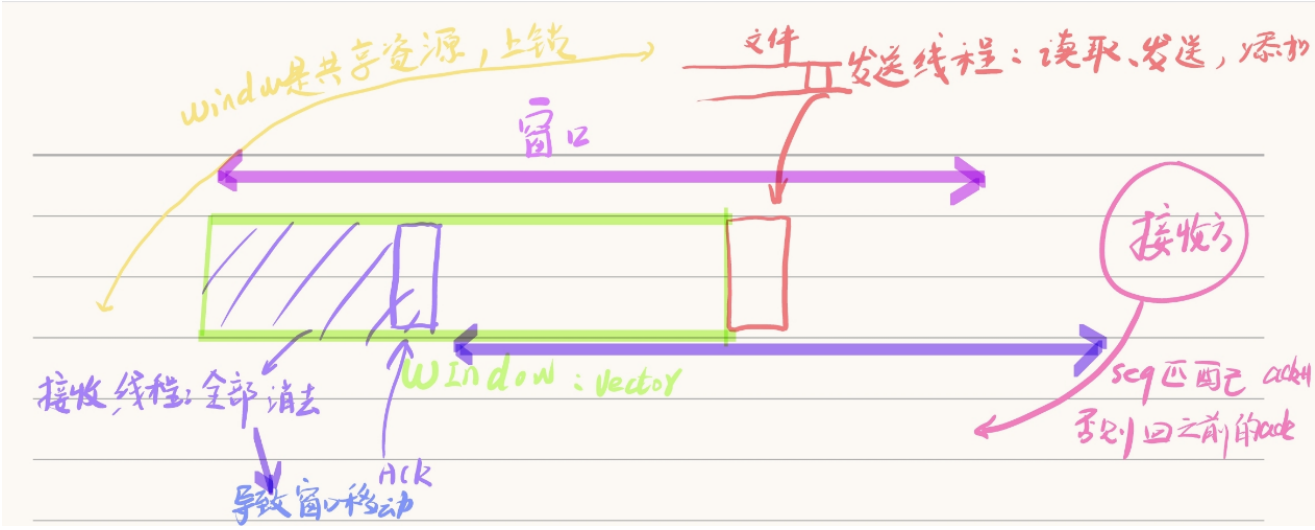
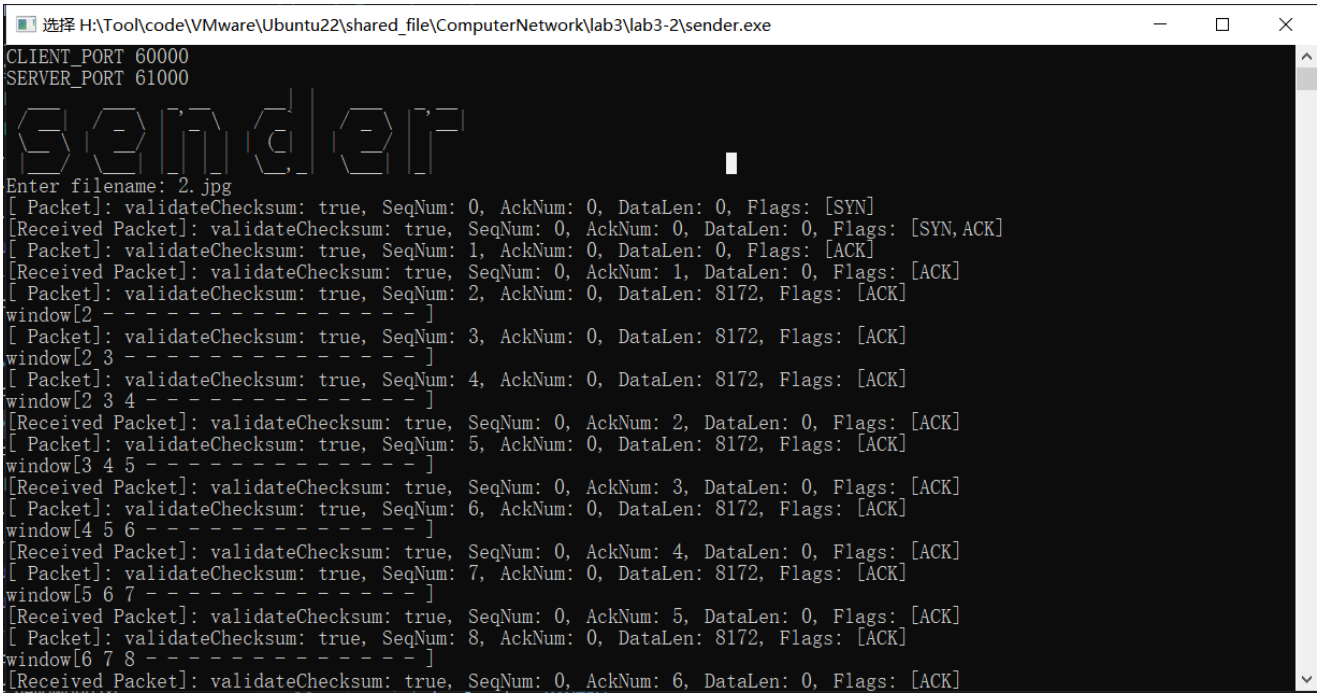
lab3-2实验报告



