南開大學
Nankai University

计算机学院

计算机网络实验报告

实验 3.2：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 12 月 5 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 接收到数据包后进行差错检测和校验 4
3.3 三次握手传输过程中的超时重传和快速重传机制 4

4 文件传输：采用 GBN 机制进行流量控制 **5**
4.1 文件传输的基本流程 5
4.2 报文传输前数据包的构建 8
4.3 发送端采取 GBN 机制进行流量控制 9
4.4 接收到数据包后进行差错检测和校验 15
4.5 文件传输过程中的超时重传和快速重传机制 16

5 三次挥手释放连接 **17**
5.1 三次挥手的基本流程 17
5.2 接收到数据包后进行差错检测和校验 17

6 文件传输效果和性能测试指标 **18**
6.1 停等机制与 GBN 机制文件传输性能对比 18
6.2 丢包率为 5% 时停等机制与 GBN 机制文件传输性能对比 20
6.3 延时为 10ms 时停等机制与 GBN 机制文件传输性能对比 21
6.4 不同延时情况下停等机制与 GBN 机制文件传输性能对比 22
6.5 不同丢包率情况下停等机制与 GBN 机制文件传输性能对比 23
6.6 不同传输机制的横向对比 24

7 实验内容总结 **25**

1 实验要求

在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认，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

在本次实验中，我们需要用到窗口大小这个数据段，而在 3-1 实验中我们将窗口大小恒定置为了 1，因此首先简要的回顾一下实验 3-1 我们设计的报文格式。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，即发送端和接收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，只用于接收端和发送端进行数据校验**，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**。

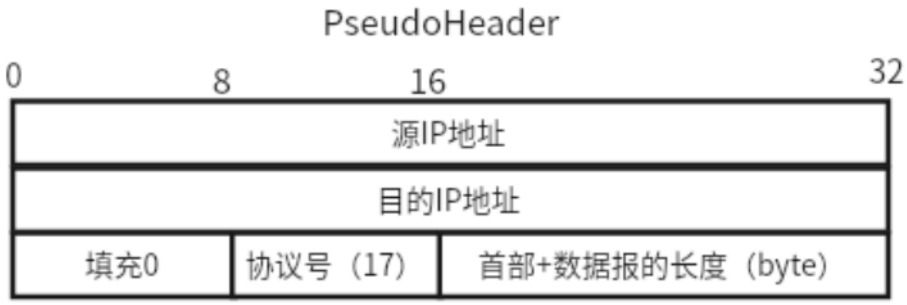

图 2.1: 伪首部报文格式

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 位校验和用于接收端和发送端进行数据校验，16 位窗口大小，**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，用于通知接收端当前文件传输完毕，接收端可以继续准备接收下一个文件。

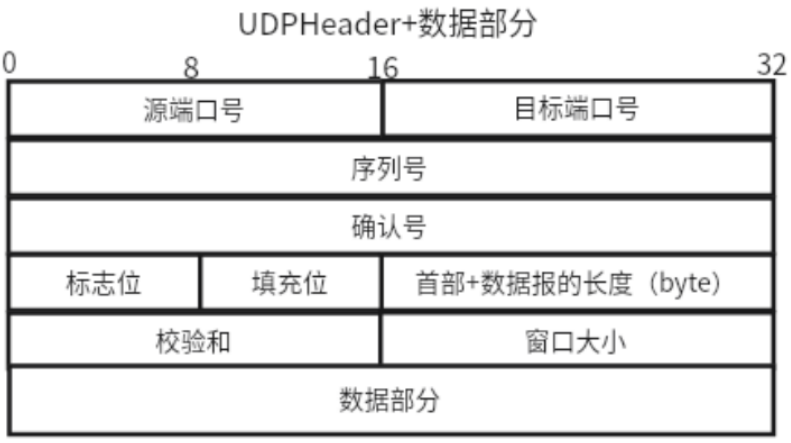

图 2.2: 首部报文格式

3 三次握手建立连接

GitHub 链接：https://github.com/lhz191/computer_network.git

由于实验 3-2 中三次握手建立连接的过程与实验 3-1 相同，因此不再进行具体的实现代码展示，只是简单回顾一下我们实现的效果。重点讲解文件传输中采用的 GBN 流量控制机制。

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，即当发送端希望与接收端建立连接时，会尝试与接收端进行三次握手，确保双方能够成功建立连接**。发送端会首先向接收端发送带有 SYN 标志位的数据报，表示希望建立连接，接收端在接收到发送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，同时也希望与发送端建立连接，最后发送端回复 ACK 确认信号，完成连接的建立。**具体实现效果如下图所示：**

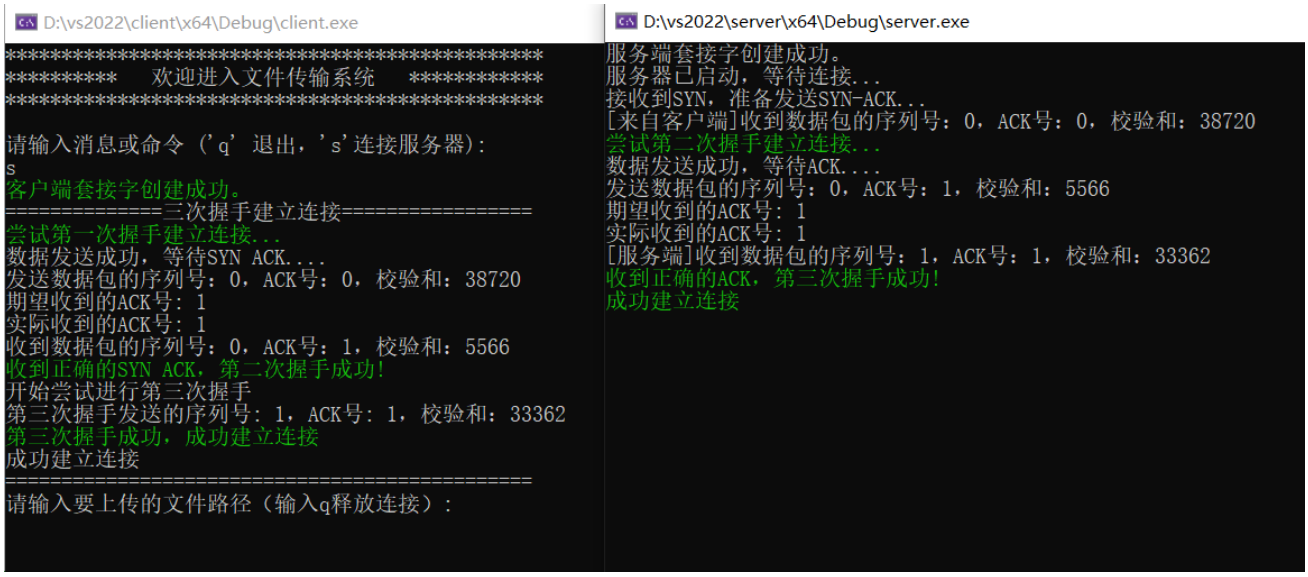

图 3.3: 三次握手的实现效果

3.2 接收到数据包后进行差错检测和校验

在发送端接收到来自对方的 SYN、ACK 数据包时，**需要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。**而当接收端接收到发送端返回的 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，接收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，则接收端会将该数据包丢弃，等待发送端进行重传。

3.3 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**，当发送端向接收端发送 SYN 信号或数据包，或者接收端向发送端发送 SYN、ACK 信号时，会**使用 select 系统调用来进行超时检测**。如果发送端在预设的时间内没有收到接收端的确认包，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（代码里是检测发送端的 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发送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，接着发送端会接收回复的 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，发送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，发送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，发送端会记录超时情况下当前数据包重传的次数，如果在超时后发送端重传某个数据包的次数超过了三次，则会认为当前接收端无法建立连接，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，并确保发送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，当接收端向发送端发送 SYN、ACK 信号并等待发送端确认信号回复时，也会采取上述机制。

- 同时，为了提高传输的效率，发送端还实现了**快速重传机制**。**当发送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，发送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，当接收端向发送端发送 SYN、ACK 信号并等待发送端确认信号回复时，也会采取上述机制。

  具体实现效果如下图所示：

图 3.4: 超时重传和避免无限重传

4 文件传输：采用 GBN 机制进行流量控制

4.1 文件传输的基本流程

当发送端需要进行文件传输时，首先，发送端会向接收端发送传输的文件名称，也就是说接收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，接收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，接收端才会将 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，接收端会对发送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。一旦接收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，接收端会向发送端返回一个 ACK 确认包，通知发送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，用于确保发送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。**由于本实验中固定接收端的滑动窗口大小为 1，并且采取累积确认的方式，也就是说接收端每次接收时都期望得到自己想要的数据包，对于其他的数据包不进行接收。因此我们可以使用一个变量 wanted 记录接收端期望得到的数据包**，只有当接收端接收的数据包的序列号等于 wanted 并且校验和检验正确时，接收端才进行接收并将得到的数据写入文件，同时更新 wanted，然后向发送端发送 ack 确认号。当接收端接收到数据包的序列号不等于 wanted 或者校验和检验出错时，则向发送端发送累积确认的 ack 号，即接收端希望得到的 ack 号。当进行文件名接收时，将 wanted 进行初始化，避免影响到下一次的文件接收。

