsoft Visual Studio 调试控制台
服务端套接字创建成功。
服务器已启动，等待连接...
接收到SYN，准备发送SYN-ACK...
[来自客户端]收到数据包的序列号：0，ACK号：0，校验和：38720
尝试第二次握手建立连接...
数据发送成功，等待ACK...
发送数据包的序列号：0，ACK号：1，校验和：5566
期望收到的ACK号：1
实际收到的ACK号：0
[服务端]收到数据包的序列号：0，ACK号：0，校验和：38720
实际收到的ACK号：0
[服务端]收到数据包的序列号：0，ACK号：0，校验和：38720
数据传输超时，没有收到ACK，进行数据重传...
数据发送成功，等待ACK...
发送数据包的序列号：0，ACK号：1，校验和：5566
期望收到的ACK号：1
数据传输超时，没有收到ACK，进行数据重传...
数据发送成功，等待ACK...
发送数据包的序列号：0，ACK号：1，校验和：5566
期望收到的ACK号：1
数据传输超时，没有收到ACK，进行数据重传...
达到最大重试次数，握手失败，无法建立连接。
无法建立连接

```

图 3.4: 差错检测和校验

具体的代码实现如下所示：

```

1 int bytesReceived = recvfrom(serverSocket, buffer, BUFFER_SIZE, 0, (sockaddr*)&recvAddr,
2     &recvAddrLen);
3 if (bytesReceived > 0) {
4     UDPHeader* recvHeader = (UDPHeader*)buffer;
5     cout << "[服务端] 收到数据包的序列号：" << recvHeader->sequenceNumber << "，ACK 号："
6     << recvHeader->acknowledgmentNumber << "，校验和：" << recvHeader->checksum << endl;
7     // 检查是否收到正确的 ACK
8     if (recvHeader->flags == ACK_FLAG && recvHeader->acknowledgmentNumber == expectedAckNumber)
9         setConsoleColor(10);
10        cout << "收到正确的 ACK，第三次握手成功!" << endl;
11        setConsoleColor(7);
12        handshakeComplete = true; // 握手成功，退出循环
13        return true;
14    }

```

3.4 三次握手传输过程中的超时重传和快速重传机制

在三次握手的过程中，我采取了如下机制以确保传输的稳定性和可靠性。

- 首先，我采用了**超时重传机制**，当客户端向服务端发送 SYN 信号或数据包，或者服务端向客户端发送 SYN、ACK 信号时，会使用 **select** 系统调用来进行**超时检测**。如果客户端在预设的时间内没有收到服务端的确认包，则会认为数据包已经丢失，触发超时重传机制，重新发送该数据包。这里我使用了 **select** 系统调用来检测是否超时。具体来说，**select** 通过监控指定的套接字（代码里是检测客户端的 `clientSocket` 套接字）是否可以接收到数据，同时设置一个超时时间（以微秒为单位）。当超时发生时，**select** 会返回 0，允许程序根据超时情况执行处理逻辑，重传数据包。

并继续执行。而当超时情况不发生即客户端正常接收到回复的 ACK 信号时, select 会返回一个大于 0 的正常值, 接着客户端会接收回复的 ACK 信号并继续进行后续处理。而如果我们使用普通的时钟进行计时的话, 客户端如果没有接收到回复的 ACK 信号, 则会被阻塞在等待状态, 时钟也不会继续计时, 这样就会导致无法触发超时重传机制。并且通过 select 系统调用进行超时检测, 客户端能够在等待期间继续执行其他任务, 不仅提升了程序的响应效率, 还能确保在确认包丢失或网络状况不佳时及时进行重传, 从而增强了连接的可靠性和稳定性。

- 同时, 为了避免重传的次数过多, 客户端会记录超时情况下当前数据包重传的次数, 如果在超时后客户端重传某个数据包的次数超过了三次, 则会认为当前服务端无法建立连接, 终止重传并关闭连接。这一机制旨在防止无限重传带来的性能浪费, 并确保客户端在无法建立连接的情况下及时终止连接尝试, 避免过多的资源消耗。类似的, 当服务端向客户端发送 SYN、ACK 信号并等待客户端确认信号回复时, 也会采取上述机制。

具体实现效果如下图所示:

```

Microsoft Visual Studio 调试控制台
***** 欢迎进入文件传输系统 *****
*****
请输入消息或命令 ('q' 退出, 's' 连接服务器):
s
客户端套接字创建成功。
=====三次握手建立连接=====
尝试第一次握手建立连接...
数据发送成功, 等待SYN ACK...
发送数据包的序列号: 0, ACK号: 0, 校验和: 38720
期望收到的ACK号: 1
数据传输超时, 没有收到ack, 进行数据重传...
数据发送成功, 等待SYN ACK...
发送数据包的序列号: 0, ACK号: 0, 校验和: 38720
期望收到的ACK号: 1
数据传输超时, 没有收到ack, 进行数据重传...
数据发送成功, 等待SYN ACK...
发送数据包的序列号: 0, ACK号: 0, 校验和: 38720
期望收到的ACK号: 1
数据传输超时, 没有收到ack, 进行数据重传...
达到最大重试次数, 握手失败, 服务器当前无法建立连接。
=====

```

图 3.5: 超时重传和避免无限重传

- 同时, 为了提高传输的效率, 客户端还实现了快速重传机制。当客户端连续收到三个不符合预期的确认号 (例如, 错误的确认号) 时, 客户端会认为当前数据包已丢失, 并立即重传 SYN 包或数据包, 而无需等待超时。这一机制能够有效减少由网络延迟、丢包等原因引起的不必要等待, 确保数据能够尽快传输, 从而提高了整体的传输速度和系统的响应能力。快速重传的实现大大降低了因网络波动导致的长时间等待, 提升了网络连接的稳定性和数据交互的效率。类似的, 当服务端向客户端发送 SYN、ACK 信号并等待客户端确认信号回复时, 也会采取上述机制。

以客户端为例, 具体的代码实现如下所示:

```

1 bool performThreeWayHandshake(SOCKET clientSocket, const sockaddr_in& clientAddr, const
2   sockaddr_in& serverAddr, UDPHeader& header, const char* data, int dataLength,
3   uint32_t& sequenceNumber, uint32_t& acknowledgmentNumber) {
4   PseudoHeader pseudoHeader = createPseudoHeader(clientAddr.sin_addr.s_addr,
5   serverAddr.sin_addr.s_addr, HEADER_SIZE + dataLength);

```



