



南開大學

Nankai University

计算机学院

计算机网络实验报告

实验 3.3：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 13 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：进行拥塞控制，动态调整窗口大小	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	9
4.3 发送端采取 GBN 机制进行流量控制	10
4.4 发送端采取拥塞控制机制动态调整窗口大小	11
4.5 接收到数据包后进行差错检测和校验	19
4.6 文件传输过程中的超时重传和快速重传机制	20
5 三次挥手释放连接	21
5.1 三次挥手的基本流程	21
5.2 接收到数据包后进行差错检测和校验	21
6 文件传输效果和性能测试指标	22
6.1 引入拥塞控制后的 GBN 机制文件传输性能提升分析	22
6.2 丢包率为 5% 时 GBN 机制与引入拥塞控制时的文件传输性能对比	24
6.3 延时为 10ms 时 GBN 机制与引入拥塞控制时的文件传输性能对比	25
6.4 不同延时情况下 GBN 机制与引入拥塞控制时的文件传输性能对比	26
6.5 不同丢包率情况下采用 GBN 机制与引入拥塞控制时的文件传输性能对比	27
6.6 不同传输机制的横向对比	28
7 实验内容总结	29

1 实验要求

在实验 3-2 的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，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) 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) 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

首先简要的回顾一下之前实验中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。

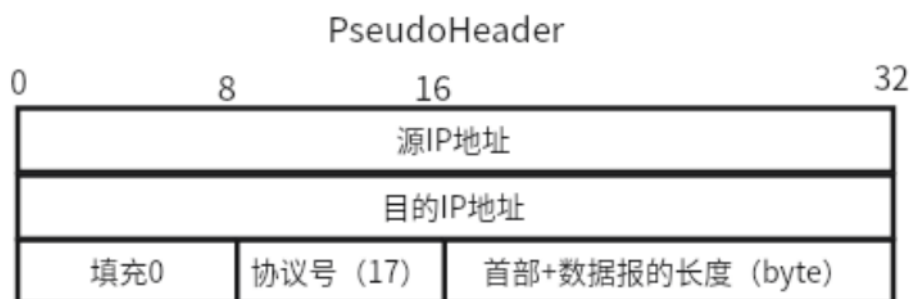


图 2.1: 伪首部报文格式

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。窗口大小的加入是实现滑动窗口机制和拥塞控制机制的关键，在我们的程序中这个标志位用来根据接收端的接收程度和当前网络的拥塞状况来协调发送端的窗口大小。当然，在实验 3-3 中，接收端的窗口大小被固定为 1，因此发送端滑动窗口的大小仅取决于当前网络的拥塞程度。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。

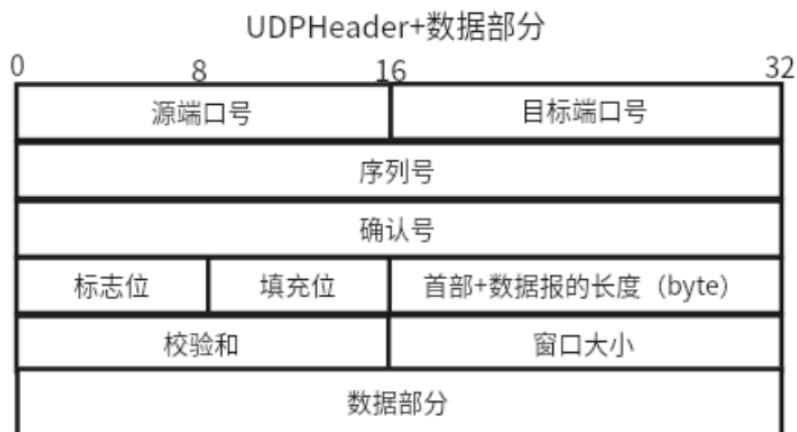


图 2.2: 首部报文格式

3 三次握手建立连接

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

由于实验 3-3 中三次握手建立连接的过程与之前的实验相同, 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拥塞控制机制。

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:

```

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):

```

```

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

```

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

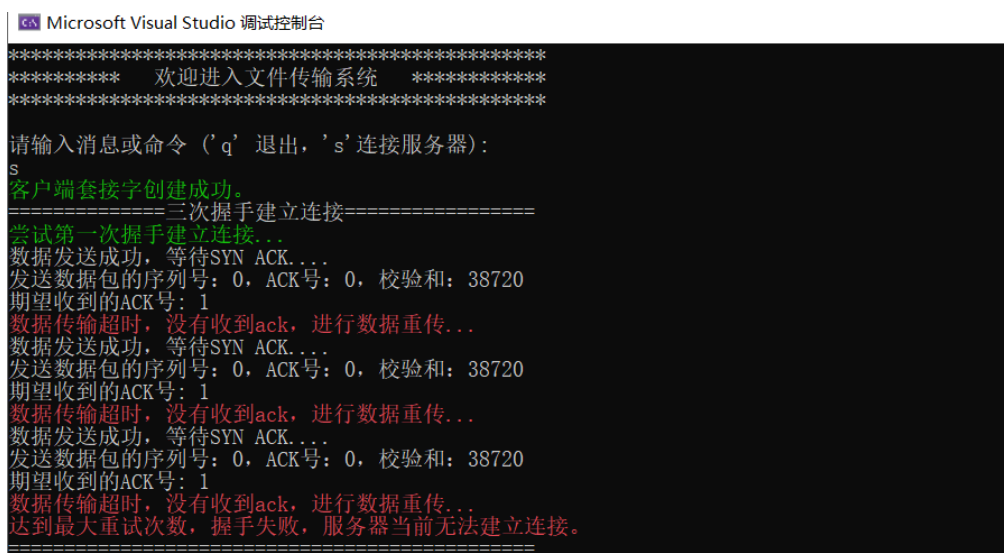
在发送端接收到来自对方的 SYN、ACK 数据包时，需要先从接收到的字节流中提取出 UDP 数据报的首部，然后对数据包的各个字段进行检查和验证，以确保数据的可靠性和完整性。需要查看首部的标志位是否符合自己的要求，并且检查确认号是否匹配，匹配的话则发送 ACK 信号进行确认。而当接收端接收到发送端返回的 ACK 确认信号后，也需要进行类似的操作，检查数据包的首部字段和确认号是否匹配。并且当进行数据传输时，接收端还需要根据数据包的源 IP 地址和目的 IP 地址重新构造出伪首部来对校验和字段进行校验，通过检验校验和的正确性来确定数据在传输过程中没有被篡改或丢失。如果得到的数据包校验和错误，即数据在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



```

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 3.4: 超时重传和避免无限重传

4 文件传输：进行拥塞控制，动态调整窗口大小

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;

```

```

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        wanted = ackHeader.acknowledgmentNumber;
30        cout << "发送数据包的序列号: " << ackHeader.sequenceNumber << ", ACK 号: " <<
31            ackHeader.acknowledgmentNumber << ", 校验和: " << ackHeader.checksum << endl;
32    }
33    continue;
34 }
35 else
36 {
37     cout << "===== 进行数据接收 =====" << endl;
38     cout << "[来自客户端] 收到数据包的序列号: " << recvHeader->sequenceNumber << ", ACK 号: " <<
39         recvHeader->acknowledgmentNumber << ", 校验和: " << recvHeader->checksum << endl;
40     char* data = buffer_fin + HEADER_SIZE;
41     int dataLength = bytesReceived - HEADER_SIZE;
42     if (recvHeader->sequenceNumber == wanted && checkheader(pseudorecvHeader, *recvHeader
43         , data, bytesReceived - HEADER_SIZE) == true)
44     {
45         // 保存到文件
46         outFile.write(data, dataLength);
47         cout << "已写入数据块: " << dataLength << " 字节" << endl;