杜怡兴-2112847



前情提要:协议设计

1. 数据设计

变量名称	含义	长度 (字节)
checksum	校验和	2
seqNum	序列号	4
ackNum	累计确认号	4
dataLen	数据长度	2
flags	标志位	1
packetNum	数据包序号	2
reserved	保留字段	5

- **总长度**: 20字节
- **最大数据长度**: 8172字节

双方代码都需要引入**协议头文件**`protocol.h`

```
struct Packet {
    uint16_t checksum = 0;
    uint32_t seqNum = 0;
    uint32_t ackNum = 0;
    uint16_t dataLen = 0;
    uint8_t  flags = 0;
    uint16_t packetNum = 0;
    char          reserved[5] = {0, 0, 0, 0, 0};
    char          message[8172];
};
#pragma pack(pop)
```

2. 时序设计

四次握手

1. 发送方发送SYN
2. 接收方发送SYN, ACK
3. 发送方发送ACK
4. 接收方发送ACK

四次挥手过程

1. 发送方发送FIN ACK:
2. 接收方发送ACK:
3. 接收方发送FIN ACK:
4. 发送方发送ACK

动态调整 RTO

- 使用 `timeout` 变量表示超时时间**RTO**，初始值为 200 毫秒，最大值为 2000 毫秒，最小值为 100 毫秒。
- 如果**发生超时重传**，则会**增加超时时间 `timeout` 的值**，以延长下一次的等待时间，从而降低丢包的可能性。
- 如果**成功接收**到期望的数据包，会**降低超时时间 `timeout` 的值**，以缩短下一次的等待时间，从而加快数据传输的速度。

差错校验

1. `calculateChecksum` **计算校验和**：
   - 将数据包按16位分组相加，然后取反得到校验和值
2. `setChecksum` **设置校验和**：
   - 清除之前的校验和值并重新计算
3. `validateChecksum` **验证校验和**：
   - 使用 `calculateChecksum` 函数计算校验和，与0比较，相等则校验和有效

超时重传

在等待期间，发送方使用 `recvfrom` 函数来接收来自接收方的响应。此函数是**阻塞的**，意味着它会一直等待，直到有数据到达或者等待超时。**阻塞状态会在两种情况下终止**：

1. **计时超时**：发送方设置了一个超时时间（RTO）,如果超过了这个时间, `recvfrom` 函数结束，发送方会认为接收方没有响应，发送方会触发超时重传机制。
2. **收到包**：收到包时， `recvfrom` 函数结束，发送方会检查接收到的数据包是否满足期望的条件。

3.实验环境

如何编译

前提: **windows安装g++并添加到环境变量**

在含有`receiver.cpp`、`sender.cpp`和`protocol.cpp`的文件夹搜索栏输入**cmd**打开控制台，输入

```
g++ receiver.cpp -o receiver -lws2_32
g++ sender.cpp -o sender -lws2_32
```