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;
```

```
    cout << "收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: " <<
        recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum << endl;
    // 创建文件路径，并打开文件
    string filePath = "D:/new/" + fileName;  // 拼接文件路径
    outFile.open(filePath, ios::binary);  // 使用文件路径创建文件
    if (!outFile) {
        cerr << "无法创建文件: " << filePath << endl;
        return;
    }
    flag=1;
    char* data = buffer_fin + HEADER_SIZE;
    int dataLength = bytesReceived - HEADER_SIZE;
    if (checkheader(pseudorecvHeader, *recvHeader, data, bytesReceived - HEADER_SIZE) == true)
    {
        acknowledgmentNumber = recvHeader->sequenceNumber + 1;
        UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort,
        ++sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
        PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
        clientAddr.sin_addr.s_addr, HEADER_SIZE);
        vector<char> ackPacket = createPacket(ackHeader, nullptr, 0, ackPseudoHeader);
        sendto(serverSocket, ackPacket.data(), ackPacket.size(), 0, (sockaddr*)&clientAddr,
                clientAddrLen);
        cout << "已发送 ACK 确认!" << endl;
        wanted = ackHeader.acknowledgmentNumber;
        cout << "发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
            ackHeader.acknowledgmentNumber << ", 校验和: " << ackHeader.checksum << endl;
    }
    continue;
}
else
{
    cout << "================== 进行数据接收 ================" << endl;
    cout << "[来自客户端] 收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: " <<
    recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum << endl;
    char* data = buffer_fin + HEADER_SIZE;
    int dataLength = bytesReceived - HEADER_SIZE;
    if (recvHeader->sequenceNumber == wanted && checkheader(pseudorecvHeader, *recvHeader
            , data, bytesReceived - HEADER_SIZE) == true)
    {
        // 保存到文件
        outFile.write(data, dataLength);
        cout << "已写入数据块: " << dataLength << " 字节" << endl;
```

```
48
49          acknowledgmentNumber = recvHeader->sequenceNumber + 1;
50          UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort,
51                  ++sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
52          PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
53                  clientAddr.sin_addr.s_addr, HEADER_SIZE);
54          vector<char> ackPacket = createPacket(ackHeader, nullptr, 0, ackPseudoHeader);
55          sendto(serverSocket, ackPacket.data(), ackPacket.size(), 0, (sockaddr*)&clientAddr
56                      , clientAddrLen);
57          cout << "已发送 ACK 确认!" << endl;
58          wanted = ackHeader.acknowledgmentNumber;
59          cout << "[服务端] 发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
60                  ackHeader.acknowledgmentNumber << ", 校验和: " << ackHeader.checksum << endl;
61          cout << "================================================" << endl;
62      }
63      else if (recvHeader->sequenceNumber != wanted  checkheader(pseudorecvHeader, *recvHeader,
64          data, bytesReceived - HEADER_SIZE) == false)
65      {
66          acknowledgmentNumber = wanted;
67          UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort,
68                  ++sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
69          PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
70                  clientAddr.sin_addr.s_addr, HEADER_SIZE);
71          vector<char> ackPacket = createPacket(ackHeader, nullptr, 0, ackPseudoHeader);
72          sendto(serverSocket, ackPacket.data(), ackPacket.size(), 0, (sockaddr*)&clientAddr
73                      , clientAddrLen);
74          cout << "已重新发送 ACK 确认!" << endl;
75          cout << "[服务端] 发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
76                  ackHeader.acknowledgmentNumber << ", 校验和: "
77                  << ackHeader.checksum << endl;
78          cout << "================================================" << endl;
79      }
80  }
```

当文件传输完毕后，发送端会发送一个带有 FILE_END 标志的数据报，用来告知接收端当前文件已经传输完毕。而接收端在接收到这个报文后，会回复一个带有 ACK 标志的数据报，作为对发送端的确认响应并告知发送端文件已成功接收并且完整无误。**接着接收端会关闭文件流，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。

采取 GBN 机制进行流量控制的具体实现效果如下图所示：

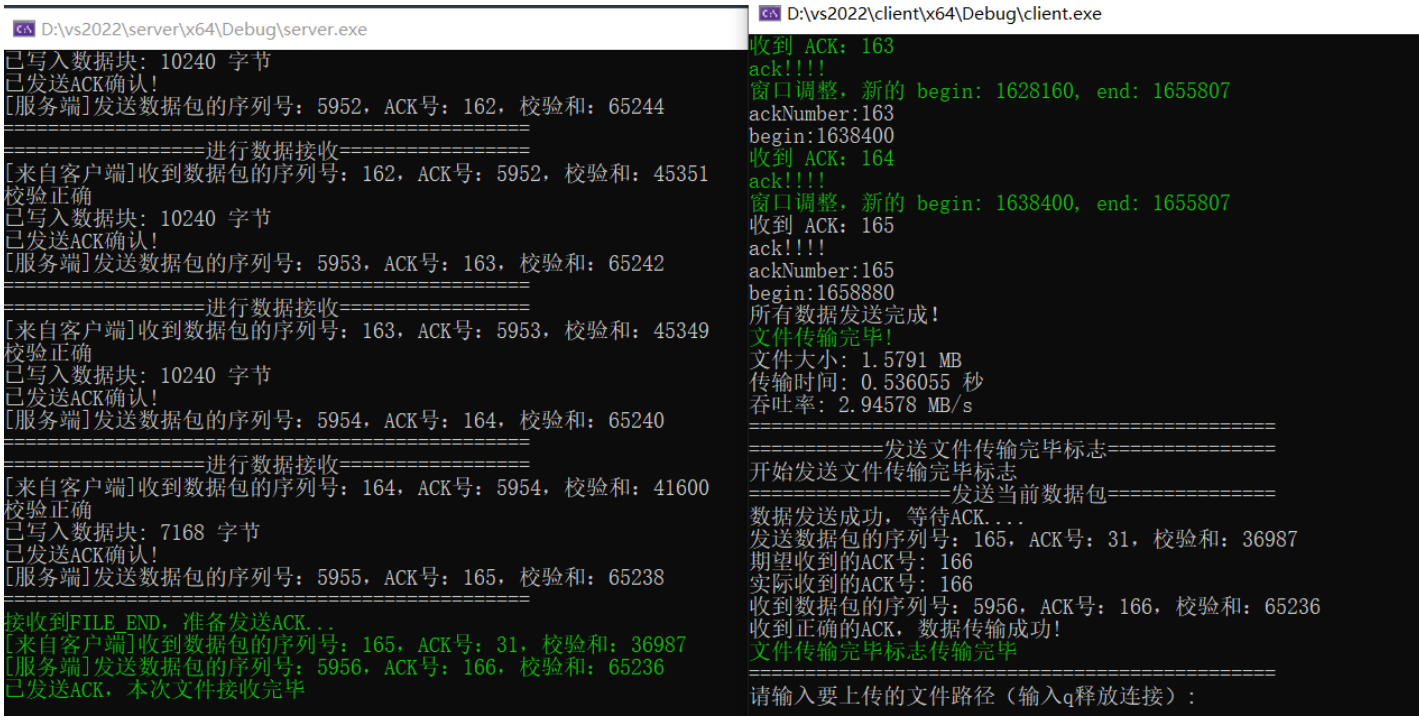

图 4.5: 文件传输的实现效果

从图中可以看出，相比实验 3-1 中采取的停等机制，采取 GBN 机制进行流量控制的文件传输速率和吞吐率均得到了提高，并且当增加传输文件大小时，传输速率和吞吐率均有进一步提高，这是因为滑动窗口机制的引入，允许在传输过程中同时发送多个数据包，从而减少了等待时间并提高了并发度，使得整体传输效率大幅提升。与传统的停等机制相比，GBN 协议能够通过合理的快速重传机制和滑动窗口控制，实现更高效的数据传输和更快的响应速度。

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，发送端在进行文件传输时，首先需要利用传入的发送端源 IP 地址和接收端目的 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;
```

```
    // 伪首部求和
    sum += (pseudoHeader.srcIP >> 16) & 0xFFFF; // 源 IP 高 16 位
    sum += pseudoHeader.srcIP & 0xFFFF;         // 源 IP 低 16 位
    sum += (pseudoHeader.destIP >> 16) & 0xFFFF; // 目的 IP 高 16 位
    sum += pseudoHeader.destIP & 0xFFFF;         // 目的 IP 低 16 位
    sum += (pseudoHeader.reserved << 8) + pseudoHeader.protocol;
    // reserved 和 protocol 拼成 16 位
    sum += pseudoHeader.udpLength; // UDP 长度
    // UDP 头部求和
    sum += ntohs(udpHeader.sourcePort);  // 源端口
    sum += ntohs(udpHeader.destPort);    // 目的端口
    sum += (udpHeader.sequenceNumber >> 16) & 0xFFFF; // 序列号高 16 位
    sum += udpHeader.sequenceNumber & 0xFFFF;         // 序列号低 16 位
    sum += (udpHeader.acknowledgmentNumber >> 16) & 0xFFFF; // 确认号高 16 位
    sum += udpHeader.acknowledgmentNumber & 0xFFFF;         // 确认号低 16 位
    sum += (udpHeader.flags << 8) + udpHeader.reserved; // flags 和 reserved 拼成 16 位
    sum += udpHeader.length;        // 长度字段
    sum += udpHeader.windowSize;    // 窗口大小
    sum += udpHeader.checksum;
    // 数据部分求和
    for (int i = 0; i < dataLength; i += 2) {
        uint16_t word = (data[i] << 8) + (i + 1 < dataLength ? data[i + 1] : 0);
        sum += word;
    }
    // 处理进位
    while (sum > 0xFFFF) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // 取反，返回校验和
    uint16_t checksum = ~static_cast<uint16_t>(sum);
    //cout << "Final checksum: " << checksum << endl;
    return checksum;
}
```