```



```

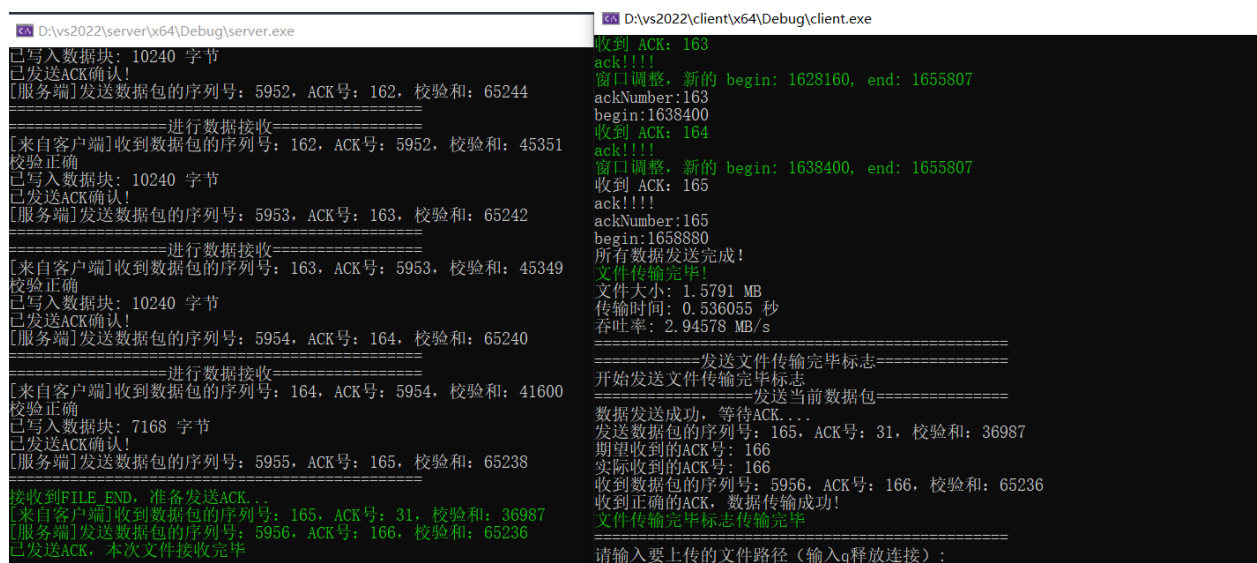
48
49     acknowledgmentNumber = recvHeader->sequenceNumber + 1;
50     UDPHeader ackHeader = createUDPHeader(serverPort, recvHeader->sourcePort,
51         ++sequenceNumber, acknowledgmentNumber, ACK_FLAG, 0);
52     PseudoHeader ackPseudoHeader = createPseudoHeader(serverAddr.sin_addr.s_addr,
53         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。

在实验 3-3 中，我们仍然采用 GBN 机制来进行流量控制，同时为了优化传输效率和适应不同的网络环境，我们新引入了拥塞控制机制来动态调整滑动窗口的大小，能够根据当前的网络状态进行动态调整。

采用 GBN 机制进行流量控制的具体实现效果如下图所示：



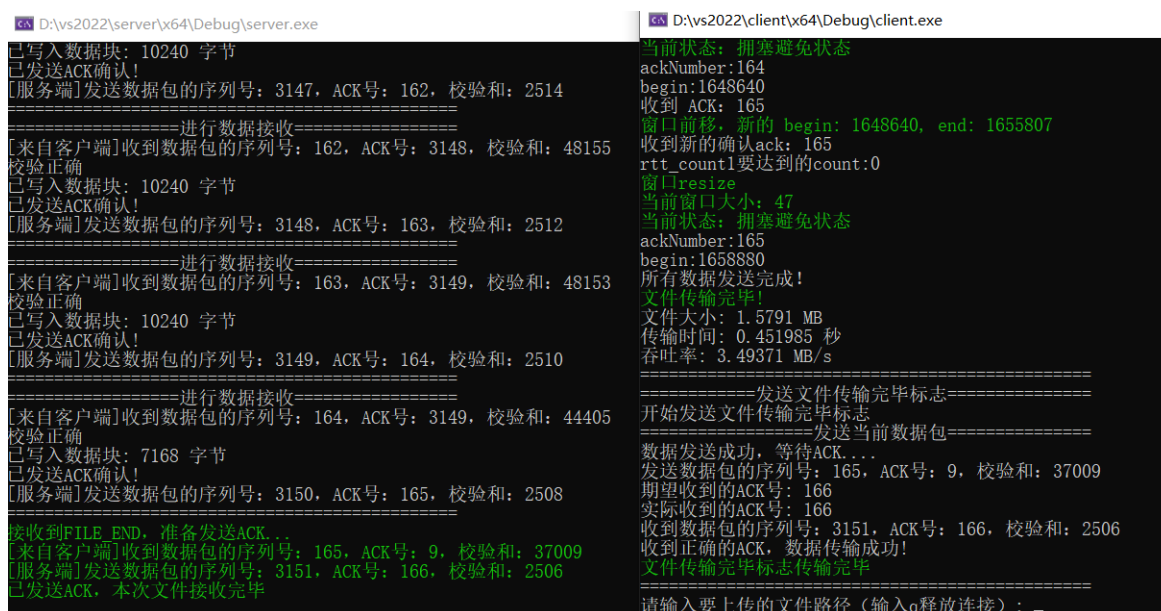
```
D:\vs2022\server\x64\Debug\server.exe
已写入数据块: 10240 字节
已发送ACK确认!
[服务端]发送数据包的序列号: 5952, ACK号: 162, 校验和: 65244
=====
-----进行数据接收-----
[来自客户端]收到数据包的序列号: 162, ACK号: 5952, 校验和: 45351
校验正确
已写入数据块: 10240 字节
已发送ACK确认!
[服务端]发送数据包的序列号: 5953, ACK号: 163, 校验和: 65242
=====
-----进行数据接收-----
[来自客户端]收到数据包的序列号: 163, ACK号: 5953, 校验和: 45349
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: 5954, ACK号: 164, 校验和: 65240
=====
-----进行数据接收-----
[来自客户端]收到数据包的序列号: 164, ACK号: 5954, 校验和: 41600
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: 5955, ACK号: 165, 校验和: 65238
=====
接收到FILE END, 准备发送ACK...
[来自客户端]收到数据包的序列号: 165, ACK号: 31, 校验和: 36987
[服务端]发送数据包的序列号: 5956, ACK号: 166, 校验和: 65236
已发送ACK, 本次文件接收完毕

D:\vs2022\client\x64\Debug\client.exe
收到 ACK: 163
ack!!!!
窗口调整, 新的 begin: 1628160, end: 1655807
ackNumber:163
begin:1638400
收到 ACK: 164
ack!!!!
窗口调整, 新的 begin: 1638400, end: 1655807
收到 ACK: 165
ack!!!!
ackNumber:165
begin:1658880
所有数据发送完成!
文件传输完毕!
文件大小: 1.5791 MB
传输时间: 0.536055 秒
吞吐率: 2.94578 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: 165, ACK号: 31, 校验和: 36987
期望收到的ACK号: 166
实际收到的ACK号: 166
收到数据包的序列号: 5956, ACK号: 166, 校验和: 65236
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):
```

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。

进一步引入拥塞控制机制的具体实现效果如下图所示：



```
D:\vs2022\server\x64\Debug\server.exe
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: 3147, ACK号: 162, 校验和: 2514
=====
-----进行数据接收-----
[来自客户端]收到数据包的序列号: 162, ACK号: 3148, 校验和: 48155
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: 3148, ACK号: 163, 校验和: 2512
=====
-----进行数据接收-----
[来自客户端]收到数据包的序列号: 163, ACK号: 3149, 校验和: 48153
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: 3149, ACK号: 164, 校验和: 2510
=====
-----进行数据接收-----
[来自客户端]收到数据包的序列号: 164, ACK号: 3149, 校验和: 44405
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: 3150, ACK号: 165, 校验和: 2508
=====
接收到FILE END, 准备发送ACK...
[来自客户端]收到数据包的序列号: 165, ACK号: 9, 校验和: 37009
[服务端]发送数据包的序列号: 3151, ACK号: 166, 校验和: 2506
已发送ACK, 本次文件接收完毕

D:\vs2022\client\x64\Debug\client.exe
当前状态: 拥塞避免状态
ackNumber:164
begin:1648640
收到 ACK: 165
窗口前移, 新的 begin: 1648640, end: 1655807
收到新的确认ack: 165
rtt_countl要达到的count:0
窗口resize
当前窗口大小: 47
当前状态: 拥塞避免状态
ackNumber:165
begin:1658880
所有数据发送完成!
文件传输完毕!
文件大小: 1.5791 MB
传输时间: 0.451985 秒
吞吐率: 3.49371 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: 165, ACK号: 9, 校验和: 37009
期望收到的ACK号: 166
实际收到的ACK号: 166
收到数据包的序列号: 3151, ACK号: 166, 校验和: 2506
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):
```

图 4.6: 文件传输的实现效果

通过图 4.6 与图 4.5 的对比可以看出，由于我们在 GBN 机制的基础上新引入了拥塞控制机制，系统能够根据网络的拥塞程度实现滑动窗口大小的动态调整。因此与实验 3-2 中仅使用 GBN 机制和实验 3-1 中仅使用停等机制相比，文件传输速率和吞吐率均得到了进一步提升，并且从图 4.6 中可以明

显看出，由于我们在本机进行文件传输，网络状态较好，很少出现网络拥塞，因此滑动窗口不断增大，提高了文件传输速率。相比而言，在仅采用 GBN 机制的情况下，由于窗口大小是固定的，无法根据网络状况进行动态调整，导致在较好的网络环境中未能充分发挥其潜力。因此，虽然 GBN 机制提高了传输速率和吞吐量，但与引入拥塞控制机制后的效果相比，仍显得有所不足。引入拥塞控制机制后，系统能够根据网络的实时负载情况灵活调整滑动窗口大小，使得在网络良好的情况下能够最大化地利用可用带宽，从而进一步优化传输效率和响应速度。

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;
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 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。

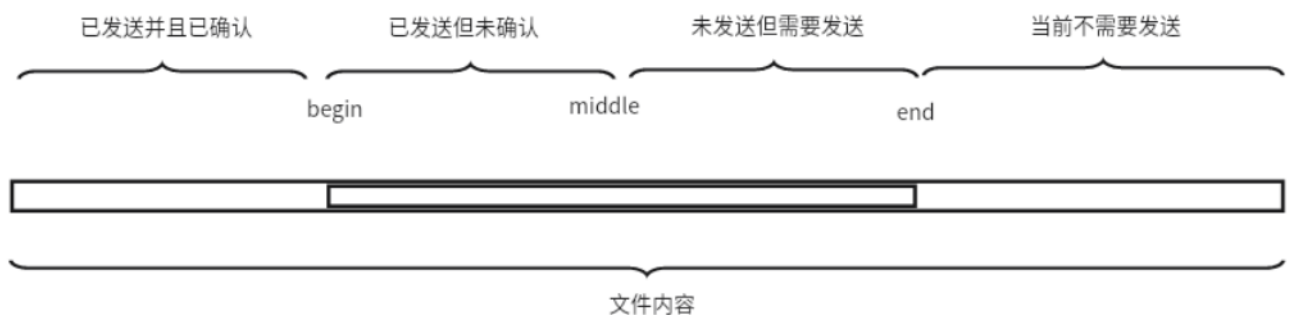


图 4.7: 采用滑动窗口机制进行数据发送

发送端发送线程的具体实现思路如下所示：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。

```

D:\vs2022\server\x64\Debug\server.exe
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: 3384, ACK号: 1147, 校验和: 1292
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1147, ACK号: 3385, 校验和: 18417
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: 3385, ACK号: 1148, 校验和: 1290
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1148, ACK号: 3386, 校验和: 20411
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: 3386, ACK号: 1149, 校验和: 1288
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1149, ACK号: 3387, 校验和: 48416
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: 3387, ACK号: 1150, 校验和: 1286
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1150, ACK号: 3388, 校验和: 21807
校验正确
已写入数据块: 10240 字节
已发送ACK确认!