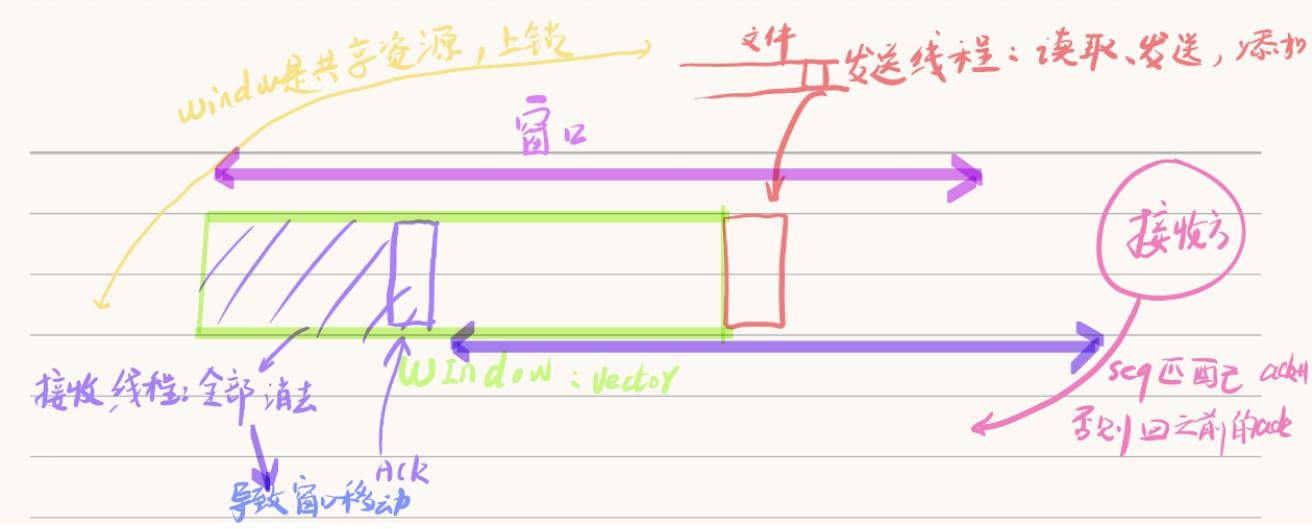
就可以看到生成了**sender.exe**和**receiver.exe**

如何传输

1. 在可执行文件的同一目录下，创建1个文件夹 **source**,将要传输的测试文件放入。
2. **配置好路由器**，**启动** `sender.exe` 和 `receiver.exe` 可执行文件，**输入传输文件名。**
3. 传输完成后，文件将会出现在**根目录**中。

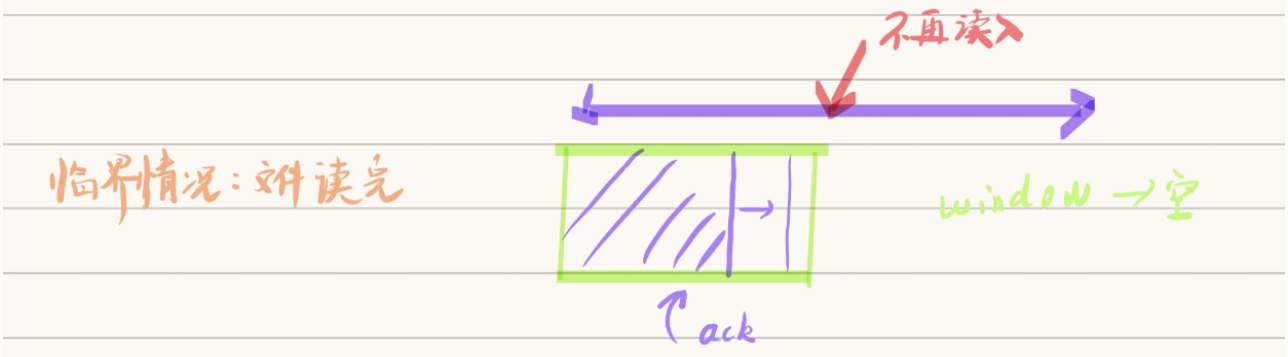
实现思路

滑动窗口的实现细节



在我的实现中，`vector<Packet> window` 被用来实现滑动窗口机制。

- **窗口内容**：**该窗口存储的是正在传输但尚未收到确认的数据包。**这包括了那些已经发送出去但还没有被接收方确认的数据包。
- **窗口大小控制**：窗口的大小是有限制的，即 `window.size()` 小于一个预设的最大值 `N`。`N` 是滑动窗口的大小，它**限制了可以在不收到确认的情况下发送的最大数据包数量。**这个限制是为了控制网络上的数据流量，防止发送方一次性发送过多数据，从而导致网络拥塞或数据丢失。
- **数据包状态分类**：
  - **未读取**：这些是还未被读取和发送的数据包。
  - **发送后未确认**：这些数据包已被发送出去，但还在等待接收方的确认。这些数据包位于 `window` 中。
  - **确认**：一旦数据包被接收方确认，它们就会从 `window` 中移除。
- **窗口操作优雅性**：与固定一个窗口并标记每个数据包的状态（如未读取、发送后未确认）相比，我的方法更为优雅。我将数据包添加到 `window` 中，并在收到确认后移除它们。这不仅使得实现更简洁，而且易于理解和维护，并且逻辑等价。