```

6     vector<char> packet = createPacket(header, data, dataLength, pseudoHeader);
7     cout << "===== 三次握手建立连接 =====" << endl;
8     // 三次握手：第一步，发送 SYN
9     setConsoleColor(10); // 设置为绿色
10    cout << "尝试第一次握手建立连接..." << endl;
11    setConsoleColor(7);
12    int retries = 0;
13    int wrongAckCount = 0; // 记录连续收到的错误 ACK 数量
14    bool three_wrong = false;
15    bool reach_time = false;
16    bool handshakeComplete = false; // 用来标记三次握手是否完成
17    while (retries < MAX_RETRIES && !handshakeComplete) {
18        // 重置重试和错误 ACK 计数
19        wrongAckCount = 0;
20        three_wrong = false;
21        reach_time = false;
22        // 发送数据包
23        if (sendto(clientSocket, packet.data(), packet.size(), 0, (sockaddr*)&serverAddr,
24        sizeof(serverAddr)) == SOCKET_ERROR) {
25            cerr << "发送数据包失败!" << endl;
26            return false;
27        }
28        cout << "数据发送成功，等待 SYN ACK...." << endl;
29        cout << "发送数据包的序列号：" << header.sequenceNumber << ", ACK 号："
30        << header.acknowledgmentNumber << ", 校验和：" << header.checksum << endl;
31        uint32_t expectedAckNumber = header.sequenceNumber + 1;
32        cout << "期望收到的 ACK 号：" << expectedAckNumber << endl;
33        // 等待 ACK，持续超时检测
34        while (!handshakeComplete) {
35            char buffer[BUFFER_SIZE] = {};
36            sockaddr_in recvAddr = {};
37            int recvAddrLen = sizeof(recvAddr);
38            // 使用 select 进行超时判断，持续监测 ACK 接收
39            fd_set readfds;
40            struct timeval timeout;
41            timeout.tv_sec = 0;
42            timeout.tv_usec = TIMEOUT_MS * 1000;
43            FD_ZERO(&readfds);
44            FD_SET(clientSocket, &readfds);
45            int result = select(0, &readfds, nullptr, nullptr, &timeout);
46            if (result > 0 && FD_ISSET(clientSocket, &readfds)) {
47                // 有数据到达

```



```
48     int bytesReceived = recvfrom(clientSocket, buffer, BUFFER_SIZE, 0,
49     (sockaddr*)&recvAddr, &recvAddrLen);
50     if (bytesReceived > 0) {
51         UDPHeader* recvHeader = (UDPHeader*)buffer;
52         cout << "实际收到的 ACK 号: " << recvHeader->acknowledgmentNumber << endl;
53         cout << "收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: "
54         << recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum
55         << endl;
56         // 检查是否收到正确的 ACK
57         if (recvHeader->flags == SYN_ACK_FLAG
58         && recvHeader->acknowledgmentNumber == expectedAckNumber) {
59             setConsoleColor(10); // 设置为绿色
60             cout << "收到正确的 SYN ACK, 第二次握手成功!" << endl;
61             setConsoleColor(7);
62             cout << "开始尝试进行第三次握手" << endl;
63             ++sequenceNumber;
64             acknowledgmentNumber = recvHeader->sequenceNumber + 1;
65             UDPHeader header_new = createUDPHeader(header.sourcePort,
66             header.destPort, sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
67             PseudoHeader pseudoHeader_new = createPseudoHeader(
68             clientAddr.sin_addr.s_addr, serverAddr.sin_addr.s_addr, HEADER_SIZE);
69             vector<char> packet_new = createPacket(header_new, nullptr,
70             0, pseudoHeader_new);
71             // 输出第三次握手的序列号和确认号
72             cout << "第三次握手发送的序列号: " << header_new.sequenceNumber
73             << ", ACK 号: " << header_new.acknowledgmentNumber
74             << ", 校验和: " << header_new.checksum << endl;
75             if (sendto(clientSocket, packet_new.data(), packet_new.size(), 0,
76             (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) {
77                 cout << "第三次握手失败" << endl;
78                 return false;
79             }
80             else {
81                 setConsoleColor(10);
82                 cout << "第三次握手成功, 成功建立连接" << endl;
83                 setConsoleColor(7);
84                 handshakeComplete = true; // 握手成功, 退出循环
85             }
86         }
87         else {
88             wrongAckCount++;
89             if (wrongAckCount >= 3) {
```

```
90         cerr << "连续收到 3 个错误的 ACK, 准备重传数据包!" << endl;
91         cout << "进行数据重传..." << endl;
92         three_wrong = true;
93         break; // 进入重传逻辑
94     }
95 }
96 }
97 }
98 else if (result == 0) {
99     // 如果超时, 进行重传
100     setConsoleColor(12); // 设置为红色
101     cout << "数据传输超时, 没有收到 ack, 进行数据重传..." << endl;
102     setConsoleColor(7);
103     reach_time = true;
104     retries++;
105     if (retries >= MAX_RETRIES) {
106         setConsoleColor(12); // 设置为红色
107         cerr << "达到最大重试次数, 握手失败, 服务器当前无法建立连接。" << endl;
108         setConsoleColor(7);
109         cout << "===== " << endl;
110         closesocket(clientSocket);
111         WSACleanup();
112         return false;
113     }
114     break; // 退出等待 ACK 的循环
115 }
116 }
117 if (three_wrong reach_time) {
118     continue; // 超时或错误 ACK 后重试
119 }
120 }
121 // 如果达到最大重试次数还是无法建立连接, 则返回失败
122 if (!handshakeComplete) {
123     cerr << "达到最大重试次数, 无法完成三次握手" << endl;
124     closesocket(clientSocket);
125     WSACleanup();
126     return false;
127 }
128 return true;
129 }
```

4 文件传输：采用停等机制进行流量控制

4.1 文件传输的基本流程

当客户端需要进行文件传输时，首先，客户端会向服务端发送传输的文件名称，也就是说服务端需要首先接收文件名，并创建相应的文件进行接收。后续再继续进行文件内容的传输。为了确保数据的完整性和正确性，服务端必须对接收到的文件名进行校验和差错检测，确认文件名没有出现数据传输错误。只有在文件名的校验和验证通过后，服务端才会将 flag 设置为 1，表示准备好接收文件内容，随后进入到文件内容接收的逻辑。在接收文件内容的过程中，服务端会对客户端发送的每一个数据包进行校验和检测，确保数据在传输过程中没有损坏或丢失。如果校验和验证失败，表示该数据包有误，服务端将丢弃该包并等待客户端重新发送该数据包。这种机制能够有效地保证文件内容的正确性和完整性，防止数据传输过程中出现不可恢复的错误。

一旦服务端成功接收到数据包且校验和通过，接收到的数据将被写入到指定的文件中。此时，服务端会向客户端返回一个 ACK 确认包，通知客户端该数据已经成功接收，并准备接收下一个数据包。ACK 确认包中包含当前的数据包序列号和确认号，用于确保客户端能够准确识别哪些数据包已经成功接收，哪些数据包需要重传。具体代码实现如下所示：