D:\vs2022\client\x64\Debug\client.exe
begin:11356160
收到 ACK: 1113
ack!!!!
窗口调整, 新的 begin: 11356160, end: 11663359
发送数据包的序列号: 1115, ACK号: 3348, 校验和: 40015
ackNumber:1113
begin:11376640
收到 ACK: 1114
ack!!!!
窗口调整, 新的 begin: 11376640, end: 11683839
发送数据包的序列号: 1116, ACK号: 3349, 校验和: 28758
收到 ACK: ackNumber:1115
begin:11386880
1115
ack!!!!
窗口调整, 新的 begin: 11386880, end: 11694079
发送数据包的序列号: 1117, ACK号: 3350, 校验和: 16874
收到 ACK: ackNumber:1116
begin:11397120
1116
ack!!!!
窗口调整, 新的 begin: 11397120, end: 11704319
收到 ACK: 1117
ack!!!!
发送数据包的序列号: 1118, ACK号: 3352, 校验和: 12140
ackNumber:1117
begin:11417600
收到 ACK: 1118
ack!!!!
窗口调整, 新的 begin: 11417600, end: 11724799

```

图 4.8: 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。

4.4 发送端采取拥塞控制机制动态调整窗口大小

在本实验中，我借鉴了课上所学的 RENO 算法的思想，通过实现拥塞控制机制来优化文件传输过程中的性能。具体实现思路如下：首先，我们为系统设置一个阈值 ssthresh，该值用于决定何时从慢启动阶段转向拥塞避免阶段。

当进行文件传输时，滑动窗口的大小从 1 开始增长，这一阶段我们称为慢启动阶段。在这个阶段，发送端每接收到一个 ack 就将滑动窗口的大小加 1，这种增长方式使得传输速率能够快速提高。一旦窗口的大小达到或超过了阈值 ssthresh，这时系统进入拥塞避免状态。在拥塞避免阶段中，为了控制滑动窗口的增长速度，避免出现拥塞，发送端每接收到 $(middle - begin) / BUFFER_SIZE$ 个 ACK，

将滑动窗口的大小加 1。在上述接收 ack 并增加滑动窗口大小的过程中，一旦出现了超时情况，在一定时间内没有收到 ack，则认为此时网络拥塞程度较高，将窗口大小重置为 1，并且 ssthresh 重置为当前窗口的一半，进入慢启动状态。如果没有出现超时情况，但连续收到了三个重复的 ack 确认号，则认为此时网络出现拥塞，但拥塞程度不高，将窗口大小重置为当前窗口大小的一半，并且 ssthresh 重置为当前窗口的一半加 3，进入拥塞避免状态。通过这种方式，系统能够在检测到较小的拥塞时及时采取措施，避免大规模的性能下降。

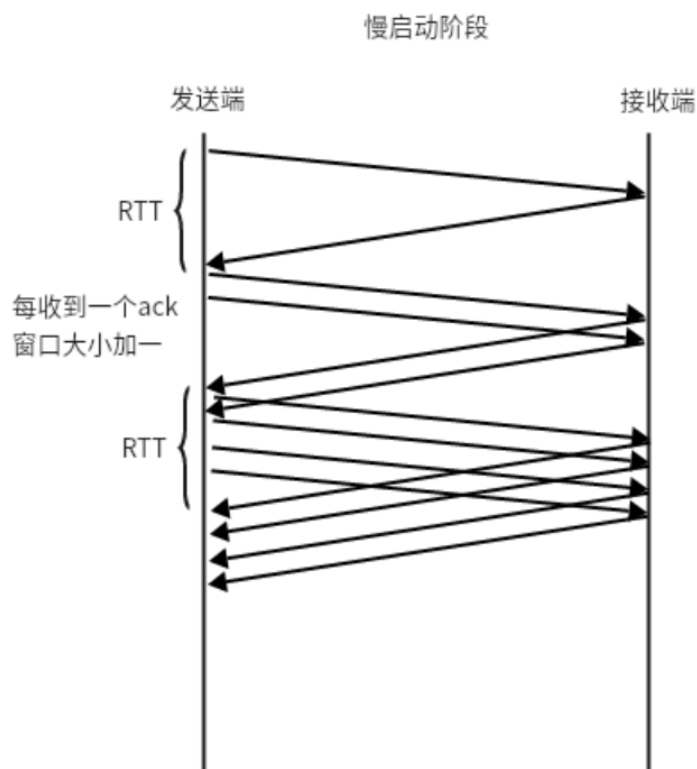


图 4.9: 慢启动阶段

慢启动阶段是拥塞控制中的第一个阶段，其核心思想是通过逐步增加滑动窗口的大小来提高传输速率，从而尽量避免一开始就过度占用带宽造成网络拥塞。在这一阶段，接收端的滑动窗口每接收到一个 ACK 确认报文，窗口的大小便增加 1。这个过程直到滑动窗口的大小达到一个预定的阈值 ssthresh，从而转入拥塞避免阶段。慢启动阶段的增长方式简单直接，能够快速提升传输速率。然而，在网络环境较好的情况下，这一阶段的增长速度可能过快，导致网络拥塞。因此，后续会根据网络反馈逐步调整滑动窗口的大小增长策略。具体代码实现如下所示：

```

C:\选择 D:\vs2022\server\x64\Debug\server.exe
[服务端]发送数据包的序列号: 24, ACK号: 26, 校验和: 5773
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 26, ACK号: 22, 校验和: 37238
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: 25, ACK号: 27, 校验和: 5771
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 27, ACK号: 23, 校验和: 45259
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: 26, ACK号: 28, 校验和: 5769
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 28, ACK号: 23, 校验和: 482
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: 27, ACK号: 29, 校验和: 5767
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 29, ACK号: 23, 校验和: 25766
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: 28, ACK号: 30, 校验和: 5765
=====进行数据接收=====

C:\选择 D:\vs2022\client\x64\Debug\client.exe
当前状态: 慢启动状态
ackNumber:26
begin:235520
收到 ACK: 26
窗口前移, 新的 begin: 235520, end: 389119
发送数据包的序列号: 31, ACK号: 25, 校验和: 15715
收到新的确认ack: 26
窗口resize
当前窗口大小: 16
收到 ACK: 27
当前状态: 慢启动状态
ackNumber:27
begin:245760
收到 ACK: 28
窗口前移, 新的 begin: 245760, end: 409599
发送数据包的序列号: 32, ACK号: 27, 校验和: 60452
ackNumber:28
begin:256000
窗口前移, 新的 begin: 256000, end: 419839
发送数据包的序列号: 33, ACK号: 27, 校验和: 31987
收到 ACK: 29
发送数据包的序列号: 34, ACK号: 28, 校验和: 55028
收到新的确认ack: 29
窗口resize
当前窗口大小: 17
当前状态: 慢启动状态
ackNumber:29
begin:266240
收到 ACK: 30
窗口前移, 新的 begin: 266240, end: 440319

```

图 4.10: 慢启动阶段, 每收到一个 ack 窗口大小加 1