4.3 发送端采取 GBN 机制进行流量控制

本实验中我们使用 GBN 机制来进行流量控制，具体来说我使用了三个指针来维护一个发送窗口，分别是 begin，middle 和 end。begin 指向当前发送窗口的首个字节，end 指向当前发送窗口的最后一个字节，middle 指向当前还没有进行发送的首个字节。begin 之前代表我们已经发送并且已经收到 ack 确认信号的数据包，begin 和 middle 之间代表我们已发送但是还没有接收到确认的数据包，middle 和 end 之间代表我们需要进行发送但是还没有进行发送的数据包。end 之后代表不在当前发送窗口当前不需要进行发送的数据包，如下图所示。

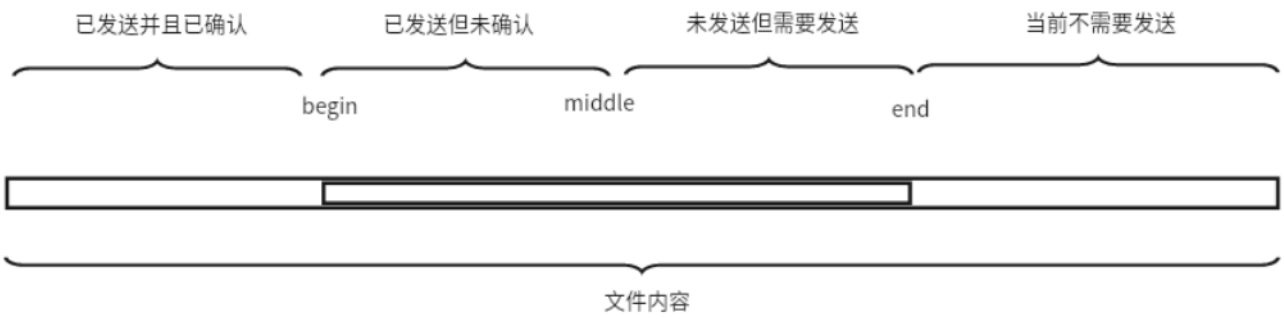

图 4.6: 采用滑动窗口机制进行数据发送

发送端发送线程的具体实现代码如下所示：只要 begin 指针小于 fileSize，即文件没有传输完毕，循环就一直执行，如果 middle 小于 end，即当前窗口还有需要进行发送的数据包，就构造首部，伪首部并计算校验和，利用 createPacket 函数组装成数据包并进行发送。一旦接收到接收端发来的 ack 确认信号后，就将滑动窗口向前移动，具体逻辑是 middle 不动，begin 移动到接收到的确认号所指示的位置，指向最新的已发送但是未收到确认信号的数据包，同时 end 根据我们设定的窗口大小向后移动。middle 只有当发生超时触发超时重传机制或数据包丢失的时候才进行移动，移动到 begin 的位置，表示当前窗口内的所有数据包都需要进行重新发送。同时为了提高数据传输的效率，我们设置了一个快速重传机制。使用一个 ackHistory 的循环数组不断记录最近三次收到的 ack 确认号，如果连续三次收到的 ack 确认号相同，则判定数据包已经丢失，触发快速重传机制，将 middle 移动到 begin 的位置，重发所有数据包。

同时我们需要设置一个接收线程不断接收来自接收端的 ack 确认信号，以确保滑动窗口可以及时向前滑动，不断更新需要发送的数据包。并且在收到 ack 确认信号后可以及时通知发送线程进行窗口位置的更新。同时接收线程还需要检测超时情况的发生：当一段时间内没有收到 ack 信号时，则认为此时超时情况发生，将 is_timeout 信号置为 true 并通知发送线程进行超时重传。