```

1  if (flag == 0)
2  {
3      // 第一次收到数据包，数据包内容是文件名
4      string fileName(buffer_fin + HEADER_SIZE, bytesReceived - HEADER_SIZE); // 提取文件名
5      cout << "接收到文件名：" << fileName << endl;
6      cout << "收到数据包的序列号：" << recvHeader->sequenceNumber << ", ACK 号：" <<
7          recvHeader->acknowledgmentNumber << ", 校验和：" << recvHeader->checksum << endl;
8      // 创建文件路径，并打开文件
9      string filePath = "D:/new/" + fileName; // 拼接文件路径
10     outFile.open(filePath, ios::binary); // 使用文件路径创建文件
11     if (!outFile) {
12         cerr << "无法创建文件：" << filePath << endl;
13         return;
14     }
15     flag=1;
16     char* data = buffer_fin + HEADER_SIZE;
17     int dataLength = bytesReceived - HEADER_SIZE;
18     if (checkheader(pseudorecvHeader, *recvHeader, data, bytesReceived - HEADER_SIZE) == true)
19     {
20         acknowledgmentNumber = recvHeader->sequenceNumber + 1;
21         UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort,
22         ++sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
23         PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
24         clientAddr.sin_addr.s_addr, HEADER_SIZE);
25         vector<char> ackPacket = createPacket(ackHeader, nullptr, 0, ackPseudoHeader);
26         sendto(serverSocket, ackPacket.data(), ackPacket.size(), 0, (sockaddr*)&clientAddr,

```

```

27         clientAddrLen);
28     cout << "已发送 ACK 确认!" << endl;
29     cout << "发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
30         ackHeader.acknowledgmentNumber << ", 校验和: " << ackHeader.checksum << endl;
31 }
32 continue;
33 }
34 else
35 {
36     cout << "===== 进行数据接收 =====" << endl;
37     cout << "[来自客户端] 收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: "
38         << recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum << endl;
39     char* data = buffer_fin + HEADER_SIZE;
40     int dataLength = bytesReceived - HEADER_SIZE;
41     if (checkheader(pseudorecvHeader, *recvHeader, data, bytesReceived - HEADER_SIZE) == true)
42     {
43         // 保存到文件
44         outFile.write(data, dataLength);
45         cout << "已写入数据块: " << dataLength << " 字节" << endl;
46         acknowledgmentNumber = recvHeader->sequenceNumber + 1;
47         UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort,
48             ++sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
49         PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
50             clientAddr.sin_addr.s_addr, HEADER_SIZE);
51         vector<char> ackPacket = createPacket(ackHeader, nullptr, 0, ackPseudoHeader);
52         sendto(serverSocket, ackPacket.data(), ackPacket.size(), 0, (sockaddr*)&clientAddr,
53             clientAddrLen);
54         cout << "已发送 ACK 确认!" << endl;
55         cout << "[服务端] 发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
56             ackHeader.acknowledgmentNumber << ", 校验和: " << ackHeader.checksum << endl;
57         cout << "===== " << endl;
58     }
59 }

```

当文件传输完毕后，客户端会发送一个带有 FILE_END 标志的数据报，用来告知服务端当前文件已经传输完毕。而服务端在接受到这个报文后，会回复一个带有 ACK 标志的数据报，作为对客户端的确认响应并告知客户端文件已成功接收并且完整无误。接着服务端会关闭文件流，完成文件的写入操作，并重置 flag 为 0，继续进入循环，准备接收下一个文件的文件名和文件内容，然后重复上述接收过程。具体实现效果如下图所示：

D:\vs2022\client\x64\Debug\client.exe	D:\vs2022\server\x64\Debug\server.exe
收到数据包的序列号: 366, ACK号: 368, 校验和: 5089 收到正确的ACK, 数据传输成功! 当前数据包传输完毕	已写入数据块: 10240 字节 已发送ACK确认! [服务端]发送数据包的序列号: 364, ACK号: 366, 校验和: 5093
=====发送当前数据包=====	=====进行数据接收=====
数据发送成功, 等待ACK...	[来自客户端]收到数据包的序列号: 366, ACK号: 366, 校验和: 63320
发送数据包的序列号: 368, ACK号: 368, 校验和: 51118	校验正确
期望收到的ACK号: 369	已写入数据块: 10240 字节
实际收到的ACK号: 369	已发送ACK确认!
收到数据包的序列号: 367, ACK号: 369, 校验和: 5087	[服务端]发送数据包的序列号: 365, ACK号: 367, 校验和: 5091
收到正确的ACK, 数据传输成功!	=====进行数据接收=====
当前数据包传输完毕	[来自客户端]收到数据包的序列号: 367, ACK号: 367, 校验和: 14026
文件传输完毕!	校验正确
文件大小: 1.77131 MB	已写入数据块: 10240 字节
传输时间: 32.9568 秒	已发送ACK确认!
吞吐率: 0.0537464 MB/s	[服务端]发送数据包的序列号: 366, ACK号: 368, 校验和: 5089
=====发送文件传输完毕标志=====	=====进行数据接收=====
开始发送文件传输完毕标志	[来自客户端]收到数据包的序列号: 368, ACK号: 368, 校验和: 51118
=====发送当前数据包=====	校验正确
数据发送成功, 等待ACK...	已写入数据块: 3913 字节
发送数据包的序列号: 369, ACK号: 369, 校验和: 36446	已发送ACK确认!
期望收到的ACK号: 370	[服务端]发送数据包的序列号: 367, ACK号: 369, 校验和: 5087
实际收到的ACK号: 370	=====接收到的FILE_END, 准备发送ACK...=====
收到数据包的序列号: 368, ACK号: 370, 校验和: 5085	[来自客户端]收到数据包的序列号: 369, ACK号: 369, 校验和: 36446
收到正确的ACK, 数据传输成功!	[服务端]发送数据包的序列号: 368, ACK号: 370, 校验和: 5085
文件传输完毕标志传输完毕	已发送ACK, 本次文件接收完毕
=====	
请输入要上传的文件路径 (输入0释放连接):	

图 4.6: 文件传输的实现效果

具体代码实现如下所示:

```

1 else if (recvHeader->flags == FILE_END)
2 {
3     setConsoleColor(10);
4     cout << "接收到 FILE_END, 准备发送 ACK..." << endl;
5     cout << "[来自客户端] 收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: "
6         << recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum << endl;
7     setConsoleColor(7);
8     acknowledgmentNumber = (recvHeader->sequenceNumber) + 1;
9     UDPHeader synHeader = createUDPHeader(serverPort, recvHeader->sourcePort, ++sequenceNumber,
10        acknowledgmentNumber, ACK_FLAG, 0);
11     // 发送数据包
12     PseudoHeader pseudosynHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
13        clientAddr.sin_addr.s_addr, HEADER_SIZE);
14     vector<char> synAckPacket = createPacket(synHeader, nullptr, 0, pseudosynHeader);
15     sendto(serverSocket, synAckPacket.data(), synAckPacket.size(), 0, (sockaddr*)&clientAddr,
16        clientAddrLen);
17     setConsoleColor(10);
18     cout << "[服务端] 发送数据包的序列号: " << synHeader.sequenceNumber << ", ACK 号: " <<
19         synHeader.acknowledgmentNumber << ", 校验和: " << synHeader.checksum << endl;
20     cout << "已发送 ACK, 本次文件接收完毕" << endl;
21     setConsoleColor(7);

```

```

22 // 关闭文件流, 准备接收下一个文件
23 if (outFile.is_open()) {
24     outFile.close();
25 }
26 flag = 0;
27 continue;
28 }

```

4.2 报文传输前数据包的构建

与上述三次握手的流程类似, 客户端在进行文件传输时, 首先需要利用传入的客户端源 IP 地址和服务端目的 IP 地址构造出伪首部。然后利用 `createPacket` 函数根据传入的伪首部和首部计算出校验和并填充到首部的校验和字段, 并将 UDP 首部与数据部分进行拼接, 构造出数据报。然后进入到后续的传输逻辑。

计算校验和的逻辑如下所示, 首先对伪首部的各个字段进行求和, 由于校验和我们设置为 16 位, 对于 32 位的源 IP 地址和目的 IP 地址, 我们需要进行拆分, 并累加到 `sum` 中, 接下来依次对伪首部的剩余字段进行求和, 不足 16 位的需要进行拼接或者填充 0 字段, 以确保满足 16 位对齐。然后采取类似的流程对首部的各个字段进行求和。对于数据部分, 由于我们需要进行 16 位即 2 字节对齐, 因此我们每次取 2 个字节进行拼接, 并累加到 `sum` 中, 如果数据长度为奇数, 则将最后一个字节与 0 填充位拼接形成一个 16 位数据。最后将所有求和结果累加在一起后, 处理进位, 即如果总和超过 16 位 (大于 0xFFFF), 则进行进位回卷, 将高位部分加到低位部分, 直到总和不再超过 16 位。最后, 将最终的 `sum` 进行取反操作, 得到最终的校验和。

```

1 uint16_t calculateChecksum(const PseudoHeader& pseudoHeader, const UDPHeader& udpHeader,
2     const char* data, int dataLength) {
3     uint32_t sum = 0;
4     // 伪首部求和
5     sum += (pseudoHeader.srcIP >> 16) & 0xFFFF; // 源 IP 高 16 位
6     sum += pseudoHeader.srcIP & 0xFFFF;         // 源 IP 低 16 位
7     sum += (pseudoHeader.destIP >> 16) & 0xFFFF; // 目的 IP 高 16 位
8     sum += pseudoHeader.destIP & 0xFFFF;         // 目的 IP 低 16 位
9     sum += (pseudoHeader.reserved << 8) + pseudoHeader.protocol;
10    // reserved 和 protocol 拼成 16 位
11    sum += pseudoHeader.udpLength; // UDP 长度
12    // UDP 头部求和
13    sum += ntohs(udpHeader.sourcePort); // 源端口
14    sum += ntohs(udpHeader.destPort);  // 目的端口
15    sum += (udpHeader.sequenceNumber >> 16) & 0xFFFF; // 序列号高 16 位
16    sum += udpHeader.sequenceNumber & 0xFFFF;         // 序列号低 16 位
17    sum += (udpHeader.acknowledgmentNumber >> 16) & 0xFFFF; // 确认号高 16 位
18    sum += udpHeader.acknowledgmentNumber & 0xFFFF;         // 确认号低 16 位
19    sum += (udpHeader.flags << 8) + udpHeader.reserved; // flags 和 reserved 拼成 16 位

```



```

20     sum += udpHeader.length;           // 长度字段
21     sum += udpHeader.windowSize;       // 窗口大小
22     sum += udpHeader.checksum;
23     // 数据部分求和
24     for (int i = 0; i < dataLength; i += 2) {
25         uint16_t word = (data[i] << 8) + (i + 1 < dataLength ? data[i + 1] : 0);
26         sum += word;
27     }
28     // 处理进位
29     while (sum > 0xFFFF) {
30         sum = (sum & 0xFFFF) + (sum >> 16);
31     }
32     // 取反，返回校验和
33     uint16_t checksum = ~static_cast<uint16_t>(sum);
34     //cout << "Final checksum: " << checksum << endl;
35     return checksum;
36 }

```

4.3 接收到数据包后进行差错检测和校验

当进行文件传输时，服务端需要根据数据包的源 IP 地址和目的 IP 地址重新构造出伪首部来对校验和字段进行校验，通过检验校验和的正确性来确定数据在传输过程中没有被篡改或丢失。如果得到的数据包校验和错误，即数据在传输过程中可能被篡改，则服务端会将该数据包丢弃，等待客户端进行重传。具体的代码实现如下所示：

```

1  cout << "===== 进行数据接收 =====" << endl;
2  cout << "[来自客户端] 收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: "
3      << recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum << endl;
4  char* data = buffer_fin + HEADER_SIZE;
5  int dataLength = bytesReceived - HEADER_SIZE;
6  if (checkheader(pseudorecvHeader, *recvHeader, data, bytesReceived - HEADER_SIZE) == true)
7  {
8      // 保存到文件
9      outFile.write(data, dataLength);
10     cout << "已写入数据块: " << dataLength << " 字节" << endl;
11     acknowledgmentNumber = recvHeader->sequenceNumber + 1;
12     UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort, ++sequenceNumber,
13         acknowledgmentNumber, ACK_FLAG, 0);
14     PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
15         clientAddr.sin_addr.s_addr, HEADER_SIZE);
16     vector<char> ackPacket = createPacket(ackHeader, nullptr, 0, ackPseudoHeader);
17     sendto(serverSocket, ackPacket.data(), ackPacket.size(), 0, (sockaddr*)&clientAddr,

```