```

1  else
2  {
3      size += 1;
4      setConsoleColor(10);
5      cout << "窗口 resize" << endl;
6      cout << "当前窗口大小: " << size << endl;
7      setConsoleColor(7);
8      if (size >= ssthresh)
9      {
10         setConsoleColor(10);
11         cout << "进入拥塞避免阶段" << endl;
12         cout << size << ssthresh<<"!!!" << endl;
13         setConsoleColor(7);
14         exceed = 1;
15         rtt_count = 0;
16     }
17     else
18     {
19         exceed = 0;
20         setConsoleColor(10);
21         cout << "当前状态: 慢启动状态" << endl;
22         setConsoleColor(7);
23     }
24 }

```

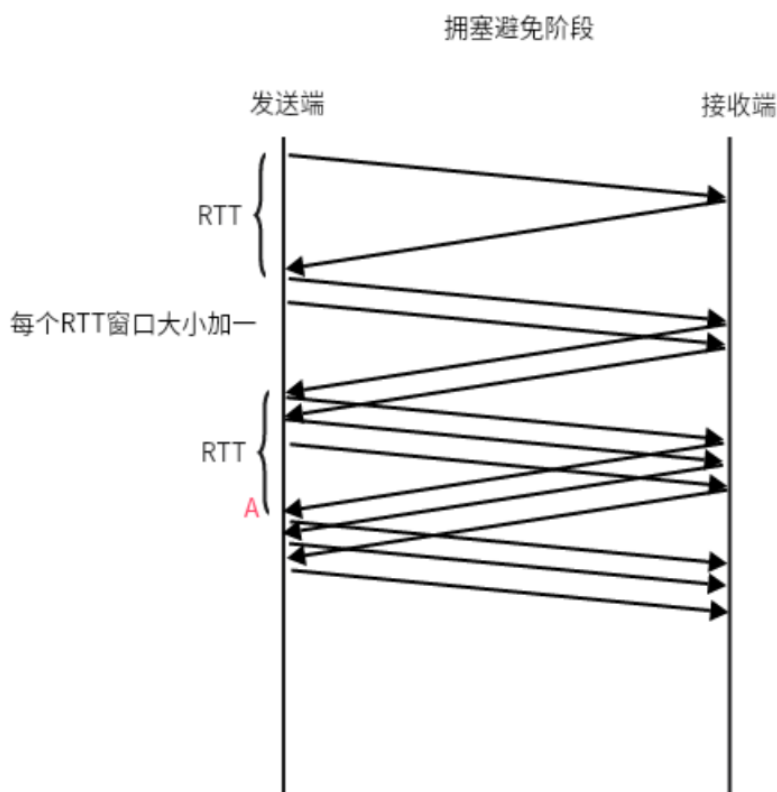


图 4.11: 拥塞避免阶段

与慢启动阶段相比，拥塞避免阶段的目标是更精确地控制滑动窗口的增长速度，避免因窗口过大而导致网络拥塞。具体来说，一旦进入拥塞避免阶段，滑动窗口的大小需要根据网络的实际反馈进行动态调整。这里我借鉴了 RENO 算法的思想并根据实验的实际情况做了一些改进， $(middle - begin)/BUFFER_SIZE$ 是当前窗口大小时发送包的数量，本质就是判断如果发送的数据包都收到了 ack，证明此时网络状态良好，然后将窗口大小加 1。

结合算法的核心思想，进入拥塞避免阶段之后，其实是每收到一个 ack，窗口大小增长 $1/size$ ，这里我选择设置一个 `rtt_count` 变量进行计数，用于记录收到的确认 ack 信号的数量。每当 `rtt_count` 达到 $(middle - begin)$ 个数据包时，就将窗口大小加 1，并且重置 `rtt_count`。

这里以图 4.10 中的 A 点进行算法思路的解释，在 A 点，这时发送端发送了三个数据包，假设此时还没有收到返回的 ack 确认信号，也就是说此时 `begin` 与 `end` 之间有三个数据包（当前窗口大小为 3），`begin` 与 `middle` 之间也有 3 个数据包（代表已发送但未确认），此时只有当系统收到 $(middle - begin)$ 个确认信号也就是 3 个 ack 信号时，才会将窗口大小加 1。这时收到了第一个返回的 ack 确认信号，`rtt_count` 加 1，由于接收线程和发送线程是两个线程并行实现的，此时发送线程窗口前移，接着发送了下一个数据包。此时，由于窗口前移，因此 `begin` 与 `end` 同时前移一个数据包，由于接收到了一个 ack，且发送了下一个数据包，`middle` 也前移一个数据包。当接收到 A 下面第二个返回的 ack 确认信号后， $(middle - begin)$ 依然为 3 个数据包，并将 `rtt_count` 加 1。只有当 `rtt_count` 大于等于 $(middle - begin)/BUFFER_SIZE$ ，也就是上一个窗口 `size` 发送的数据包全都收到之后，这时我们认为网络状态良好，将窗口大小加 1。

具体的代码实现如下所示：


```

D:\vs2022\server\x64\Debug\server.exe
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: 206, ACK号: 153, 校验和: 5464
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 153, ACK号: 201, 校验和: 16313
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: 207, ACK号: 154, 校验和: 5462
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 154, ACK号: 201, 校验和: 39368
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: 208, ACK号: 155, 校验和: 5460
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 155, ACK号: 202, 校验和: 30578
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: 209, ACK号: 156, 校验和: 5458
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 156, ACK号: 203, 校验和: 43601
校验正确

D:\vs2022\client\x64\Debug\client.exe
158, ACK号: 205, 校验和: 50539
收到新的确认ack: 152
rtt_count7要达到的count:8
ackNumber:152
begin:1525760
窗口前移, 新的 begin: 1525760, end: 1607679
收到 ACK: 153
发送数据包的序列号: 159, ACK号: 207, 校验和: 33630
ackNumber:153
begin:1536000
收到 ACK: 154
窗口前移, 新的 begin: 1536000, end: 1617919
收到 ACK: 155
发送数据包的序列号: 160, ACK号: 209, 校验和: 23387
收到 ACK: 156
收到新的确认ack: 157
rtt_count8要达到的count:8
收到 ACK: 157
窗口resize
当前窗口大小: 9
当前状态: 拥塞避免状态
收到 ACK: 158
ackNumber:159
begin:1597440
收到 ACK: 159
窗口前移, 新的 begin: 1597440, end: 1689599
收到 ACK: 160
发送数据包的序列号: 161, ACK号: 214, 校验和: 51111
收到 ACK: 161

```

图 4.12: 拥塞避免阶段, 上一个窗口 size 发送的数据包全都收到之后窗口大小加 1

```

1  if (exceed)
2  {
3      rtt_count++;
4      cout << "rtt_count" << rtt_count<<"要达到的 count:"<<
5      (middle - begin) / BUFFER_SIZE << endl;
6      if (rtt_count >= (middle - begin) / BUFFER_SIZE)
7      {
8          size += 1;
9          rtt_count = 0;
10         setConsoleColor(10);
11         cout << "窗口 resize" << endl;
12         cout << "当前窗口大小: " << size << endl;
13         cout << "当前状态: 拥塞避免状态" << endl;
14         setConsoleColor(7);
15     }
16 }

```

同时, 发送线程还需要在超时情况发生或者连续收到三个重复的 ack 确认信号时进行阶段的调整, 并对窗口大小 size 和阈值大小 ssthresh 进行调整。具体来说: 当发生超时事件时, 意味着网络出现了严重的拥塞或丢包, 发送端会重置窗口大小为 1, 并将阈值 ssthresh 设置为当前窗口的一半, 重新进入慢启动阶段以恢复网络的稳定性。而在连续收到三个重复的 ACK 时, 表示发生了局部丢包, 但网络拥塞程度较低, 发送端会执行快速重传机制, 此时会将 ssthresh 设置为窗口的一半, 并将窗口大小重置为 ssthresh + 3, 进入拥塞避免阶段。通过这种方式, 发送端能够灵活地调整数据传输过程中的窗口大小和拥塞阈值, 从而在保证网络稳定的同时提高传输效率。

```

=====进行数据接收=====
来自客户端]收到数据包的序列号: 43, ACK号: 35, 校验和: 41464
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: 42, ACK号: 44, 校验和: 5737
=====进行数据接收=====
来自客户端]收到数据包的序列号: 47, ACK号: 37, 校验和: 62560
已重新发送ACK确认!
服务端]发送数据包的序列号: 43, ACK号: 44, 校验和: 5736
=====进行数据接收=====
来自客户端]收到数据包的序列号: 48, ACK号: 37, 校验和: 62120
已重新发送ACK确认!
服务端]发送数据包的序列号: 44, ACK号: 44, 校验和: 5735
=====进行数据接收=====
来自客户端]收到数据包的序列号: 50, ACK号: 38, 校验和: 58835
已重新发送ACK确认!
服务端]发送数据包的序列号: 45, ACK号: 44, 校验和: 5734
=====进行数据接收=====
来自客户端]收到数据包的序列号: 51, ACK号: 39, 校验和: 43163
已重新发送ACK确认!
服务端]发送数据包的序列号: 46, ACK号: 44, 校验和: 5733
=====进行数据接收=====
begin:399360
窗口前移, 新的 begin: 399360, end: 604159
收到 ACK: 43
发送数据包的序列号: 54, ACK号: 42, 校验和: 6695
收到新的确认ack: 43
rtt_count4要达到的count:13
ackNumber:43
begin:409600
收到 ACK: 44
窗口前移, 新的 begin: 409600, end: 614399
发送数据包的序列号: 55, ACK号: 43, 校验和: 683
ackNumber:44
begin:419840
收到 ACK: 44
窗口前移, 新的 begin: 419840, end: 624639
发送数据包的序列号: 56, ACK号: 44, 校验和: 18689
ackNumber:44
begin:419840
窗口前移, 新的 begin: 419840, end: 624639
收到 ACK: 44
触发快速重传机制
出现丢包情况, 但拥塞不严重, 将ssthresh重置为size/2, 将窗口重置为 ssthresh + 3
当前的ssthresh: 10当前窗口大小: 13
收到 ACK: 44
进入拥塞避免阶段
收到新的确认ack: 44
ackNumber:44
begin:419840

```

图 4.13: 连续收到三个重复的 ACK 时调整滑动窗口大小

进行拥塞控制的总体实现代码如下所示，进行了上述各种情况的处理：