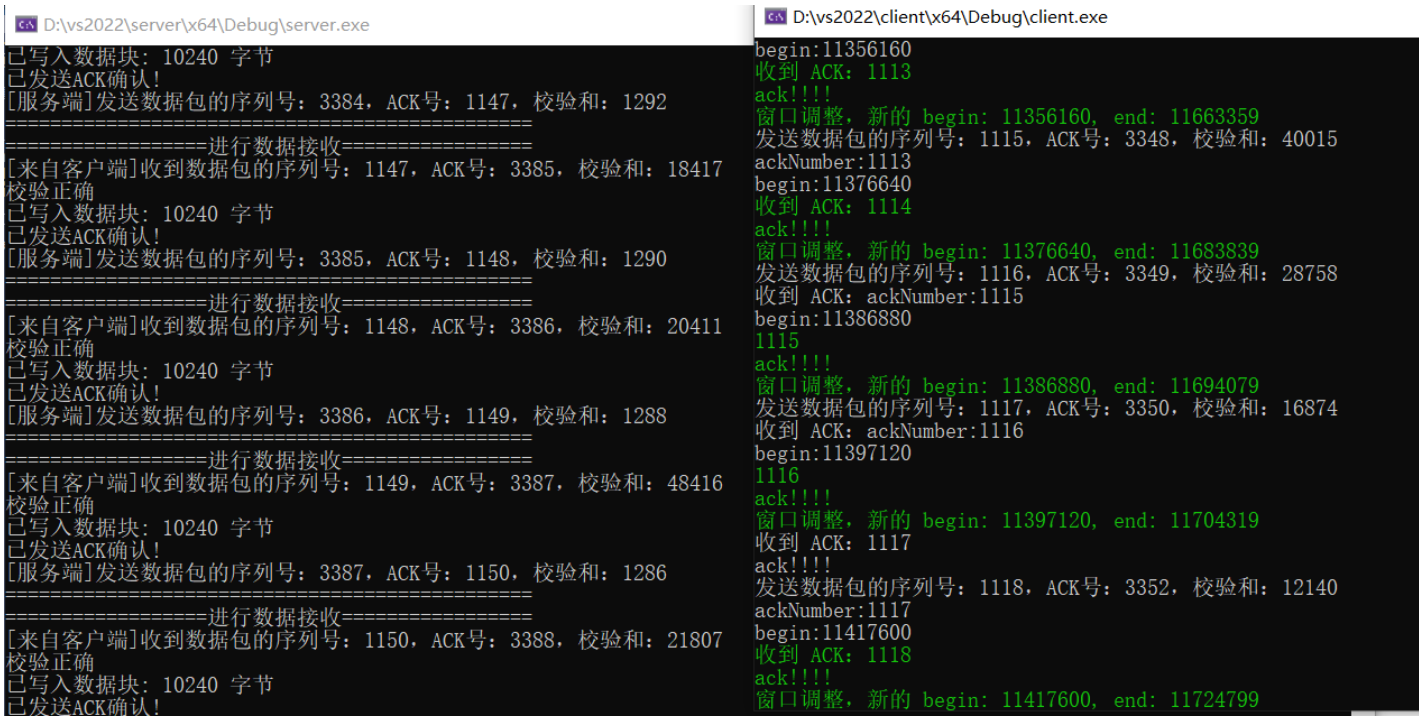

图 4.7: 采用 GBN 机制进行流量控制

采取 GBN 机制进行流量控制的文件传输效果如上图所示，可以看出，在理想情况下，发送端和接收端几乎是并行工作的。当发送端发送一个数据包后，接收端在短时间内收到数据包并立即发送回相应的 ack 确认信号。由于本机进行传输，RTT 往返时延较短，接收端可以在非常短的时间内确认数据包，发送端接收到 ack 确认信号，接收窗口前移，并且继续发送下一个数据包。这种机制确保了发送端和接收端的数据交换几乎没有阻塞，提升了传输效率。通过这种流量控制方式，GBN 机制能够在保证数据可靠传输的同时，提高了传输效率，避免了单一的等待 ack 确认机制造成的低效率和延迟。

```
#include <random>
#include <chrono>
#include <thread>
// 定义丢包率和延时
const int PACKET_DROP_RATE = 5; // 每 100 个包丢掉 PACKET_DROP_RATE 个
const int PACKET_DELAY_MS = 0;  // 每发一个包延时 PACKET_DELAY_MS 毫秒
bool sendData(int clientSocket, const sockaddr_in& clientAddr, const sockaddr_in& serverAddr,
std::vector<char>& fileContent, long fileSize, int size, uint16_t sourcePort, uint16_t destPort,
  uint32_t& sequenceNumber, uint32_t& acknowledgmentNumber) {
    int begin = 0;
    int middle = 0;
    int end = min((begin + size * BUFFER_SIZE - 1), (fileSize - 1));
    char data[BUFFER_SIZE];
    int ackCount = 0;
    uint32_t ackHistory[3] = { 0 };
    ackReceived = false;
    // 随机数生成器，用于丢包
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(1, 100);
    while (begin < fileSize) {
        if (is_timeout) {
            setConsoleColor(12);
            cout << "触发超时重传机制" << endl;
            setConsoleColor(7);
            middle = begin;
            is_timeout = false;
        }

        if (ackReceived) {
            if ((ackNumber - 3) * BUFFER_SIZE < begin) {
                continue;
            }
            ackReceived = false;
            cout << "ackNumber:" << ackNumber << endl;
```

```
            begin = (ackNumber - 3) * BUFFER_SIZE;
            if (begin >= fileSize) {
                cout << "所有数据发送完成！" << endl;
                finish = 1;
                return true;
            }
            end = min(begin + size * BUFFER_SIZE - 1, fileSize - 1);
            setConsoleColor(10);
            cout << "窗口调整，新的 begin: " << begin << ", end: " << end << endl;
            setConsoleColor(7);
            ackHistory[ackCount++] = ackNumber;
            if (ackCount >= 3) {
                ackCount -= 3;
            }
            if (ackHistory[0] == ackHistory[1] && ackHistory[1] == ackHistory[2]) {
                setConsoleColor(12);
                cout << "触发快速重传机制" << endl;
                setConsoleColor(7);
                middle = begin;
                continue;
            }
        }
        // 发送数据包
        if (middle < end) {
            if (end - middle + 1 < BUFFER_SIZE) {
                for (int i = middle; i <= end; i++) {
                    data[i - middle] = fileContent[i];
                }
                uint16_t bytesRead = end - middle + 1;
                sequenceNumber = middle / BUFFER_SIZE + 3;
                uint32_t ack = seqNumber + 1;
                UDPHeader dataHeader = createUDPHeader(sourcePort, destPort, sequenceNumber,
                        ack, 0, bytesRead);
                PseudoHeader pseudoHeader = createPseudoHeader(clientAddr.sin_addr.s_addr,
                    serverAddr.sin_addr.s_addr, HEADER_SIZE + bytesRead);
                vector<char> packet = createPacket(dataHeader, data, bytesRead, pseudoHeader);
                // 丢包判断
                if (dist(gen) > PACKET_DROP_RATE) {
                    if (sendto(clientSocket, packet.data(), packet.size(), 0,
                        (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) {
                        cerr << "发送数据包失败!" << endl;
                        return false;
```

```
78                      }
79                  }
80                  else {
81                      setConsoleColor(12);
82                      cout << "数据包丢弃，序列号：" << sequenceNumber << endl;
83                      setConsoleColor(7);
84                  }
85                  middle = end;
86              }
87              else {
88                  for (int i = middle; i <= middle + BUFFER_SIZE - 1; i++) {
89                      data[i - middle] = fileContent[i];
90                  }
91                  uint16_t bytesRead = BUFFER_SIZE;
92                  sequenceNumber = middle / BUFFER_SIZE + 3;
93                  uint32_t ack = seqNumber + 1;
94                  UDPHeader dataHeader = createUDPHeader(sourcePort, destPort, sequenceNumber,
95                          ack, 0, bytesRead);
96                  PseudoHeader pseudoHeader = createPseudoHeader(clientAddr.sin_addr.s_addr,
97                          serverAddr.sin_addr.s_addr, HEADER_SIZE + bytesRead);
98                  vector<char> packet = createPacket(dataHeader, data, bytesRead, pseudoHeader);
99                  cout << "发送数据包的序列号：" << dataHeader.sequenceNumber << ", ACK 号：" <<
100                 dataHeader.acknowledgmentNumber << ", 校验和：" << dataHeader.checksum << endl;
101                 // 丢包判断
102                 if (dist(gen) > PACKET_DROP_RATE) {
103                     if (sendto(clientSocket, packet.data(), packet.size(), 0,
104                         (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) {
105                         cerr << "发送数据包失败!" << endl;
106                         return false;
107                     }
108                 }
109                 else {
110                     setConsoleColor(12);
111                     cout << "数据包丢弃，序列号：" << sequenceNumber << endl;
112                     setConsoleColor(7);
113                 }
114                 middle = middle + BUFFER_SIZE;
115             }
116         }
117     }
118     return true;
119 }
```

发送端接收线程的具体实现代码如下所示：

```
#include <condition_variable>
#include <mutex>
std::mutex mtx;
std::condition_variable cv;
bool is_timeout = false;   // 超时标志
// 接收 ACK 的线程
void receiveAck(int clientSocket) {
    fd_set readfds;
    struct timeval timeout;
    while (finish == 0) {
        cout << "ack!!!!" << endl;
        char buffer[BUFFER_SIZE] = {};
        sockaddr_in recvAddr = {};
        int recvAddrLen = sizeof(recvAddr);
        timeout.tv_sec = 0;
        timeout.tv_usec = TIMEOUT_MS * 1000;
        FD_ZERO(&readfds);
        FD_SET(clientSocket, &readfds);
        int result = select(clientSocket + 1, &readfds, nullptr, nullptr, &timeout);
        if (result > 0) {
            int bytesReceived = recvfrom(clientSocket, buffer, BUFFER_SIZE, 0,
                    (sockaddr*)&recvAddr, &recvAddrLen);
            if (bytesReceived > 0) {
                UDPHeader* recvHeader = (UDPHeader*)buffer;
                if (recvHeader->flags == ACK_FLAG) {
                    ackNumber = recvHeader->acknowledgmentNumber;
                    seqNumber = recvHeader->sequenceNumber;
                    std::cout << "收到 ACK: " << ackNumber << std::endl;
                    ackReceived = true;  // 收到 ACK, 设置标志位
                }
            }
        }
        else {
            if (finish == 1) {
                return;
            }
            else {
                is_timeout = true;
            }
        }
```

```
    }
    return;
}
```

4.4 接收到数据包后进行差错检测和校验

当进行文件传输时，**接收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，则接收端会将该数据包丢弃，等待发送端进行重传。

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**。

```
bool checkheader(const PseudoHeader& pseudoHeader, const UDPHeader& udpHeader,
    const char* data, int dataLength) {
    uint32_t sum = 0;
    // 伪首部求和
    sum += (pseudoHeader.srcIP >> 16) & 0xFFFF; // 源 IP 高 16 位
    sum += pseudoHeader.srcIP & 0xFFFF;         // 源 IP 低 16 位
    sum += (pseudoHeader.destIP >> 16) & 0xFFFF; // 目的 IP 高 16 位
    sum += pseudoHeader.destIP & 0xFFFF;         // 目的 IP 低 16 位
    sum += (pseudoHeader.reserved << 8) + pseudoHeader.protocol;
    // reserved 和 protocol 拼成 16 位
    sum += pseudoHeader.udpLength; // UDP 长度
    // UDP 头部求和
    sum += ntohs(udpHeader.sourcePort);  // 源端口
    sum += ntohs(udpHeader.destPort);    // 目的端口
    sum += (udpHeader.sequenceNumber >> 16) & 0xFFFF; // 序列号高 16 位
    sum += udpHeader.sequenceNumber & 0xFFFF;         // 序列号低 16 位
    sum += (udpHeader.acknowledgmentNumber >> 16) & 0xFFFF; // 确认号高 16 位
    sum += udpHeader.acknowledgmentNumber & 0xFFFF;         // 确认号低 16 位
    sum += (udpHeader.flags << 8) + udpHeader.reserved; // flags 和 reserved 拼成 16 位
    sum += udpHeader.length;        // 长度字段
    sum += udpHeader.windowSize;    // 窗口大小
    sum += udpHeader.checksum;      // 校验和字段
    // 数据部分求和
    for (int i = 0; i < dataLength; i += 2) {
        uint16_t word = (data[i] << 8) + (i + 1 < dataLength ? data[i + 1] : 0);
```

```
        sum += word;
    }
    // 处理进位
    while (sum > 0xFFFF) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // 检查结果
    if ((sum & 0xFFFF) == 0xFFFF) { // 所有位为 1 则校验正确
        cout << "校验正确" << endl;
        return true;
    }
    else {
        cout << "校验错误" << endl;
        return false;
    }
}
```

4.5 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，发送端会等待接收端返回 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**。使用路由器进行丢包后，具体实现效果如下所示：

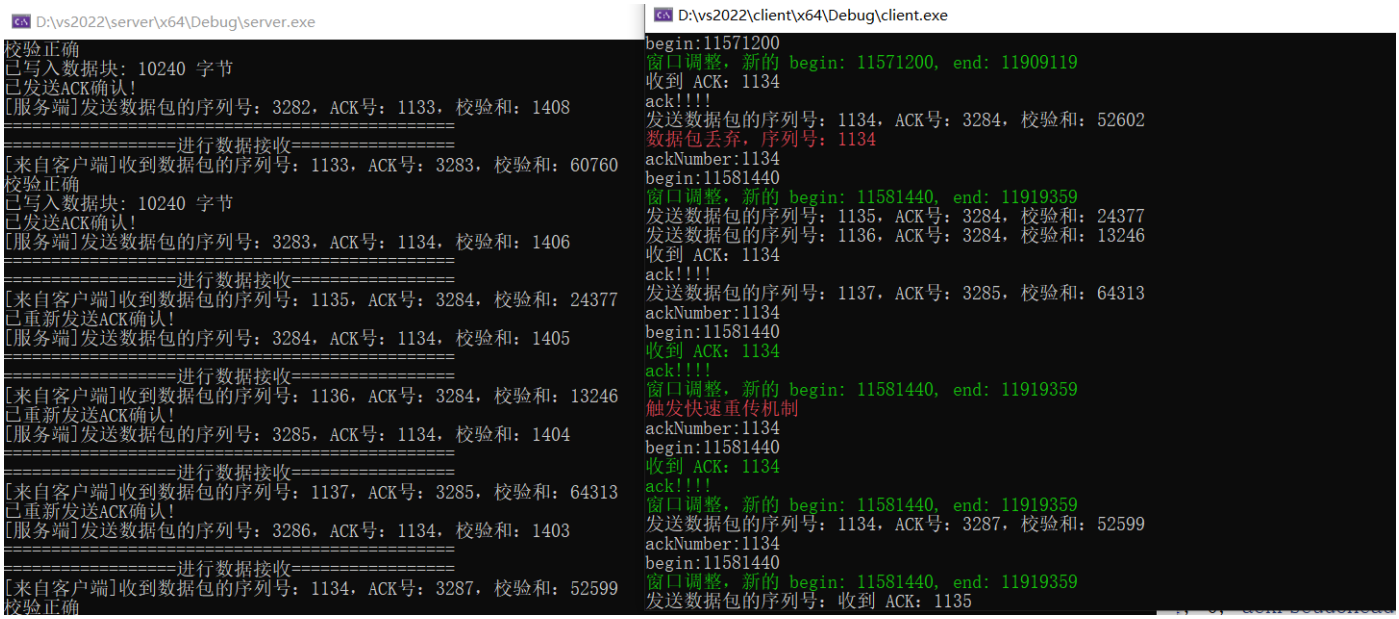

图 4.8: 文件传输过程中的超时重传和快速重传机制

从上图可以看出，快速重传机制在采取滑动窗口进行文件传输时非常有效，由于接收端采取累积确认的方式进行数据接收，当接收到的数据包序列号不是所期望的时，接收端会向发送端发送累计确