```

18     clientAddrLen);
19     cout << "已发送 ACK 确认!" << endl;
20     cout << "[服务端] 发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
21         ackHeader.acknowledgmentNumber << ", 校验和: " << ackHeader.checksum << endl;
22     cout << "===== " << endl;
23 }

```

差错检测对校验和进行检验的逻辑如下所示，与计算校验和的逻辑类似，将伪首部和首部的各个字段相加，对于超过 16 位的字段进行拆分，不足 16 位的字段进行拼接或填充 0，然后与数据部分进行相加。在完成这些字段的求和后，接着处理进位，如果总和超出了 16 位（即大于 0xFFFF），就进行进位回卷，将高位加到低位上，直到所有进位被处理完毕。最后，我们根据计算的和进行校验。如果最终的和结果与 0xFFFF（即所有位为 1）匹配，则认为校验正确，返回 true；如果不匹配，则返回 false，表示校验失败。

```

1 bool checkheader(const PseudoHeader& pseudoHeader, const UDPHeader& udpHeader,
2     const char* data, int dataLength) {
3     uint32_t sum = 0;
4     // 伪首部求和
5     sum += (pseudoHeader.srcIP >> 16) & 0xFFFF; // 源 IP 高 16 位
6     sum += pseudoHeader.srcIP & 0xFFFF;         // 源 IP 低 16 位
7     sum += (pseudoHeader.destIP >> 16) & 0xFFFF; // 目的 IP 高 16 位
8     sum += pseudoHeader.destIP & 0xFFFF;         // 目的 IP 低 16 位
9     sum += (pseudoHeader.reserved << 8) + pseudoHeader.protocol;
10    // reserved 和 protocol 拼成 16 位
11    sum += pseudoHeader.udpLength; // UDP 长度
12    // UDP 头部求和
13    sum += ntohs(udpHeader.sourcePort); // 源端口
14    sum += ntohs(udpHeader.destPort);   // 目的端口
15    sum += (udpHeader.sequenceNumber >> 16) & 0xFFFF; // 序列号高 16 位
16    sum += udpHeader.sequenceNumber & 0xFFFF;         // 序列号低 16 位
17    sum += (udpHeader.acknowledgmentNumber >> 16) & 0xFFFF; // 确认号高 16 位
18    sum += udpHeader.acknowledgmentNumber & 0xFFFF;         // 确认号低 16 位
19    sum += (udpHeader.flags << 8) + udpHeader.reserved; // flags 和 reserved 拼成 16 位
20    sum += udpHeader.length; // 长度字段
21    sum += udpHeader.windowSize; // 窗口大小
22    sum += udpHeader.checksum; // 校验和字段
23    // 数据部分求和
24    for (int i = 0; i < dataLength; i += 2) {
25        uint16_t word = (data[i] << 8) + (i + 1 < dataLength ? data[i + 1] : 0);
26        sum += word;
27    }
28    // 处理进位

```

```

29     while (sum > 0xFFFF) {
30         sum = (sum & 0xFFFF) + (sum >> 16);
31     }
32     // 检查结果
33     if ((sum & 0xFFFF) == 0xFFFF) { // 所有位为 1 则校验正确
34         cout << "校验正确" << endl;
35         return true;
36     }
37     else {
38         cout << "校验错误" << endl;
39         return false;
40     }
41 }

```

4.4 文件传输过程中的超时重传和快速重传机制

文件传输过程中采取的超时重传、快速重传和限制重传次数等机制与建立连接时的三次握手基本类似，在发送数据包后，客户端会等待服务端返回 ACK 确认信号并查看首部的确认号是否匹配，匹配则继续发送下一个数据包，否则进行重传。重传超过三次后则会认为连接已断开，避免无限重传带来的资源浪费。由于这里的流程与前面基本类似，因此不再重复介绍，重点分析一下文件传输过程中客户端采用的停等机制。使用路由器进行丢包后，具体实现效果如下所示：

```

=====发送当前数据包=====
数据发送成功，等待ACK...
发送数据包的序列号：169，ACK号：169，校验和：51853
期望收到的ACK号：170
实际收到的ACK号：170
收到数据包的序列号：168，ACK号：170，校验和：5485
收到正确的ACK，数据传输成功！
当前数据包传输完毕

=====发送当前数据包=====
数据发送成功，等待ACK...
发送数据包的序列号：170，ACK号：170，校验和：47576
期望收到的ACK号：171
实际收到的ACK号：171
收到数据包的序列号：169，ACK号：171，校验和：5483
收到正确的ACK，数据传输成功！
当前数据包传输完毕

=====发送当前数据包=====
数据发送成功，等待ACK...
发送数据包的序列号：171，ACK号：171，校验和：22622
期望收到的ACK号：172
数据传输超时，没有收到ACK，进行数据重传...
=====发送当前数据包=====
数据发送成功，等待ACK...
发送数据包的序列号：171，ACK号：171，校验和：22622
期望收到的ACK号：172