1.0 可视化

```
[ Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 0, Flags: [SYN]
[Received Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 0, Flags: [SYN,
[ Packet]: validateChecksum: true, SeqNum: 1, AckNum: 0, DataLen: 0, Flags: [ACK]
[Received Packet]: validateChecksum: true, SeqNum: 0, AckNum: 1, DataLen: 0, Flags: [ACK]
[ Packet]: validateChecksum: true, SeqNum: 2, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[2 - - - - - - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 3, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[2 3 - - - - - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 4, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[2 3 4 - - - - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 5, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[2 3 4 5 - - - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 6, AckNum: 0, DataLen: 8172, Flags: [ACK]
[Receivedwindow[2 3 4 5 6 Packet]: validateChecksum: true, SeqNum: 0, AckNum: 2, DataLen:
- - - - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 7, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[3 4 5 6 7 - - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 8, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[3 4 5 6 7 8 - - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 9, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[3 4 5 6 7 8 9 - - - - - - - - - ]
[ Packet]: validateChecksum: true, SeqNum: 10, AckNum: 0, DataLen: 8172, Flags: [ACK]
window[3 4 5 6 7 8 9 10 - - - - - - - - ]
```

在可视化部分，我通过实现 `printWindow` 函数来打印窗口内的数据包。该函数遍历窗口内的数据包，并将它们的序列号打印出来，**如果window大小<N，则打印“-”表示空位置**，从而直观地展示了窗口内的数据包情况。

如图可以看到，**开始时，数据包不断填入窗口的过程。**之后窗口大小将会在N附近调整。

1.1 线程分工

- `senderThread`：负责从文件中读取数据，封装成数据包，并发送。当滑动窗口未满时，继续发送更多数据包。
- `receiverThread`：负责接收对方发送的确认包（ACK），并据此更新滑动窗口。

1.2 sendAndReceive 的逻辑

在 `senderThread` 函数中，我实现了滑动窗口机制来控制数据包的发送。这个函数的核心逻辑如下：

- **检查文件末尾和窗口大小**：线程首先检查是否已经到达文件末尾（`feof(inFile)`），同时确保滑动窗口未满（`window.size() < N`）。这是因为当文件已经读取完毕，且滑动窗口内的数据包数量小于窗口最大大小时，表示没有更多数据需要发送。
- **读取并发送数据包**：如果以上条件满足，线程继续从文件中读取数据，并将其封装为数据包。每读取一段数据，就创建一个新的数据包，并将其发送出去。随后，这个数据包被添加到滑动窗口中，表示该数据包已发送但还未收到确认。
- **线程退出条件**：当文件读取完毕，并且滑动窗口中没有剩余的数据包时（即 `feof(inFile) && window.size() == 0`），线程终止。这表示所有的数据已经被发送并且确认。

```
void senderThread() {
    while (true) {
        // 检查文件末尾和窗口大小，读取并发送数据包，更新滑动窗口
        if (feof(inFile) && window.size() == 0) break;
    }
    cout << "senderThread finished." << endl;
}
```

1.3 receiverThread 的逻辑

- 接收数据包：线程不断接收来自发送方的数据包，这些包含了发送方对接收到数据包的确认信息（ACK）。
- 更新滑动窗口：当收到ACK包时，我根据ACK号来更新滑动窗口。更新的方法是**移除所有序列号小于或等于收到的ACK号的数据包**。这意味着所有这些数据包已被发送方成功接收。

```
void receiverThread() {
    while (true) {
        // 接收确认包逻辑
        if (receivedPacket.ackNum >= window[0].seqNum && receivedPacket.ackNum <=
window.back().seqNum) {
            lock_guard<mutex> lock(windowMutex);
            while (window.size() && window[0].seqNum <= receivedPacket.ackNum) {
                window.erase(window.begin()); // 消去ack以及以前的数据包
            }
        }
        // 其他逻辑
    }
}
```