认的 ack 号，告知发送端自己需要的数据包。当发送端连续三次收到相同的 ACK 确认号时，就会判断出这个数据包已经丢失，接着会调整滑动窗口中 middle 指针的位置，进行窗口内数据包的重新发送。快速重传机制的优势在于，发送端能够在没有等待超时的情况下及时发现丢包并进行重传，避免了等待超时带来的延迟，提高了文件传输的实时性和效率。

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，即只会从发送端向接收端传输文件，因此我们只需要三次挥手即可完成连接的释放。前面说到，当发送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，接收端接收到这个数据报后会回复 ACK 信号。而当发送端接收到接收端的回复后，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：**首先发送端会发送一个带有 FIN 标志位的数据报，表示发送端已经没有文件需要传输了，希望断开连接。当接收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，**同时由于接收端不需要向发送端传输文件，因此接收端回复带有 FIN 标志位和 ACK 标志位的数据报，表示接收端也希望断开连接。当发送端收到接收端的回复信号后，三次挥手结束，连接被成功释放**。具体实现效果如下图所示：

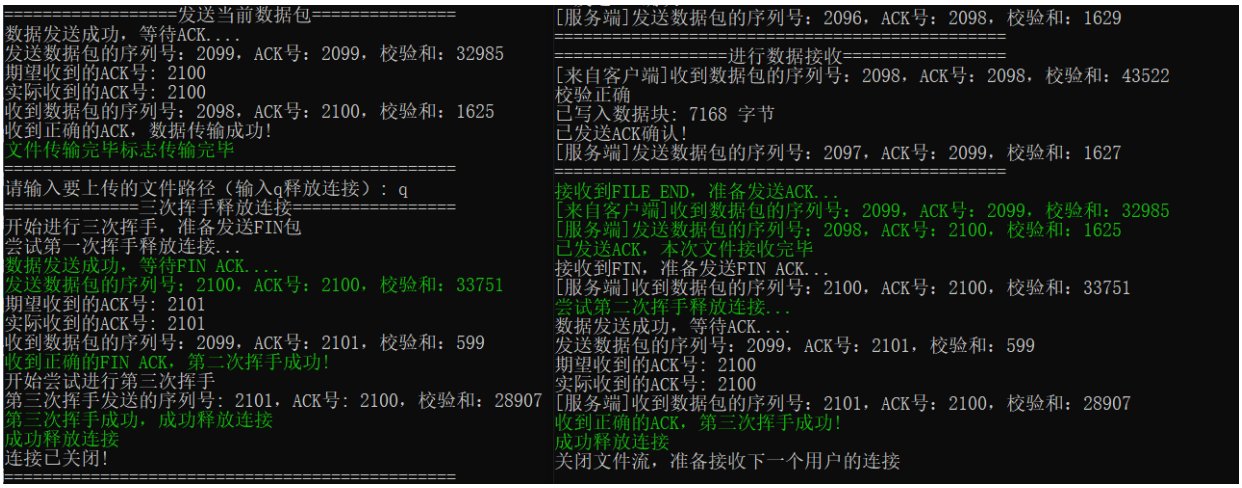

图 5.9: 三次挥手的具体实现效果

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发送端和接收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：

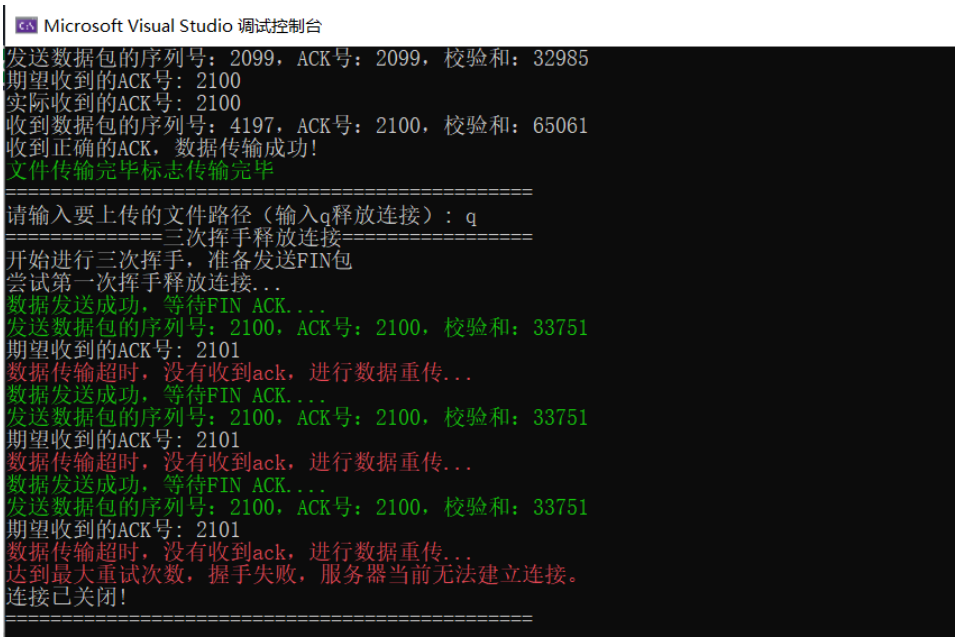

图 5.10: 三次挥手中的超时重传

当三次挥手释放连接后发送端会退出，**但接收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**:

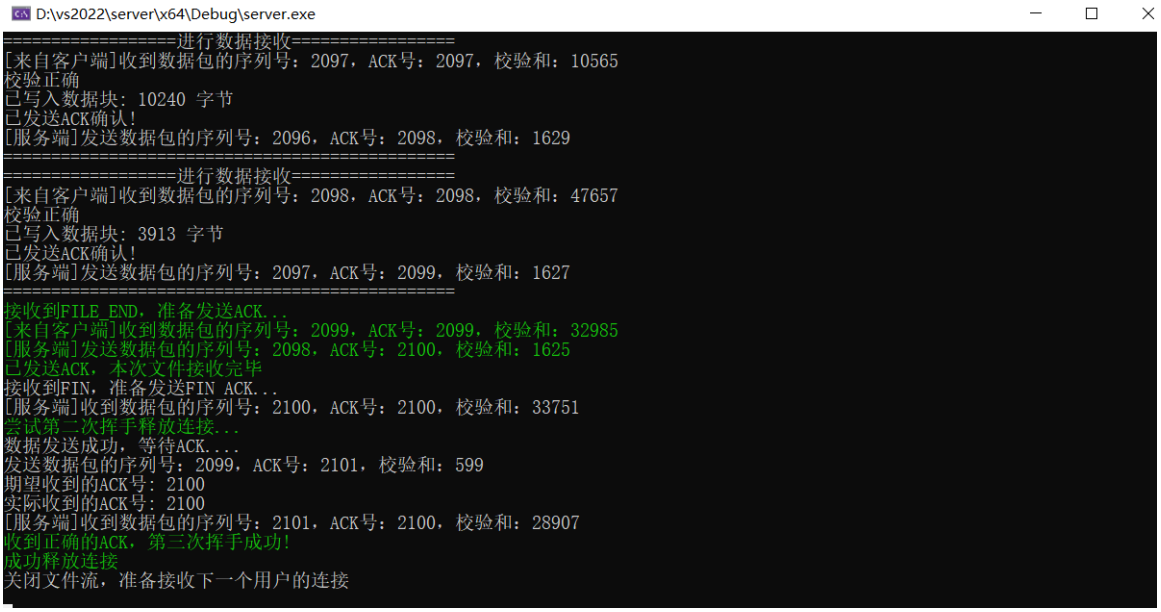

图 5.11: 接收端等待下一个用户进行连接

6 文件传输效果和性能测试指标

6.1 停等机制与 GBN 机制文件传输性能对比

本实验中要求使用传输时间和吞吐率作为性能测试指标，首先我对比了不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件的传输时间，如下表所示。

传输文件大小 (MB)	停等机制 (s)	GBN 机制 window_size=10(s)	size=20(s)	size=30(s)	size=40(s)
1.5791	0.7356	0.6374	0.5690	0.5473	0.5360
1.7713	0.8065	0.6814	0.6099	0.6163	0.6101
5.6525	2.4038	2.0760	1.7181	1.6506	1.7137
11.4145	4.8120	3.3383	3.2653	3.2475	3.1113