[服务端]发送数据包的序列号：167，ACK号：169，校验和：5487
=====进行数据接收=====
[来自客户端]收到数据包的序列号：169，ACK号：169，校验和：51853
校验正确
已写入数据块：10240 字节
已发送ACK确认！
[服务端]发送数据包的序列号：168，ACK号：170，校验和：5485
=====进行数据接收=====
[来自客户端]收到数据包的序列号：170，ACK号：170，校验和：47576
校验正确
已写入数据块：10240 字节
已发送ACK确认！
[服务端]发送数据包的序列号：169，ACK号：171，校验和：5483
=====进行数据接收=====
[来自客户端]收到数据包的序列号：171，ACK号：171，校验和：22622
校验正确
已写入数据块：10240 字节
已发送ACK确认！
[服务端]发送数据包的序列号：170，ACK号：172，校验和：5481
=====进行数据接收=====
[来自客户端]收到数据包的序列号：172，ACK号：172，校验和：25251
校验正确
已写入数据块：10240 字节

```

图 4.7: 文件传输过程中的超时重传机制

4.5 客户端采取停等机制进行流量控制

在本次实验中，客户端采取了停等机制进行流量控制，确保数据的可靠传输。具体实现流程如下：客户端通过 sendDataAndWaitForAck 函数发送数据包并等待服务端的确认信号。首先，客户端会根据传入的参数构造伪首部、UDP 首部以及数据包内容，计算出校验和并将其封装成完整的数据包发送给服务端。在发送数据包后，客户端会进入一个等待 ACK 的状态，并通过 select 函数设置超时机制，等待服务端返回确认。

在等待 ACK 的过程中，客户端会检查接收到的 ACK 包的序列号和确认号是否符合预期。如果接收到的 ACK 号与期望的确认号匹配，表示数据传输成功，客户端返回 true，并继续发送下一个数据包。如果客户端在超时时间内没有收到 ACK，或者接收到的 ACK 包存在错误（例如 ACK 号不匹配），则会执行重传机制。客户端会记录连续错误的 ACK 数量，当错误 ACK 达到三次时，客户端认为可能是网络不稳定，准备重新发送数据包。如果数据包发送失败或超时的重试次数达到最大限制 (MAX_RETRIES)，则认为数据传输失败，终止操作并关闭套接字。通过这种方式，客户端可以在每次发送数据后，通过等待 ACK 确认的方式确保数据的可靠性。如果出现错误或超时，客户端会自动重传数据包，从而增强传输的可靠性和稳定性。

停等机制确保在每次发送数据时，发送方都要等待接收方确认收到数据后才会继续发送下一个数据。这种机制的关键特点是每次发送一个数据包后，发送方会停止发送，直到收到接收方的确认 (ACK)，然后才会继续发送下一个数据包。这种机制的优点是实现简单，但缺点是传输效率较低，特别是在高延迟或丢包率高的网络环境下，因为每个数据包的发送都必须等待确认才能进行下一步操作。因此，停等机制适用于较小的数据量和可靠的网络连接，而在进行大规模数据传输时，需要采用更加复杂的协议，如滑动窗口协议来提高效率，在后续实验中我们会采用滑动窗口机制进行优化。

停等机制的具体实现代码如下所示：

```

1 // 发送数据包并等待 ACK
2 bool sendDataAndWaitForAck(SOCKET clientSocket, const sockaddr_in& clientAddr,
3     const sockaddr_in& serverAddr, UDPHeader& header, const char* data, int dataLength) {
4     PseudoHeader pseudoHeader = createPseudoHeader(clientAddr.sin_addr.s_addr,
5         serverAddr.sin_addr.s_addr, HEADER_SIZE + dataLength);
6     vector<char> packet = createPacket(header, data, dataLength, pseudoHeader);
7     int retries = 0;
8     int wrongAckCount = 0; // 记录连续收到的错误 ACK 数量
9     bool three_wrong = false;
10    bool reach_time = false;
11    while (retries < MAX_RETRIES) {
12        // 重置重试和错误 ACK 计数
13        wrongAckCount = 0;
14        three_wrong = false;
15        reach_time = false;
16        // 开始计时
17        auto startTime = chrono::steady_clock::now();
18        // 发送数据包
19        if (sendto(clientSocket, packet.data(), packet.size(), 0, (sockaddr*)&serverAddr,
20            sizeof(serverAddr)) == SOCKET_ERROR) {
21            cerr << "发送数据包失败!" << endl;
22            return false;
23        }
24        cout << "===== 发送当前数据包 =====" << endl;
25        cout << "数据发送成功，等待 ACK..." << endl;
26        cout << "发送数据包的序列号：" << header.sequenceNumber << ", ACK 号：" <<

```

```

27     header.acknowledgmentNumber << ", 校验和: " << header.checksum << endl;
28     uint32_t expectedAckNumber = header.sequenceNumber + 1;
29     cout << "期望收到的 ACK 号: " << expectedAckNumber << endl;
30     // 使用 select 进行超时判断
31     fd_set readfds;
32     struct timeval timeout;
33     timeout.tv_sec = 0;
34     timeout.tv_usec = TIMEOUT_MS * 1000; // 设置超时时间
35     while (true) {
36         FD_ZERO(&readfds);
37         FD_SET(clientSocket, &readfds);
38         // 等待接收或超时
39         int result = select(0, &readfds, nullptr, nullptr, &timeout);
40         if (result > 0) {
41             // 如果 select 返回大于 0, 表示有数据到达
42             if (FD_ISSET(clientSocket, &readfds)) {
43                 char buffer[BUFFER_SIZE] = {};
44                 sockaddr_in recvAddr = {};
45                 int recvAddrLen = sizeof(recvAddr);
46                 int bytesReceived = recvfrom(clientSocket, buffer, BUFFER_SIZE, 0,
47                     (sockaddr*)&recvAddr, &recvAddrLen);
48                 if (bytesReceived > 0) {
49                     UDPHeader* recvHeader = (UDPHeader*)buffer;
50                     cout << "实际收到的 ACK 号: " << recvHeader->acknowledgmentNumber
51                         << endl;
52                     cout << "收到数据包的序列号: " << recvHeader->sequenceNumber <<
53                         ", ACK 号: " << recvHeader->acknowledgmentNumber << ", 校验和: " <<
54                         recvHeader->checksum << endl;
55                     // 检查是否收到正确的 ACK
56                     if (recvHeader->flags == ACK_FLAG && recvHeader->acknowledgmentNumber
57                         == expectedAckNumber) {
58                         cout << "收到正确的 ACK, 数据传输成功!" << endl;
59                         return true; // 收到正确的 ACK, 数据传输成功
60                     }
61                     else {
62                         wrongAckCount++;
63                         if (wrongAckCount >= 3) {
64                             setConsoleColor(12);
65                             cerr << "连续收到 3 个错误的 ACK, 准备重传数据包!" << endl;
66                             cout << "进行数据重传..." << endl;
67                             setConsoleColor(7);
68                             three_wrong = true;

```