```

1 while (begin < fileSize) {
2     if (is_timeout) {
3         setConsoleColor(12);
4         cout << "触发超时重传机制" << endl;
5         setConsoleColor(7);
6         middle = begin;
7         is_timeout = false;
8         setConsoleColor(12);
9         cout << "检测到拥塞情况, 将阈值重置为 size/2, 窗口大小重置为 1" << endl;
10        setConsoleColor(7);
11        ssthresh = size / 2;
12        size = 1;
13        setConsoleColor(10);
14        cout << "当前窗口大小: " << size << "当前的 ssthresh: " << ssthresh << endl;
15        setConsoleColor(7);
16        exceed = 0;
17    }
18    if (ackReceived) {
19        if ((ackNumber - 3) * BUFFER_SIZE < begin) {
20            continue;
21        }
22        ackReceived = false;
23        if (ackNumber != ackHistory[ackCount - 1])
24        {

```

```
25     cout << "收到新的确认 ack: "<<ackNumber << endl;
26     if (exceed)
27     {
28         rtt_count++;
29         cout << "rtt_count" << rtt_count<<"要达到的 count:"
30         << (middle - begin) / BUFFER_SIZE << endl;
31         if (rtt_count >= (middle - begin) / BUFFER_SIZE)
32         {
33             size += 1;
34             rtt_count = 0;
35             setConsoleColor(10);
36             cout << "窗口 resize" << endl;
37             cout << "当前窗口大小: " << size << endl;
38             cout << "当前状态: 拥塞避免状态" << endl;
39             setConsoleColor(7);
40         }
41     }
42     else
43     {
44         size += 1;
45         setConsoleColor(10);
46         cout << "窗口 resize" << endl;
47         cout << "当前窗口大小: " << size << endl;
48         setConsoleColor(7);
49         if (size >= ssthresh)
50         {
51             setConsoleColor(10);
52             cout << "进入拥塞避免阶段" << endl;
53             cout << size << ssthresh<<"!!!!" << endl;
54
55             setConsoleColor(7);
56             exceed = 1;
57             rtt_count = 0;
58         }
59         else
60         {
61             exceed = 0;
62             setConsoleColor(10);
63             cout << "当前状态: 慢启动状态" << endl;
64             setConsoleColor(7);
65         }
66     }
```

```
67     }
68     cout << "ackNumber:" << ackNumber << endl;
69     begin = (ackNumber - 3) * BUFFER_SIZE;
70     cout << "begin:" << begin << endl;
71     if (begin >= fileSize) {
72         cout << "所有数据发送完成！" << endl;
73         finish = 1;
74         return true;
75     }
76     end = min(begin + size * BUFFER_SIZE - 1, fileSize - 1);
77     setConsoleColor(10);
78     cout << "窗口前移，新的 begin: " << begin << ", end: " << end << endl;
79     setConsoleColor(7);
80     ackHistory[ackCount++] = ackNumber;
81     if (ackCount >= 3) {
82         ackCount -= 3;
83     }
84     if (ackHistory[0] == ackHistory[1] && ackHistory[1] == ackHistory[2]
85         && ackHistory[0] != 0) {
86         setConsoleColor(12);
87         cout << "触发快速重传机制" << endl;
88         setConsoleColor(7);
89         if (size >= ssthresh)
90         {
91             setConsoleColor(12);
92             cout << "出现丢包情况，但拥塞不严重，将 ssthresh 重置为 size/2，
93             将窗口重置为 ssthresh + 3" << endl;
94             setConsoleColor(7);
95             ssthresh = size / 2;
96             //cout << "当前的 ssthresh: " << ssthresh << endl;
97             size = size / 2 + 3;
98             setConsoleColor(10);
99             cout << "当前的 ssthresh: " << ssthresh << "当前窗口大小: " << size << endl;
100            setConsoleColor(7);
101            exceed = 1; //进入拥塞避免阶段
102            rtt_count = 0;
103            setConsoleColor(10);
104            cout << "进入拥塞避免阶段" << endl;
105            setConsoleColor(7);
106            ackHistory[0] = 0;
107            ackHistory[1] = 0;
108            ackHistory[2] = 0;
```

```

109         ackCount = 0;
110     }
111     middle = begin;
112     continue;
113 }
114 }
115 }

```

4.5 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。

```

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 }

```

4.6 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：

```

D:\vs2022\server\x64\Debug\server.exe
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: 3282, ACK号: 1133, 校验和: 1408
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1133, ACK号: 3283, 校验和: 60760
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: 3283, ACK号: 1134, 校验和: 1406
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1135, ACK号: 3284, 校验和: 24377
已重新发送ACK确认!
[服务端]发送数据包的序列号: 3284, ACK号: 1134, 校验和: 1405
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1136, ACK号: 3284, 校验和: 13246
已重新发送ACK确认!
[服务端]发送数据包的序列号: 3285, ACK号: 1134, 校验和: 1404
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1137, ACK号: 3285, 校验和: 64313
已重新发送ACK确认!
[服务端]发送数据包的序列号: 3286, ACK号: 1134, 校验和: 1403
=====进行数据接收=====
[来自客户端]收到数据包的序列号: 1134, ACK号: 3287, 校验和: 52599
校验正确

D:\vs2022\client\x64\Debug\client.exe
begin:11571200
窗口调整, 新的 begin: 11571200, end: 11909119
收到 ACK: 1134
ack!!!!
发送数据包的序列号: 1134, ACK号: 3284, 校验和: 52602
数据包丢弃, 序列号: 1134
ackNumber:1134
begin:11581440
窗口调整, 新的 begin: 11581440, end: 11919359
发送数据包的序列号: 1135, ACK号: 3284, 校验和: 24377
发送数据包的序列号: 1136, ACK号: 3284, 校验和: 13246
收到 ACK: 1134
ack!!!!
发送数据包的序列号: 1137, ACK号: 3285, 校验和: 64313
ackNumber:1134
begin:11581440
收到 ACK: 1134
ack!!!!
窗口调整, 新的 begin: 11581440, end: 11919359
触发快速重传机制
ackNumber:1134
begin:11581440
收到 ACK: 1134
ack!!!!
窗口调整, 新的 begin: 11581440, end: 11919359
发送数据包的序列号: 1134, ACK号: 3287, 校验和: 52599
ackNumber:1134
begin:11581440
窗口调整, 新的 begin: 11581440, end: 11919359
发送数据包的序列号: 收到 ACK: 1135

```

图 4.14: 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：

```
=====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 5.15: 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：


```

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 5.16: 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：

```

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 5.17: 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 引入拥塞控制后的 GBN 机制文件传输性能提升分析

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仅采用 GBN 机制进行流量控制与加入拥塞控制后 GBN 机制的文件的传输时间，如下表所示。

传输文件大小 (MB)	GBN 机制	size=10(s)	size=20(s)	size=30(s)	size=40(s)	拥塞控制机制 (s)
1.5791		0.6374	0.5690	0.5473	0.5360	0.5015
1.7713		0.6814	0.6099	0.6163	0.6101	0.5455
5.6525		2.0760	1.7181	1.6506	1.7137	1.5847
11.4145		3.3383	3.2653	3.2475	3.1113	3.1915

表 1: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

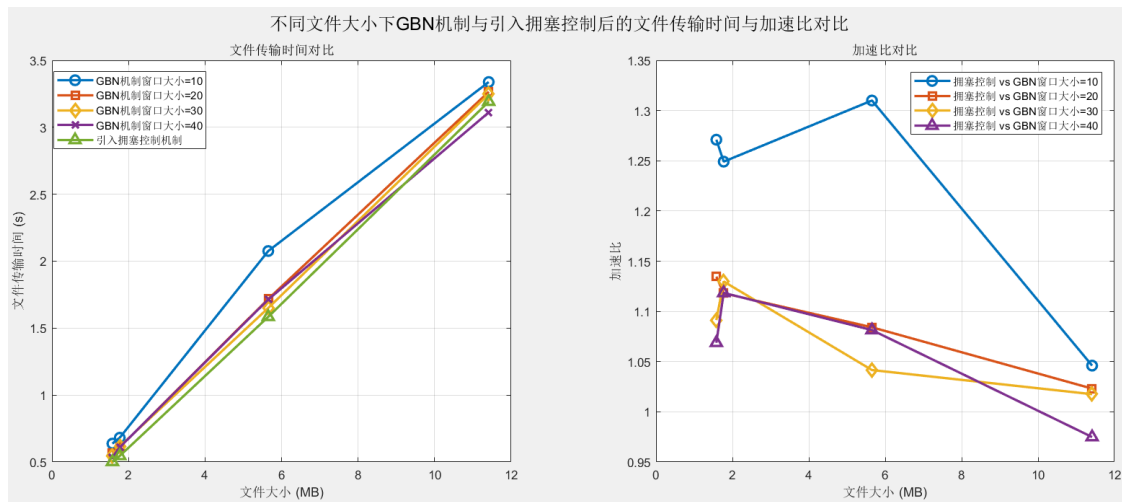


图 6.18: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间及加速比对比

传输文件大小 (MB)	GBN 机制	size=10(MB/s)	size=20	size=30	size=40	拥塞控制机制
1.5791		2.477	2.775	2.885	2.945	3.148
1.7713		2.599	2.904	2.873	2.903	3.247
5.6525		2.710	3.274	3.408	3.282	3.567
11.4145		3.419	3.496	3.514	3.668	3.576