表 1: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间对比

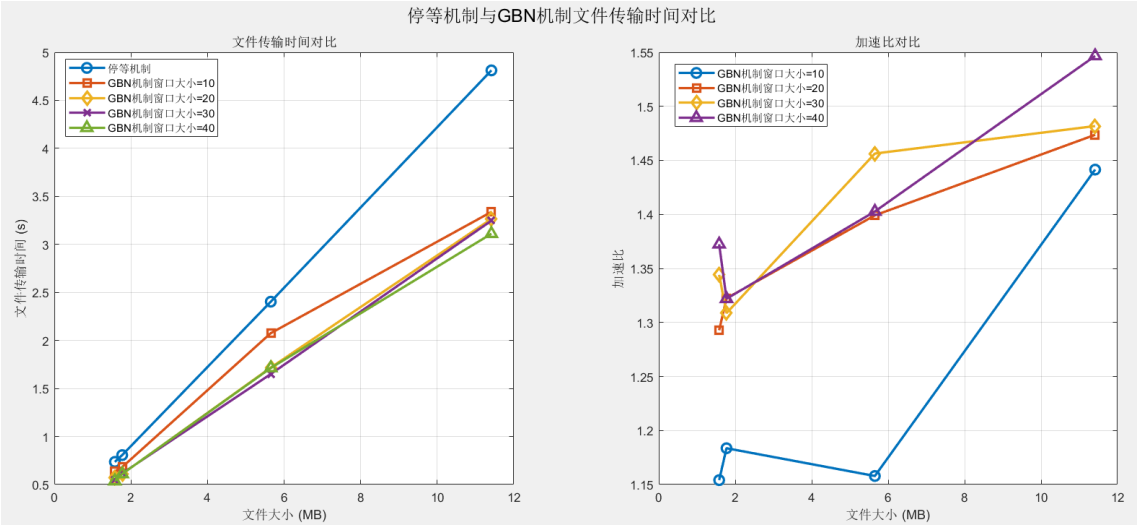

图 6.12: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间及加速比对比

传输文件大小 (MB)	停等机制 (MB/s)	GBN 机制 size=10(MB/s)	size=20	size=30	size=40
1.5791	2.147	2.477	2.775	2.885	2.945
1.7713	2.196	2.599	2.904	2.873	2.903
5.6525	2.340	2.710	3.274	3.408	3.282
11.4145	2.372	3.419	3.496	3.514	3.668

表 2: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时吞吐率大小对比

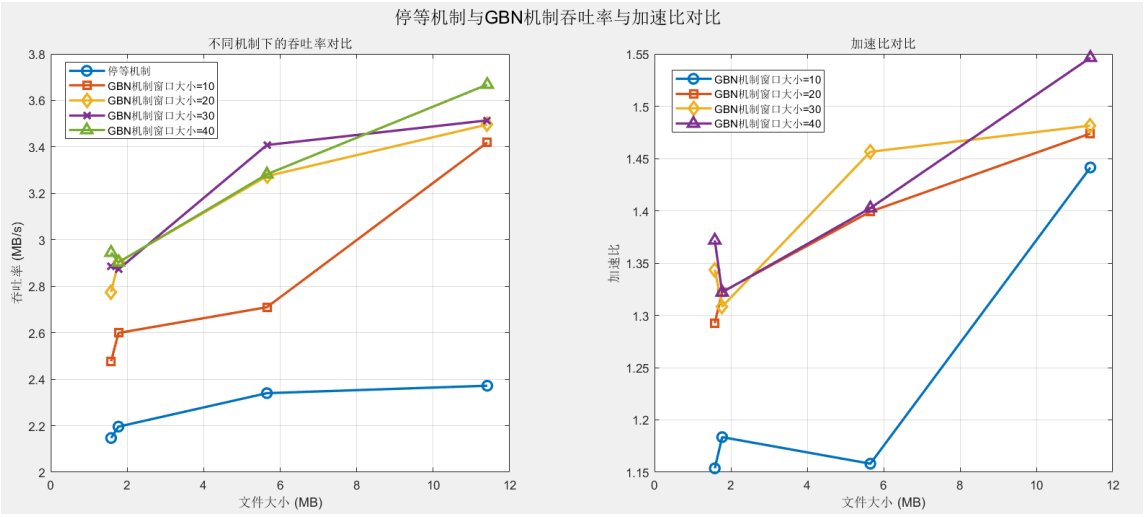

图 6.13: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时吞吐率大小及加速比对比

由表中数据和图中加速比对比可以发现，**采取不同窗口大小的 GBN 机制相较于停等机制的传输时间都有所降低，并且吞吐率都有较大的提高，由图 6.12 中的加速比对比可以发现，采取停等机制进行流量控制时的文件传输时间约为 GBN 机制的 1.5 倍左右，并且随着文件大小的增加，加速比呈现上升趋势**。这表明对于较大的文件，滑动窗口机制的优势愈发明显，能够更好地利用网络带宽，从而提高传输效率。同时我们还可以看出，窗口大小为 10 时，文件传输时间虽然有所降低，但效果并不显著。而当我们逐渐将窗口大小增加为 20、30、40 时，发现获得了更好的加速效果，传输时间明显下降。表明较大的窗口大小能够显著提升文件传输效率。尤其是在文件大小较大的情况下，GBN 机制能够有效减少等待时间，提高数据传输的吞吐率。**此外，当窗口大小增加到一定程度（如 20、30、40）时，虽然传输时间进一步降低，但加速比趋于平稳，表明此时窗口大小已经较为合适。继续增大窗口大小可能会带来一定的加速效果，但效果变得越来越不明显。因此，在实践中，选择合适的窗口大小能够平衡性能提升和资源消耗，避免过度调整导致的性能饱和**。

表 2 和图 6.13 中的吞吐率大小对比也表现出类似的规律。与传输时间的变化趋势一致，窗口为 10 时吞吐率提升较为有限，但随着窗口大小逐步增加，吞吐率呈现明显的提升趋势，尤其是在大文件传输时更为显著。随着窗口增大，传输效率逐步提高，但当窗口大小达到一定范围后，吞吐率的增长趋于平稳。表明此时的窗口大小已经能够有效匹配网络带宽，进一步增大窗口对吞吐率的提升作用变得微弱。

6.2 丢包率为 5% 时停等机制与 GBN 机制文件传输性能对比

为了探究在出现丢包情况时，采用停等机制与 GBN 机制进行流量控制的文件传输性能差异，我将丢包率固定为 5%，对比此时两种机制下的文件传输时间，实验数据如下表所示：

传输文件大小 (MB)	停等机制 (s)	GBN 机制 (s) size=10	size=20	size=30	size=40
1.5791	2.7707	0.6493	0.5737	0.6109	0.6256
1.7713	3.1219	0.7681	0.6734	0.7033	0.7001
5.6525	9.3282	1.9455	2.0901	1.9527	1.8048
11.4145	19.5941	4.1254	3.8465	3.6719	3.6728

表 3: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间对比

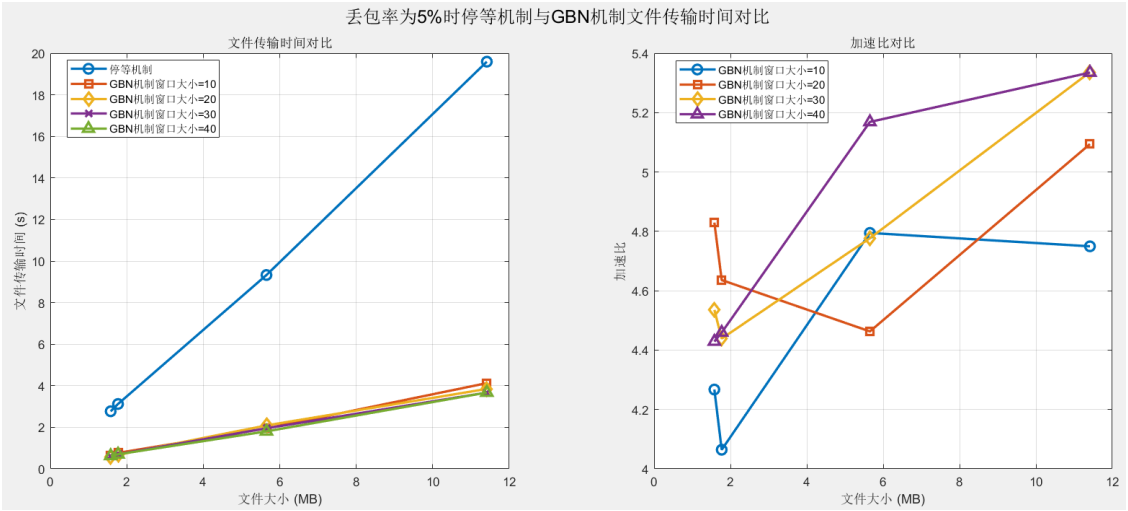

图 6.14: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间及加速比对比

由图 6.14 中的加速比对比我们可以发现，**当出现丢包情况时，采取 GBN 机制进行流量控制的传输时间远远小于停等机制，加速比大约为 4.5 左右，表明 GBN 机制在丢包环境下能有效减少传输延迟并提高带宽利用率**。并且随着文件大小的增加，加速比呈现逐步上升的趋势，进一步证明了 GBN 机制在处理大文件传输时具有显著优势。具体而言，GBN 机制能够通过窗口控制提高数据流的连续性，从而减少等待时间和传输过程中的空闲期，使得网络带宽得到更充分的利用。

同时通过分析表 3 中的数据我也发现，在传输较小文件时，单纯增大滑动窗口的大小并不一定带来预期的性能提升。以文件大小为 1.5791 MB 为例，窗口大小为 40 时的传输时间反而超过了其他窗口大小的 GBN 机制。这是因为当文件较小时，较大的窗口会导致在丢包时需要重发窗口内的所有数据包。由于窗口较大，导致需要重新传输的数据包较多，从而抵消了大窗口带来的传输优势。相反，对于小文件，采用较小的窗口大小反而能够减少丢包后需要重发的数据包数量，进而提升传输效率。**因此，在文件大小较小的情况下需要适当选择滑动窗口的大小，避免因过大的窗口导致性能反而下降**。

6.3 延时为 10ms 时停等机制与 GBN 机制文件传输性能对比

为了探究在出现延迟的情况时，采用停等机制与 GBN 机制进行流量控制的文件传输性能差异，我将延时固定为 10ms，对比此时两种机制下的文件传输时间，实验数据如下表所示：