```

69         break; // 进入重传逻辑
70     }
71 }
72 }
73 }
74 }
75 else if (result == 0) {
76     // 如果 select 返回 0, 表示超时
77     setConsoleColor(12);
78     cout << "数据传输超时, 没有收到 ACK, 进行数据重传..." << endl;
79     setConsoleColor(7);
80     reach_time = true;
81     retries++;
82     if (retries >= MAX_RETRIES) {
83         setConsoleColor(12);
84         cerr << "达到最大重试次数, 数据传输失败, 无法建立连接。" << endl;
85         setConsoleColor(7);
86         closesocket(clientSocket);
87         WSACleanup();
88         return false;
89     }
90     break; // 退出等待 ACK 的循环
91 }
92 // 如果遇到错误 ACK 或超时, 重新发送数据包
93 if (three_wrong reach_time) {
94     break;
95 }
96 }
97 }
98 return false; // 如果所有重试都失败, 返回 false
99 }

```

5 三次挥手释放连接

5.1 三次挥手的基本流程

由于本实验中要求实现单向传输, 即只会从客户端向服务端传输文件, 因此我们只需要三次挥手即可完成连接的释放。前面说到, 当客户端完成文件的传输后会向服务器发送一个包含 FILE_END 标志位的数据报, 服务端接收到这个数据报后会回复 ACK 信号。而当客户端接收到服务端的回复后, 此时当前文件传输结束。如果用户没有需要继续传输的文件, 就会输入 q 进入三次挥手过程, 进行连接的释放。流程如下: 首先客户端会发送一个带有 FIN 标志位的数据报, 表示客户端已经没有文件需要传输了, 希望断开连接。当服务端接收到这个数据报后, 会回复一个带有 FIN 标志位和 ACK 标志位

的数据报，表示同意断开连接，同时由于服务端不需要向客户端传输文件，因此服务端回复带有 FIN 标志位和 ACK 标志位的数据报，表示服务端也希望断开连接。当客户端收到服务端的回复信号后，三次挥手结束，连接被成功释放。具体实现效果如下图所示：

```
=====发送当前数据包=====
数据发送成功，等待ACK...
发送数据包的序列号：2099，ACK号：2099，校验和：32985
期望收到的ACK号：2100
实际收到的ACK号：2100
收到数据包的序列号：2098，ACK号：2100，校验和：1625
收到正确的ACK，数据传输成功！
文件传输完毕标志传输完毕
=====
请输入要上传的文件路径（输入q释放连接）：q
=====三次挥手释放连接=====
开始进行三次挥手，准备发送FIN包
尝试第一次挥手释放连接...
数据发送成功，等待FIN ACK...
发送数据包的序列号：2100，ACK号：2100，校验和：33751
期望收到的ACK号：2101
实际收到的ACK号：2101
收到数据包的序列号：2099，ACK号：2101，校验和：599
收到正确的FIN ACK，第二次挥手成功！
开始尝试进行第三次挥手
第三次挥手发送的序列号：2101，ACK号：2100，校验和：28907
第三次挥手成功，成功释放连接
成功释放连接
连接已关闭！
=====
[服务端]发送数据包的序列号：2096，ACK号：2098，校验和：1629
=====进行数据接收=====
[来自客户端]收到数据包的序列号：2098，ACK号：2098，校验和：43522
校验正确
已写入数据块：7168 字节
已发送ACK确认！
[服务端]发送数据包的序列号：2097，ACK号：2099，校验和：1627
=====
接收到FILE_END，准备发送ACK...
[来自客户端]收到数据包的序列号：2099，ACK号：2099，校验和：32985
[服务端]发送数据包的序列号：2098，ACK号：2100，校验和：1625
已发送ACK，本次文件接收完毕
接收到FIN，准备发送FIN ACK...
[服务端]收到数据包的序列号：2100，ACK号：2100，校验和：33751
尝试第二次挥手释放连接...
数据发送成功，等待ACK...
发送数据包的序列号：2099，ACK号：2101，校验和：599
期望收到的ACK号：2100
实际收到的ACK号：2100
[服务端]收到数据包的序列号：2101，ACK号：2100，校验和：28907
收到正确的ACK，第三次挥手成功！
成功释放连接
关闭文件流，准备接收下一个用户的连接
```

图 5.8: 三次挥手的具体实现效果

5.2 接收到数据包后进行差错检测和校验

与前面的三次握手与文件传输类似，三次挥手释放连接时客户端和服务端在接收到数据包之后也会进行差错检测和校验，以确保数据传输的可靠性和准确性。同时也会采取超时重传和快速重传机制，以确保数据传输的效率。具体实现效果如下图所示：

```
Microsoft Visual Studio 调试控制台
发送数据包的序列号：2099，ACK号：2099，校验和：32985
期望收到的ACK号：2100
实际收到的ACK号：2100
收到数据包的序列号：4197，ACK号：2100，校验和：65061
收到正确的ACK，数据传输成功！
文件传输完毕标志传输完毕
=====
请输入要上传的文件路径（输入q释放连接）：q
=====三次挥手释放连接=====
开始进行三次挥手，准备发送FIN包
尝试第一次挥手释放连接...
数据发送成功，等待FIN ACK...
发送数据包的序列号：2100，ACK号：2100，校验和：33751
期望收到的ACK号：2101
数据传输超时，没有收到ack，进行数据重传...
数据发送成功，等待FIN ACK...
发送数据包的序列号：2100，ACK号：2100，校验和：33751
期望收到的ACK号：2101
数据传输超时，没有收到ack，进行数据重传...
数据发送成功，等待FIN ACK...
发送数据包的序列号：2100，ACK号：2100，校验和：33751
期望收到的ACK号：2101
数据传输超时，没有收到ack，进行数据重传...
达到最大重试次数，握手失败，服务器当前无法建立连接。
连接已关闭！
=====
```

图 5.9: 三次挥手中的超时重传

当三次挥手释放连接后客户端会退出，但服务端会继续保持工作状态，继续等待下一次或下一个用户进行三次握手建立连接，如下图所示：