表 2: 不同文件大小下采用 GBN 机制与引入拥塞控制时的吞吐率大小对比

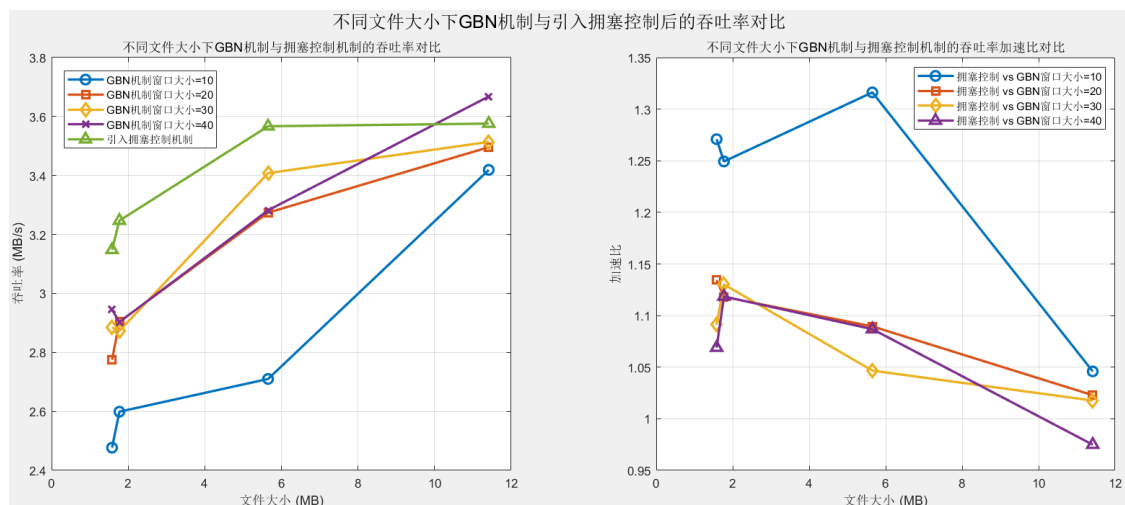


图 6.19: 不同文件大小下采用 GBN 机制与引入拥塞控制时的吞吐率大小及加速比对比

由表中数据和图中加速比对比可以发现, 尽管 GBN 机制相较于停等机制已经取得了显著的性能提升, 在 GBN 机制的基础上引入拥塞控制后, 传输速率依然得到了进一步的优化。由于我们在本机进行传输, 往返时延较小, 并且发送线程与接收线程两个线程并行工作, 在不设置丢包率的情况下, 几乎不会出现丢包的情况。因此滑动窗口大小在传输的过程中是不断增大的。这种情况下, 当 GBN 机制采用较小的滑动窗口时, 由于拥塞控制机制能够根据实时的网络反馈调整窗口大小, 不断增大滑动窗口的大小, 进而提高了传输速率。而 GBN 机制由于滑动窗口大小是固定的, 无法自适应调整, 因此在带宽利用上存在一定的局限性, 无法充分发挥网络带宽的潜力。

从表 1 中的实验数据可以看出, 采用拥塞控制后, 相较于窗口较小的 GBN 机制, 传输速率得到了显著提升, 加速比大概在 1.15 左右。尤其在文件传输较小或中等大小的情况下, 拥塞控制机制能够有效提升传输速率。这表明, 拥塞控制机制在动态调整窗口大小方面的优势可以更好地适应网络状况, 优化带宽利用。只有当传输文件大小较大时, 例如文件大小为 11.4145 MB 时, 引入拥塞控制相比窗口大小固定为 40 时的 GBN 机制, 带来了性能降低。这是因为此时传输的文件较大, GBN 机制采取固定 40 大小的滑动窗口, 而拥塞控制则需要等待一定时间窗口才能达到这个程度, 虽然拥塞控制能够根据网络拥塞情况实时调整窗口大小, 但在文件较大的情况下, 这一调整的过程较慢, 会导致初期的传输速率较低。而如果在网络拥塞的情况下, 由于拥塞控制及时调整了窗口大小, 因此发生超时情况时需要重传的数据包少, 引入拥塞控制可以实现更大的性能提升, 后面我们会继续具体研究。

表 2 和图 6.19 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, 引入拥塞控制后, 相比滑动窗口较小的 GBN 机制, 传输速率得到了显著提升。而在 GBN 机制窗口较大时, 引入拥塞控制由于初期调整增大窗口大小所需的时间较长, 导致初期的传输速率较低, 因此表现出一定的性能降低。但总体来说, 拥塞控制机制能够有效提升吞吐率和传输效率, 通过动态调整滑动窗口大小, 在不同的网络环境下提供了更为稳定和高效的传输性能。

6.2 丢包率为 5% 时 GBN 机制与引入拥塞控制时的文件传输性能对比

为了探究在出现丢包情况时, 仅采用 GBN 机制进行流量控制与加入拥塞控制后 GBN 机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)	GBN 机制 (s)	size=10	size=20	size=30	size=40	拥塞控制机制 (s)
1.5791	0.6493	0.5737	0.6109	0.6256	0.5664	
1.7713	0.7681	0.6734	0.7033	0.7001	0.6112	
5.6525	1.9455	2.0901	1.9527	1.8048	1.6784	
11.4145	4.1254	3.8465	3.6719	3.6728	3.3256	

表 3: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

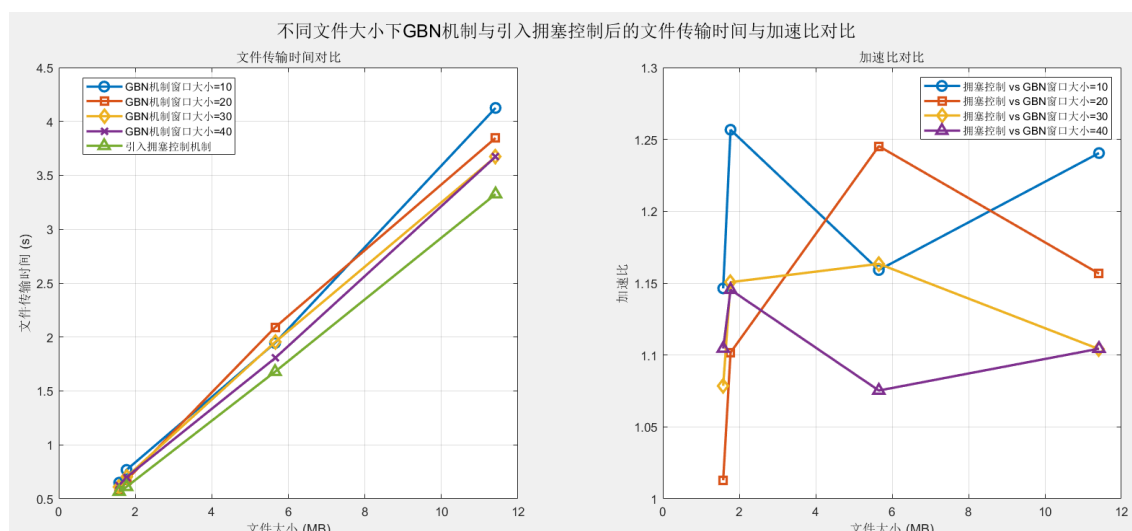


图 6.20: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间及加速比对比

由图 6.20 中的加速比对比我们可以发现, 当出现丢包情况时, 引入拥塞控制机制进一步提高了滑动窗口文件传输的速率, 加速比大约为 1.2 左右, 表明拥塞控制机制在提升传输性能方面发挥了重要作用。当 GBN 机制采取较小的滑动窗口时, 由于拥塞控制机制可以根据网络状况快速调整增大窗口大小, 从而实现了较高的传输速率。相比之下, GBN 机制由于窗口大小固定, 无法根据网络状况自适应调整, 因此在网络状况良好时, 传输速率会受到固定窗口限制, 未能充分利用网络带宽。当 GBN 机制采取较大的滑动窗口时, 尽管拥塞控制机制需要一段时间来增大窗口大小, 但在传输过程中发生丢包情况需要进行重传时, 特别是当发送端连续三次收到重复的 ack 确认信号时, 拥塞控制机制可以根据网络状况适当降低滑动窗口大小, 使得下一次发生超时的概率降低, 并且发生超时或丢包时, 由于窗口大小的降低, 因此需要重传的数据包也减少了, 进而提高了整体传输速率。与此相比, GBN 机制在发生丢包时, 必须重传当前窗口内所有未确认的数据包, 这导致了额外的开销, 特别是在网络不稳定或丢包较为频繁的情况下, 传输效率大大降低。GBN 机制的固定窗口无法动态适应网络变化, 导致无法有效应对丢包带来的性能损失。

因此, 引入拥塞控制机制能够在丢包情况下有效优化流量控制, 依据实时网络状况动态调整窗口大小, 从而显著提升传输速率。通过减少重传次数和优化带宽的利用, 拥塞控制机制不仅能够在丢包环境下提高传输效率, 还能避免网络拥塞和过度传输, 最终实现更为稳定、高效的文件传输。与仅依赖 GBN 机制的方案相比, 拥塞控制机制为文件传输提供了更加灵活和可靠的性能保障。