传输文件大小 (MB)	停等机制 (s)	GBN 机制 (s) size=10	size=20	size=30	size=40
1.5791	4.9970	2.6124	2.6547	2.6098	2.6491
1.7713	5.5655	2.9684	2.9951	2.9431	2.9498
5.6525	17.7379	9.4592	9.1149	9.2304	9.1516
11.4145	36.1160	19.3694	20.0277	20.1006	20.9793

表 4: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间对比

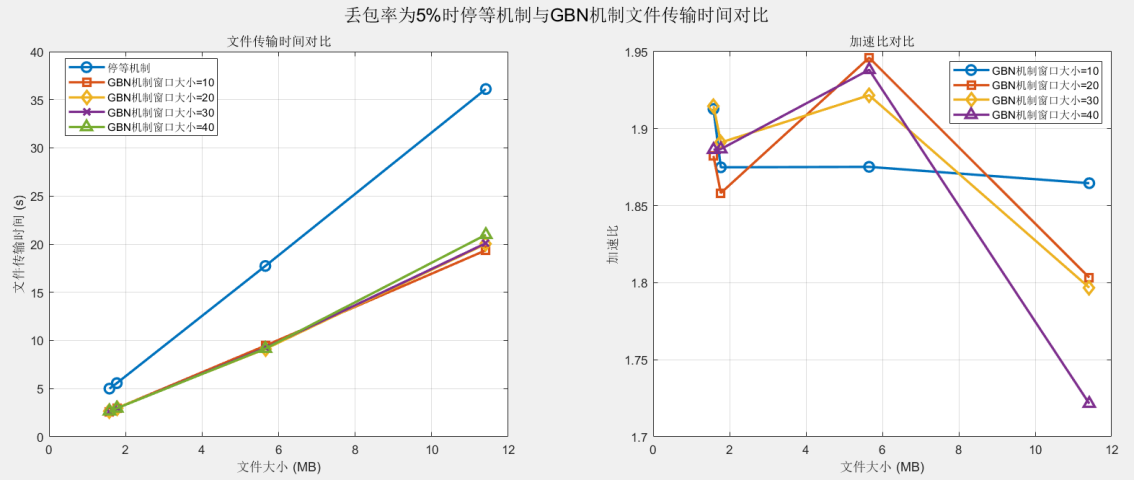

图 6.15: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间及加速比对比

首先对于停等机制与 GBN 机制的性能差异，由图 6.15 我们可以发现，尽**管在出现延迟的情况下 GBN 机制的传输性能大幅降低，但是延时为 10ms 时，采取 GBN 机制进行流量控制的性能依然优于停等机制，加速比大概在 1.85 左右**。并且在实验中我发现，由于我们引入了延时，因此滑动窗口在进行数据包发送时几乎不会出现超时或者快速重传的情况，这是因为我们引入了延时，使得每发送一个数据包之后都会等待一段时间，在这个时间内发送端接收到了 ack 确认信号，**因此不会出现超时或**

者由于发送数据包过快接收端来不及接收导致的重传，此时其实相当于停等机制，表现出相似的等待过程。

但此时又出现了另一个疑问，既然此时采用滑动窗口进行每一次数据发送都会在收到上一个数据包的确认 ack 信号之后，并且停等机制也采取了同样的延时，也就是说效果相当于停等机制，那此时停等机制的传输效率应该与滑动窗口相等啊，为什么实验数据显示依然是 GBN 机制效果更优呢。**经过多次实验测试，我认为这是由于多线程导致的性能优势。GBN 机制采取了单独的接收线程接收 ack 确认信号，这使得数据的传输和确认过程可以并行进行，而不是像停等机制那样在发送数据时阻塞等待 ACK。**因此由于多线程的引入提高了传输效率，减少了因等待 ACK 信号而产生的延迟。

对于采取相同线程数量但窗口大小不同的 GBN 机制，由表 4 中数据我们可以发现，它们的文件传输时间是近乎相等的，这也验证了上面的结论。**同时，针对我们得出的这个结论，我们可以推测当延时继续增大时，GBN 机制的性能依然会优于停等机制，而当延时继续缩小时，特别是当延时时间小于发送数据包的往返时延 rtt 时，GBN 机制的性能优势会扩大，并且性能表现会接近不存在延时情况时的传输时间**，因为如果当延时时间过了，滑动窗口没有接收到 ack 确认信号也会继续发送下一个未发送的数据包，而停等机制则会继续等待，直到收到 ack 确认信号。后面我们会继续探讨不同延时下的停等机制与 GBN 机制的传输性能对比。

6.4 不同延时情况下停等机制与 GBN 机制文件传输性能对比

为了进一步验证上面我们得出的实验结论，我将文件大小固定为 11.4145MB，对不同延时情况下停等机制与 GBN 机制的文件传输性能进行了测试，实验数据如下表所示：

延时大小 (ms)	停等机制 (s)	GBN 机制 (s) size=10	size=20	size=30	size=40
0.001	5.0134	3.7214	3.5179	3.2147	3.1179
1	26.3449	17.8958	17.8976	17.2247	17.4527
5	34.8202	18.0304	18.1492	18.0784	18.2273
10	36.1160	19.3694	20.0277	20.1006	20.9793
20	53.5960	37.3379	37.1149	37.7842	37.2957

表 5: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间对比

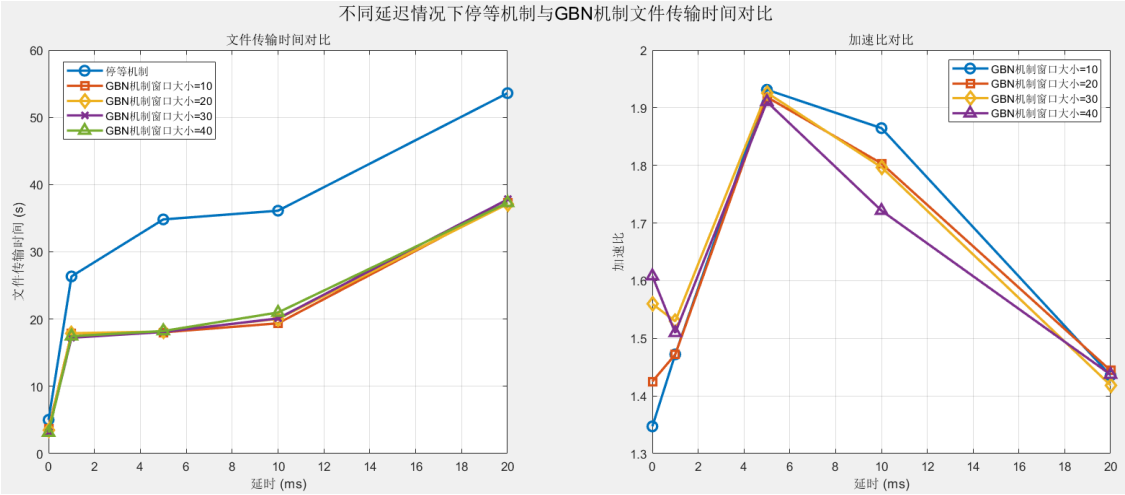

图 6.16: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间及加速比对比

表 5 中的实验数据证明了前面我们得出的结论，当延时继续增大时，GBN 机制的性能依然优

于停等机制，而当我们将延时大小降低时，在实验中我逐渐降低延时大小，当延时大小降低为 0.1 和 0.01ms 时，GBN 机制的文件传输时间虽然有所降低，但我仍然观察到发送端接收一个包之后才会发送下一个包，这说明当前的延时大小仍然高于 rtt，往返时延 rtt 应该更小。然后我继续降低延时大小，**发现当延时大小为 0.001ms 时，GBN 机制的文件传输时间与不使用延时的情况下的传输时间接近，并且相比延时时间更大的情况，文件传输花费的时间明显更少**，这说明文件传输时延应该与 0.001ms 接近，也正与我们前面得出的结论相一致。需要特别说明的是，由于实验所用的路由器程序的最低延时只能设置为 1ms，因此当我们进一步测试更低延时时间时，采用了程序模拟路由器的方式。每当发送数据包之后，程序会人为插入延时，保证低于 1ms 的延时也能在实验中得到体现。

6.5 不同丢包率情况下停等机制与 GBN 机制文件传输性能对比

为了探究不同丢包率情况下停等机制与 GBN 机制的文件传输性能对比，我将文件大小固定为 11.4145MB，对不同丢包率情况下停等机制与 GBN 机制的文件传输性能进行了测试，实验数据如下表所示：

丢包率 (%)	停等机制 (s)	GBN 机制 (s) size=10	size=20	size=30	size=40
1	13.2247	3.6864	3.5364	3.3214	3.1403
5	19.5941	4.1254	3.8465	3.6719	3.6728
10	27.4258	4.0654	3.5024	4.0027	4.5754
20	44.8128	5.5317	5.7621	5.8853	6.0621
30	74.6957	8.1107	9.7249	9.9236	10.2473