```

D:\vs2022\server\x64\Debug\server.exe
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 2097, ACK号: 2097, 校验和: 10565
校验正确
已写入数据块: 10240 字节
已发送ACK确认!
[服务端]发送数据包的序列号: 2096, ACK号: 2098, 校验和: 1629
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 2098, ACK号: 2098, 校验和: 47657
校验正确
已写入数据块: 3913 字节
已发送ACK确认!
[服务端]发送数据包的序列号: 2097, ACK号: 2099, 校验和: 1627
=====
接收到FILE_END, 准备发送ACK...
[来自客户端]收到数据包的序列号: 2099, ACK号: 2099, 校验和: 32985
[服务端]发送数据包的序列号: 2098, ACK号: 2100, 校验和: 1625
已发送ACK, 本次文件接收完毕
接收到FIN, 准备发送FIN ACK...
[服务端]收到数据包的序列号: 2100, ACK号: 2100, 校验和: 33751
尝试第二次挥手释放连接...
数据发送成功, 等待ACK...
发送数据包的序列号: 2099, ACK号: 2101, 校验和: 599
期望收到的ACK号: 2100
实际收到的ACK号: 2100
[服务端]收到数据包的序列号: 2101, ACK号: 2100, 校验和: 28907
收到正确的ACK, 第三次挥手成功!
成功释放连接
关闭文件流, 准备接收下一个用户的连接

```

图 5.10: 服务端等待下一个用户进行连接

6 文件传输效果和性能测试指标

本实验中要求使用传输时间和吞吐率作为性能测试指标，在不使用路由器进行丢包和延时的前提下，我对四个测试文件的传输性能进行了测试，如下图所示：

<pre> 文件传输完毕! 文件大小: 1.77131 MB 传输时间: 1.24355 秒 吞吐率: 1.4244 MB/s ===== --发送文件传输完毕标志-- 开始发送文件传输完毕标志 --发送当前数据包-- 数据发送成功, 等待ACK... 发送数据包的序列号: 185, ACK号: 185, 校验和: 36813 期望收到的ACK号: 186 实际收到的ACK号: 186 收到数据包的序列号: 2283, ACK号: 186, 校验和: 3354 收到正确的ACK, 数据传输成功! 文件传输完毕标志传输完毕 ===== 请输入要上传的文件路径(输入q释放连接): </pre>	<pre> 文件传输完毕! 文件大小: 1.5791 MB 传输时间: 0.392918 秒 吞吐率: 4.01891 MB/s ===== --发送文件传输完毕标志-- 开始发送文件传输完毕标志 --发送当前数据包-- 数据发送成功, 等待ACK... 发送数据包的序列号: 2099, ACK号: 2099, 校验和: 32985 期望收到的ACK号: 2100 实际收到的ACK号: 2100 收到数据包的序列号: 4197, ACK号: 2100, 校验和: 65061 收到正确的ACK, 数据传输成功! 文件传输完毕标志传输完毕 ===== 请输入要上传的文件路径(输入q释放连接): </pre>
<pre> 文件传输完毕! 文件大小: 5.62525 MB 传输时间: 3.96978 秒 吞吐率: 1.41702 MB/s ===== --发送文件传输完毕标志-- 开始发送文件传输完毕标志 --发送当前数据包-- 数据发送成功, 等待ACK... 发送数据包的序列号: 764, ACK号: 764, 校验和: 35655 期望收到的ACK号: 765 实际收到的ACK号: 765 收到数据包的序列号: 2862, ACK号: 765, 校验和: 2196 收到正确的ACK, 数据传输成功! 文件传输完毕标志传输完毕 ===== 请输入要上传的文件路径(输入q释放连接): d:/3_.jpg </pre>	<pre> 文件传输完毕! 文件大小: 11.4145 MB 传输时间: 7.08959 秒 吞吐率: 1.61004 MB/s ===== --发送文件传输完毕标志-- 开始发送文件传输完毕标志 --发送当前数据包-- 数据发送成功, 等待ACK... 发送数据包的序列号: 1935, ACK号: 1935, 校验和: 33313 期望收到的ACK号: 1936 实际收到的ACK号: 1936 收到数据包的序列号: 4033, ACK号: 1936, 校验和: 65389 收到正确的ACK, 数据传输成功! 文件传输完毕标志传输完毕 ===== 请输入要上传的文件路径(输入q释放连接): </pre>

图 6.11: 测试文件的传输性能对比

对比四个测试文件的传输性能可以发现，UDP 传输对于 txt 文本文件的传输速率和吞吐率都要

大于其他的图片文件。这一差异的主要原因在于，文本文件相对较小且数据结构简单，UDP 协议能够在较短时间内进行快速传输。而由于我们测试时选择的图片较小并且本实验采取的是停等机制进行流量控制，只有当客户端收到服务端回复的 ACK 确认信号时才继续进行传输，因此不同大小的图片传输吞吐率相差不大，且传输速率都较慢，后续实验中我们会采用滑动窗口机制来进行优化。

7 实验内容总结

本次实验通过设计并实现一个基于 UDP 协议的可靠传输协议，成功模拟了三次握手建立连接和三次挥手释放连接的过程，并通过超时重传、快速重传、流量控制、差错检测与校验等机制保障了数据传输的效率和可靠性。主要包括以下几个方面：

- 报文设计与传输：为了实现可靠的数据传输，我们重新设计了 UDP 报文格式，包括添加了序列号、确认号、标志位等字段，这些设计有效支持了建立连接和可靠传输的需求。
- 三次握手与超时重传机制：我们采用了三次握手协议来确保客户端和服务端之间可靠建立连接。通过超时重传机制，客户端能够在指定时间内未收到确认包时重新发送数据包，从而避免数据丢失和连接失败。快速重传机制进一步优化了传输效率。
- 快速重传机制：快速重传是针对数据包丢失后，提高传输效率的关键技术。在本实验中，当客户端接收到多个重复的非预期的确认包时，即表明某个数据包已经丢失，客户端会立即重新发送丢失的数据包，而不等待超时重传。这一机制大大减少了因单个包丢失导致的延迟，避免了客户端长时间等待超时的情况，从而提高了协议的传输效率。
- 流量控制：在文件传输过程中，采用了停等机制进行流量控制，确保了数据传输的有序性，有效避免了丢包和拥塞问题，一定程度上保证了数据传输的可靠性。
- 差错检测与校验：每次传输的数据包都进行了差错检测，保证了数据的完整性和准确性。通过校验和的使用，能够有效防止数据在传输过程中被篡改或丢失。
- 性能测试与评估：实验中通过吞吐率和文件传输时延的测试评估了系统的性能。结果表明，结构简单且数据量较小的文本文件（txt）传输速率和吞吐率显著高于图片文件，表明协议在处理小文件时效率更高。

同时在实现过程中，我注意到以下几个关键点：

- 停等机制虽然能够保证数据传输的可靠性，但在效率上存在明显的瓶颈，尤其是在高延迟网络下，客户端每次发送数据后都需等待确认包才能继续发送下一批数据，严重限制了系统的吞吐量。为此，后续实验计划使用滑动窗口机制，以提高流量控制的效率和整体传输速度。
- 传统阻塞模型中，如果客户端没有接收到 ACK 信号，会被阻塞在等待状态，导致时钟无法继续计时，从而无法触发超时重传机制。为了解决这个问题，我们采用了 select 系统调用进行超时检测，这使得客户端在等待确认包的同时能够继续执行其他任务。该方法不仅提高了程序响应效率，还确保在确认包丢失或网络不稳定时能够及时重传，从而增强了协议的可靠性和稳定性。

通过本次实验，我深刻理解了可靠传输协议的实现原理，并学会了如何通过协议设计和机制优化提升数据传输的可靠性和效率。此外，实验还让我意识到流量控制和超时机制在保证数据传输可靠性和效率方面的重要性，并为后续的网络协议优化和改进打下了基础。