6.3 延时为 10ms 时 GBN 机制与引入拥塞控制时的文件传输性能对比

为了探究在出现延迟的情况时, 仅采用 GBN 机制进行流量控制与加入拥塞控制后 GBN 机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)	GBN 机制 (s)	size=10	size=20	size=30	size=40	拥塞控制机制 (s)
1.5791	3.2124	3.0141	2.8842	2.6491	2.4031	
1.7713	3.2684	3.0951	2.9431	2.7498	2.4911	
5.6525	11.4592	10.8149	10.4304	9.1516	8.4452	
11.4145	25.2694	23.0277	22.7006	19.8793	16.8140	

表 4: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

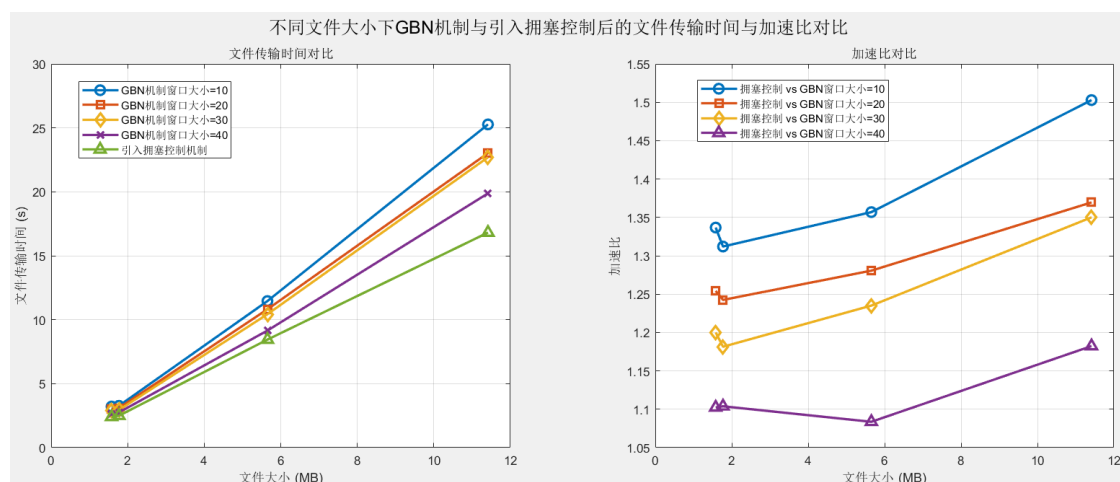


图 6.21: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间及加速比对比

首先, 对于仅采用 GBN 机制进行流量控制与加入拥塞控制后 GBN 机制的传输性能差异, 由图 6.21 中的加速比对比我们可以看出, 在延时为 10ms 的情况下, 引入拥塞控制机制后不同大小文件的传输性能都有所提高, 加速比大概在 1.35 左右。并且随着传输文件大小的增加, 加速比呈现出上升趋势, 这表明拥塞控制机制在处理较大数据量时的优势更加明显。这是因为在延时较大的情况下, 此时延时的持续时间大于发送端发送当前窗口内所有数据包的时间。因此 GBN 机制在发送完当前滑动窗口中待发送的数据包就会进入等待状态, 只有等待延时结束后, 此时之前发送数据包的返回 ack 信号才会到达, 滑动窗口会整体前移一个窗口的大小, 然后继续重复上述过程, 发送数据包并继续等待, 直到所有数据包发送完毕。

而在引入拥塞控制机制后, 系统可以通过动态调整窗口大小来适应网络状况。拥塞控制机制在延时结束接收到所有 ack 信号时, 会相应的根据当前所处的状态 (慢启动或拥塞避免) 来增加滑动窗口的大小, 这种动态调整窗口大小的机制, 可以在有延时的情况下逐步增大发送数据包的速度, 因此上述实验结果显示, 引入拥塞控制机制后不同大小文件的传输性能都有所提高。并且当发送文件较大时, 由于需要发送的数据包较多, 引入拥塞控制带来的性能提升也就越大。总体而言, 拥塞控制机制使得在延时存在的情况下, 发送端可以有效利用网络资源, 提高传输效率。通过逐步增大窗口大小, 拥塞控制机制可以显著减少传输过程中等待时间的累积, 从而带来显著的性能提升。

6.4 不同延时情况下 GBN 机制与引入拥塞控制时的文件传输性能对比

为了探究不同延时情况下 GBN 机制与引入拥塞控制时的文件传输性能对比, 我将文件大小固定为 11.4145MB, 对不同延时情况下仅采用 GBN 机制进行流量控制与加入拥塞控制后 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)	GBN 机制 (s) size=10	size=20	size=30	size=40	拥塞控制机制 (s)
1	19.8958	20.2976	19.7247	17.4527	14.0419
5	22.7304	21.1492	18.0784	16.2273	15.8102
10	25.2694	23.0277	22.7006	19.8793	16.8140
20	42.3379	40.1149	38.1842	35.2957	29.1960

表 5: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

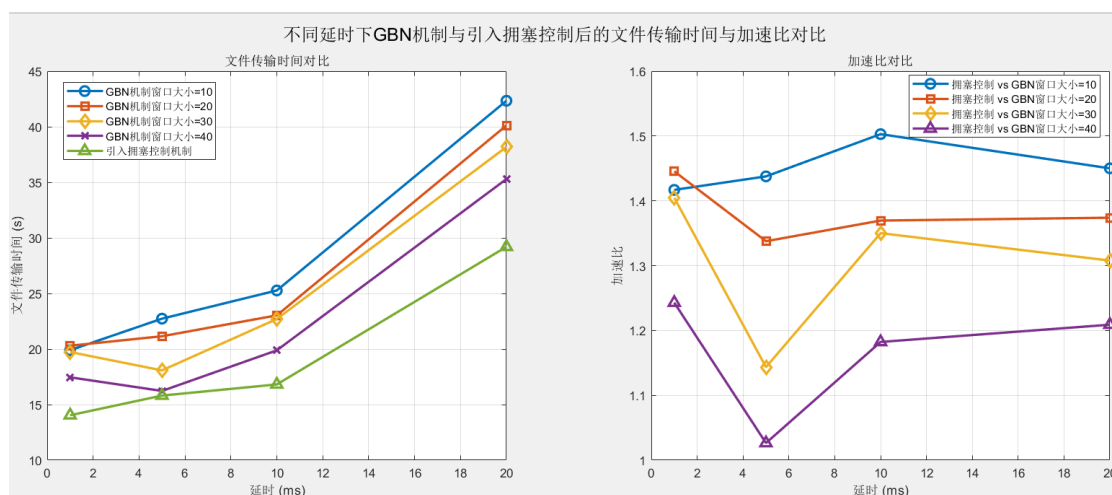


图 6.22: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间及加速比对比

由表 5 中的实验数据和图 6.22 中引入拥塞控制后的加速比对比可以得出与实验 3 类似的结论, 在引入拥塞控制机制后, 由于拥塞控制的引入, 系统能够动态调整滑动窗口的大小来适应网络状况, 在延时结束接收到所有 ack 信号时, 会相应的根据当前所处的状态 (慢启动或拥塞避免) 来增加滑动窗口的大小, 这种动态调整窗口大小的机制, 可以在有延时的情况下逐步增大发送数据包的速度从而减少了因延时带来的等待时间. 使得即便在不同延时情况下, 网络的利用率都得到了有效提升, 尤其在延迟较高的情况下, GBN 机制的等待时间往往会造成明显的性能下降, 而拥塞控制机制通过动态调整窗口大小, 缓解了这种问题。通过逐步增大发送窗口, 拥塞控制机制有效提高了网络资源的利用效率, 使得数据包能够更快地发送, 减少了等待时间的积累。

总体来看, 拥塞控制机制通过减少延时带来的等待时间, 并结合网络状态灵活调整窗口大小, 极大地优化了在不同延时条件下的传输效率。特别是在大文件传输和高延时情况下, 拥塞控制机制所带来的性能提升更加显著, 进一步证明了其在实际网络环境中的重要性。

6.5 不同丢包率情况下采用 GBN 机制与引入拥塞控制时的文件传输性能对比

为了探究不同丢包率情况下仅采用 GBN 机制进行流量控制与加入拥塞控制后 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采用 GBN 机制与引入拥塞控制时的文件传输性能进行了测试, 实验数据如下表所示:

丢包率 (%)	GBN 机制 (s)	size=10	size=20	size=30	size=40	拥塞控制机制 (s)
1	3.6864	3.5364	3.3214	3.1403	3.2029	
5	4.1254	3.8465	3.6719	3.6728	3.3256	
10	4.1654	3.9024	4.0027	4.5754	3.6275	
20	5.5317	5.7621	5.8853	6.0621	5.0128	
30	8.1107	9.7249	9.9236	10.2473	7.5957	