表 6: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间对比

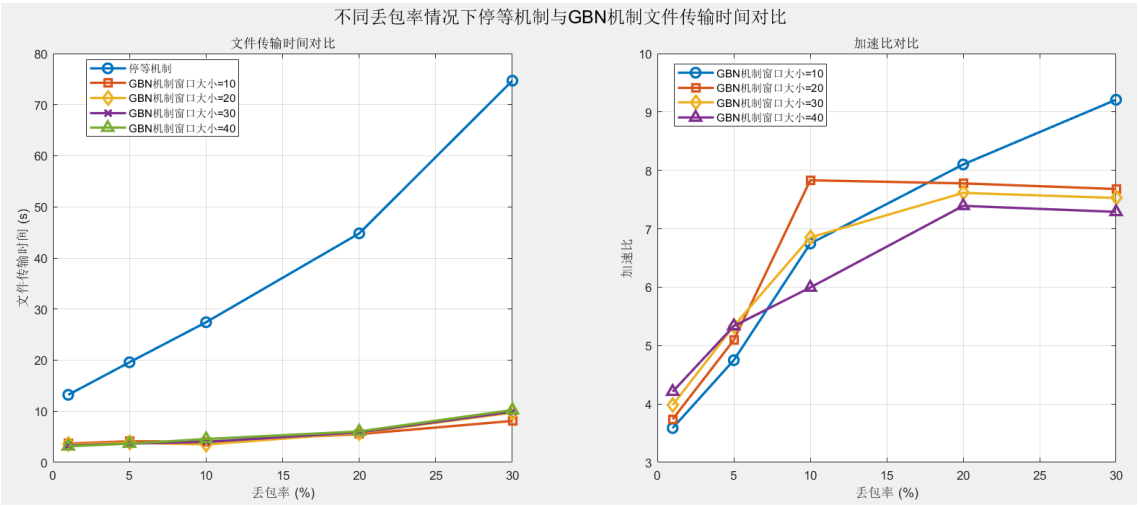

图 6.17: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间及加速比对比

由图 6.17 中不同丢包率情况下采取 GBN 机制的加速比对比图我们可以发现，**随着丢包率的提高，采取 GBN 机制进行流量控制所获得的加速比呈现出上升趋势，在丢包率为 30% 的情况下，所获得最高的加速比约为 9.2 左右，相比停等机制实现了较高的传输速率**。说明丢包对于停等机制的影响更严重，而相对来说对于 GBN 机制的影响不是很大。但当丢包率较高时，例如丢包率为 30% 时，由表 6 中的实验数据我们可以看出，此时丢包率对于 GBN 机制的传输速率也有较大影响，导致 GBN 机制的传输时间增加较多在，这是因为当丢包率较高时，丢弃关键数据包即接收端缺少的数据包的概率较高，导致重传次数较多，进而导致文件传输时间的大幅增加。

并且由表 6 中的数据我们**可以得出跟实验三（对比丢包率为 5% 时停等机制与 GBN 机制文件传输性能）类似的结论，当丢包率增加到一定程度时，即使我们进行传输的文件较大，选择窗口更大的滑动窗口会获得更高的传输速率，但由于此时丢包率高，数据包丢失的情况会经常发生，而窗口较大导致的问题就是丢包时需要重传的数据包多，这就导致选择更高的窗口传输时间反而增加的现象**。如表 6 中的数据所示，当丢包率为 1% 时，随着滑动窗口大小的增加，文件传输时间不断减少，而当丢包率提高到 30% 时，随着滑动窗口大小的增加，文件传输时间反而不断增大。不难看出，当丢包率为 1% 时，选择窗口大小 40 可以获得最高的传输速率，当丢包率为 5% 时，选择窗口大小 30 可以获得最高的传输速率，当丢包率为 10% 时，选择窗口大小 20 可以获得最高的传输速率，而当丢包率为 20% 和 30% 时，选择窗口大小 10 可以获得最高的传输速率。**因此在选择滑动窗口大小时，我们需要综合考虑丢包率和文件大小等因素，以获得最佳的文件传输效率**。

综上所述，尽管丢包率的提高会导致 GBN 机制的传输速率下降，但由于 GBN 机制不需要等待确认，相比于停等机制，它仍然能够实现较好的速率提高效果，特别是在丢包率较高的情况下，采取 GBN 机制进行流量控制实现了较高的加速比。

6.6 不同传输机制的横向对比

进一步，我们对不同传输机制在上述各种情况下的传输性能进行横向对比，传输文件大小固定为 11.4145MB，对比停等机制与 GBN 机制在不同丢包率与延时情况下的文件传输性能变化趋势，实验数据如下表所示：

	normal	丢包率 =1%	丢包率 =5%	丢包率 =10%	延时 =1ms	延时 =10ms	延时 =20ms
停等机制	4.8120	13.2247	19.5941	27.4258	26.3449	36.1160	53.5960
size=10	3.3383	3.6864	4.1254	4.0654	17.8958	19.3694	37.3379
size=20	3.2653	3.5364	3.8465	3.5024	17.8976	20.0277	37.1149
size=30	3.2475	3.3214	3.6719	4.0027	17.2247	20.1006	37.7842
size=40	3.1113	3.1403	3.6728	4.5754	17.4527	20.9793	37.2957

表 7: 不同文件大小下采用停等机制与 GBN 机制进行流量控制时文件传输时间对比

由表 7 中的实验数据我们可以发现，延迟对不同流量控制机制的传输时间影响显著，并且这种影响远大于丢包率的影响。并且延时对于 GBN 机制的传输时间的影响尤为显著，当延时仅为 1ms 时，传输时间就增加了约 5.3 倍，导致了较大的性能损失，**这是因为** GBN 机制依赖于滑动窗口技术，通过连续发送多个数据包来提高文件传输速率。然而，在实际文件传输过程中，本机的往返时延（RTT）远远小于 1ms，因此，当延迟为 1ms 时，GBN 机制每发送一个数据包就必须等待 1ms 的确认回传，导致发送速率大幅下降。更重要的是，当发生超时情况需要重传当前窗口内所有数据包时，延迟造成的性能影响将更为严重。因此，尽管 GBN 机制具有较高的传输速率，但在存在较高延迟的环境下，其性能会大幅降低，尤其是在丢包频繁和需要多次重传的情况下，延迟的负面影响更加突出。

相比之下，丢包率对停等机制的影响则更为显著。即使在较高丢包率的情况下，GBN 机制的性能损失也较小，而停等机制在丢包率为 10% 时，传输时间几乎增加了 5.7 倍，造成了显著的性能下降。原因在于停等机制的工作方式是每发送一个数据包后必须等待接收方的确认，丢失一个数据包就意味着发送方需要等待超时，才能知道该数据包已丢失并进行重传，这种重传机制导致了较长的等待时间，尤其是在丢包较多的情况下。而 GBN 机制则不同，它允许在窗口内连续发送多个数据包，因此即使某些数据包丢失，发送方依然能够继续发送剩余的数据包，避免了因等待确认而造成的时间浪费。这样，接收方可以及时返回丢失的数据包的重传请求，而无需等待所有数据包的确认，从而避免了停等机制中的冗长等待时间，提升了整体传输效率。

因此，尽管在丢包率较高的情况下，GBN 机制依然能够保持较好的性能，而停等机制则容易受到丢包的影响，导致传输效率显著降低。这两种机制在面对不同网络环境（如高丢包率或高延迟）时的表现差异，体现了它们各自的适应性和局限性。

7 实验内容总结

本次实验在实验 3-1 的基础上，通过引入滑动窗口机制，成功实现了一个基于 GBN 机制进行流量控制的可靠传输协议。我们采用固定窗口大小并支持累积确认的方式，完成了给定测试文件的传输，并对性能进行了评估与分析。实验主要围绕协议设计、流量控制、性能测试等方面展开，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，协议设计中加入了序列号、确认号、窗口大小等字段，确保数据包的顺序传输并有效支持累积确认机制。滑动窗口机制的引入使得发送端能够在等待确认的同时继续发送数据包，从而提高了文件传输效率。
- 滑动窗口机制与流量控制：与停等机制相比，滑动窗口机制允许发送端在未收到确认的情况下发送多个数据包，从而避免了由于等待确认包而产生的性能瓶颈。在实验中，滑动窗口的设计有效控制了流量，并通过累积确认减少了确认包的数量，进一步提升了协议的传输效率。
- 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，当发送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，发送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，避免了发送端长时间等待超时的情况，从而提高了协议的传输效率。
- 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：实验通过测试文件传输时延、吞吐率等指标，评估了协议在不同网络条件下的性能。通过图形分析，我们发现，滑动窗口机制显著提高了吞吐率，尤其在丢包率较高的网络环境下，相较于停等机制，滑动窗口机制能够避免过多的等待时间，提升了传输效率。此外，实验中还对不同文件大小和网络条件下的传输性能进行了对比，结果表明 GBN 机制在多种情况下均具有较高的文件传输性能。

同时在实现过程中，我注意到以下几个关键点:

- 从实验数据来看，延迟对于传输时间的影响远大于丢包率。尤其对于 GBN 机制，当延迟为 1ms 时，传输时间增加了约 5.3 倍，性能损失较大。这是因为 GBN 机制依赖滑动窗口技术，在网络延迟较高时，每发送一个数据包都需要等待 1ms 的确认回传，从而导致性能下降。
- 丢包率对停等机制的影响较为明显。尤其在丢包率较高的情况下，停等机制需要在每次丢包后等待超时，导致传输时间显著增加。在丢包率达到 10% 时，停等机制的传输时间增加了约 5.7 倍，性能损失较大。而滑动窗口机制由于能够同时发送多个数据包，减少了丢失关键数据包的风险，即使发生丢包，仍能继续发送窗口内其他数据包，避免了等待超时的影响。

通过本次实验，我深入理解了滑动窗口机制及其在提高数据传输效率和可靠性方面的作用。同时，我认识到延迟和丢包率是影响传输性能的关键因素，如何有效应对这两者的挑战是网络协议优化的关键。此外，本次实验还让我意识到，协议设计中的流量控制和超时机制对保证传输的稳定性和高效性至关重要，为后续网络协议的改进和应用提供了宝贵的经验。