表 6: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

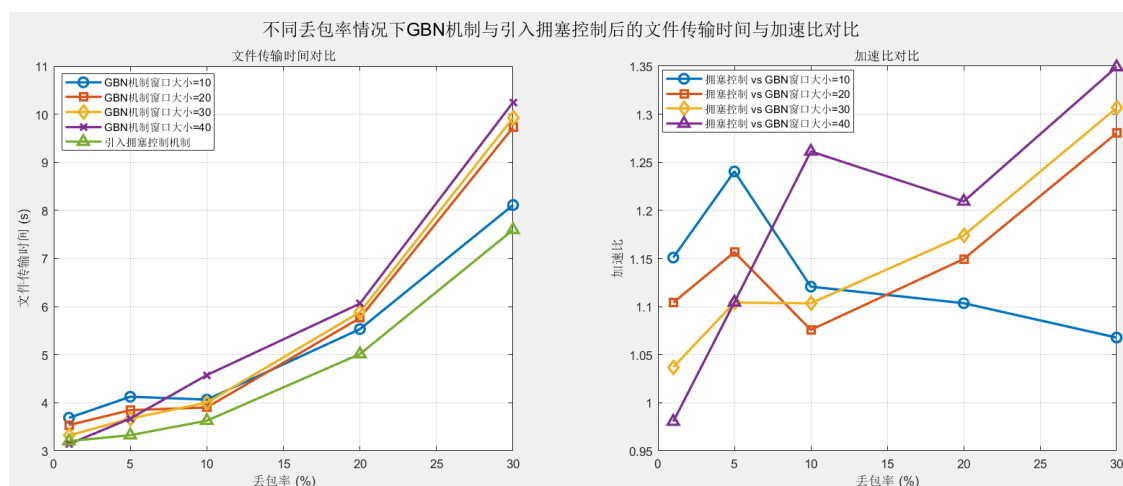


图 6.23: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间及加速比对比

由图 6.23 中不同丢包率下引入拥塞控制机制的加速比对比可以看出,随着丢包率的提高,引入拥塞控制机制相比仅采用 GBN 流量控制机制带来了显著的性能提升,加速比大约在 1.20 左右,并且随着丢包率的增加,引入拥塞控制机制所带来的加速比呈现出上升趋势。这是因为在网络出现丢包的情况下,拥塞控制机制通过动态调整滑动窗口的大小,能有效缓解因丢包导致的性能下降。当网络丢包时,特别是当连续收到三次重复的 ACK 确认信号时,拥塞控制机制能够通过适当减小滑动窗口的大小来降低后续再次发生超时的概率,同时窗口大小的减小也减少了下次发生超时或丢包时需要重传的数据包。不仅减少了因重传引发的额外开销,也提高了整体的传输速率。

与之相比,GBN 机制在发生丢包时,必须重传当前窗口内所有未确认的数据包,这会导致大量的无效重传,尤其是在网络不稳定或丢包较为频繁时,传输效率急剧下降。由于 GBN 机制的窗口大小是固定的,无法根据网络状况进行动态调整,因此其在面对丢包或超时事件时的适应性较差,导致无法有效地应对丢包带来的性能损失。

综上所述,在丢包率较高的情况下,拥塞控制机制相比于 GBN 机制能够显著提高文件传输性能,尤其是在丢包率较高时,拥塞控制机制的优势愈加明显。通过动态调整窗口大小,拥塞控制机制使得系统能够更好地适应丢包引起的网络波动,从而保持较高的传输效率。

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,对比仅采用 GBN 机制进行流量控制与加入拥塞控制后 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
size=10	3.3383	3.6864	4.1254	4.1654	19.8958	25.2694	42.3379
size=20	3.2653	3.5364	3.8465	3.9024	20.2976	23.0277	40.1149
size=30	3.2475	3.3214	3.6719	4.0027	17.7247	22.7006	38.1842
size=40	3.1113	3.1403	3.6728	4.5754	17.4527	19.8793	35.2957
拥塞控制机制	3.1915	3.2029	3.3256	3.6275	14.0419	16.8140	26.1960

表 7: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

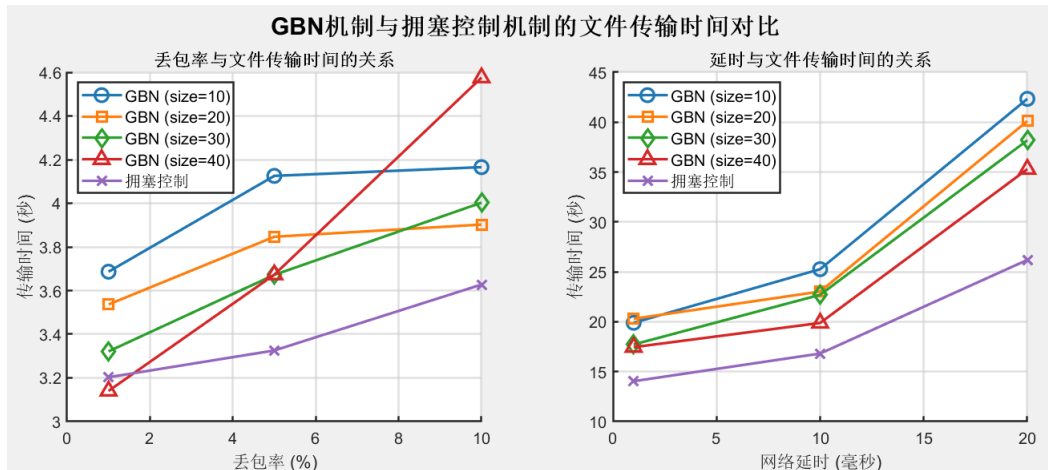


图 6.24: 不同文件大小下采用 GBN 机制与引入拥塞控制时的文件传输时间对比

由表 7 中的实验数据和图 6.24 中 GBN 机制与拥塞控制机制的传输性能对比我们可以发现，引入拥塞控制机制显著增强了系统在复杂网络环境中的应对能力。具体而言，在丢包率较高和网络延时较大的环境中，拥塞控制机制通过动态调整网络带宽，有效缓解了丢包和延时对传输性能的负面影响，从而显著提高了文件的传输速率。

对于仅采用 GBN 机制的传输方式，随着丢包率的增加，传输时间显著上升，尤其在较大的窗口大小下表现尤为明显，由左图可以看出当窗口大小为 40 时，在丢包率较低的情况下，文件传输速率较高，甚至超过了拥塞控制机制。然而，随着丢包率的升高，GBN 机制的传输速率急剧下降，最终变为最慢的传输方式。这是因为丢包率的增加导致重传次数的增加，而由于窗口较大，导致每次发生超时或丢包情况时需要重传的数据包较多，从而延长了文件的整体传输时间。相比之下，加入拥塞控制机制后，即使在较高丢包率（10%）的情况下，传输时间的增长幅度显著降低。这表明拥塞控制机制能够更有效地利用网络资源，减少因丢包造成的性能损失。

在存在网络延时的环境中，GBN 机制的传输时间随着延时的增加呈现出快速增长的趋势，尤其在较小窗口大小的情况下，传输时间的增长更加显著。这是因为较小的窗口无法充分利用网络带宽，导致大量的等待时间和低效的数据传输，进而使得文件的整体传输过程变得更加缓慢。相比之下，拥塞控制机制由于能够根据实时网络状态动态调整滑动窗口大小，发送端可以有效利用网络资源，提高传输效率。在网络延时较高时，拥塞控制机制通过逐步增大窗口大小，减少了数据传输过程中的等待时间，从而显著降低了延时带来的负面影响。尤其是在延时较大的情况下，拥塞控制机制能保持较低的传输时间增长幅度，显著优于 GBN 机制。这表明，拥塞控制机制在延时环境下具有较高的适应性，能够有效应对网络延迟的变化。

综上所述，引入拥塞控制机制显著提高了系统在复杂网络环境中的应对能力，尤其是在高丢包率或高延时的网络环境下，表现尤为出色。这表明，拥塞控制机制通过动态调整传输策略，不仅可以改善带宽利用率，还能有效缓解网络延时对文件传输性能的负面影响，从而显著提升文件传输的整体性能。

7 实验内容总结

本次实验在实验 3-2 的基础上，通过引入拥塞控制机制，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。我们通过采用拥塞机制动态控制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、拥塞控制、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：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。
- 拥塞控制：本实验引入了拥塞控制机制，旨在优化传输过程中的带宽利用率。该机制通过动态调整发送端的窗口大小，根据网络状态变化实时进行带宽分配，避免了网络拥塞对传输性能的负面影响。实验结果表明，拥塞控制机制能够有效降低传输时延，尤其在丢包率较高和网络延时较大的环境中，表现尤为突出。
- 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。实验结果显示，滑动窗口机制显著提高了吞吐率，特别是在丢包率较高的网络环境中，相比传统的停等机制，滑动窗口机制能有效减少等待时间，提升传输效率。同时，拥塞控制机制通过动态调整窗口大小，显著减少了传输过程中的延迟和丢包，提高了系统的适应性。

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。
- 引入拥塞控制机制能够在丢包情况下有效优化流量控制，依据实时网络状况动态调整窗口大小，从而显著提升传输速率。通过减少重传次数和优化带宽的利用，拥塞控制机制不仅能够在丢包环境下提高传输效率，还能避免网络拥塞和过度传输，最终实现更为稳定、高效的文件传输。与仅依赖 GBN 机制的方案相比，拥塞控制机制为文件传输提供了更加灵活和可靠的性能保障。

通过本次实验，我深入理解了网络传输协议设计中的复杂性，特别是在面对不同网络环境时，如何引入拥塞控制机制以实现最优的性能。引入拥塞控制机制不仅有效应对了丢包情况，还通过实时调整窗口大小和带宽分配来避免过度传输和网络拥塞，从而提升了协议在高延迟或不稳定网络环境下的适应性。拥塞控制的优化使得协议更具灵活性，能够根据实时网络状况调节传输行为，减少了不必要的重传和延迟，保证了更稳定的吞吐率。