通过这种方式，`receiverThread` **确保滑动窗口始终包含了那些已发送但尚未被确认接收的数据包。**一旦一个数据包被确认接收，**它和之前的数据包**就会从窗口中移除，从而为新的数据包腾出空间。

2. 使用 mutex 实现线程安全

为了在多线程环境中安全地访问和修改滑动窗口，我使用了 `mutex` 锁。这确保了在**一个线程修改窗口时**，其**他线程不会同时进行读写操作。**lock_guard的一个好处是**自动解锁。**

```
mutex windowMutex;
{//锁只在本作用域有效
	lock_guard<mutex> lock(windowMutex);
	window.push_back(sentPacket);
}
```

3. 快速重传机制

重传就是重传window当中的所有数据包，因为window存储了正在传输但还未收到确认的数据包。因为滑动窗口的大小是N而非window.size()，window.front().seqNum+N才是窗口结尾。

通过跟踪最后一个接收到的ACK号（`lastPacketAck`）和相同ACK号的计数（`samePacketCount`），我实现了快速重传机制。当连续收到超过一定数量的相同ACK时，触发快速重传。

```
int lastPacketAck = 0, samePacketCount = 0;
//核心代码
lastPacketAck=receivedPacket.ackNum;
receive();
if(receivedPacket.ackNum==lastPacketAck)samePacketCount++;
else samePacketCount=0;
if (recvResult < 0 || (samePacketCount > 5 && FastRetransmission)  ) {
//重传
}
```

4. RTO动态调整

通过维护最大和最小超时时间（`maxTimeout` 和 `minTimeout`），我实现了RTO的动态调整。根据网络状况调整超时时间，以适应不同的网络延迟情况。

```
int maxTimeout = 2000 , minTimeout = 100;
//重传
tryCount--,timeout=(timeout<maxTimeout)?timeout+50:timeout,cout << "resending" << endl;
//接收
timeout=(timeout>minTimeout)?timeout-50:timeout;
```

5. 接收方的实现

在接收方的实现中，核心逻辑包括以下几个部分：

1. **确认号（ack）的使用**：变量 `ack` 用于追踪接收方已成功接收并确认的数据包序列号。`ack` 的初始值设置为 `-1`，代表还未接收任何数据包。
2. **数据包接收逻辑**：当接收方收到的数据包序列号 `receivedPacket.seqNum` 等于 `ack + 1` 时，说明这是接收方期望接收的下一个数据包。在这种情况下，接收方会将数据包的内容写入文件，并更新 `ack`，同时发送一个确认（ACK）给发送方。

```
cppCopy codeif (receivedPacket.seqNum == ack + 1) {
    fwrite(receivedPacket.message, 1, receivedPacket.dataLen, outFile), ackLen +=
receivedPacket.dataLen;
    sentPacket = Packet(0, ++ack, 0, ACK, "");
    send();
}
```

3. **发送确认**：如果接收到的数据包不是接收方期望的下一个数据包（即序列号不等于 `ack + 1`），接收方会发送一个包含当前 `ack` 值的ACK包，告知发送方已接收到的数据包序列号。这有助于发送方正确地管理其滑动窗口。

```
cppCopy codeelse {
    sentPacket = Packet(0, ack, 0, ACK, "");
    send();
}
```

4. **ack 在发送方的作用**：接收方通过不断更新 `ack` 并发送确认，可以指导发送方的滑动窗口移动。一旦发送方接收到新的ACK，它将**从其滑动窗口中移除所有已被确认接收的数据包**。这样，发送方的窗口始终包含了那些已发送但尚未被确认接收的数据包。

6.解决最后一次挥手问题

在实现TCP协议的过程中，一个常见的问题是确保连接的正确关闭，特别是在进行最后一次挥手时确保ACK包的可靠传输。在我的实现中，我解决了最后一次挥手时ACK可能丢包的问题，采用了一种控制重试次数的方法来避免无限重传。

在 `sendAndReceive` 函数中，我引入了一个重试计数器 `tryCount`。这个计数器初始化为 `MAXTRY`，代表最大的重试次数。每次重传时，我都会减少 `tryCount` 的值。当 `tryCount` 达到零时，停止重传，这避免了无限重传的问题。

```
void sendAndReceive(uint8_t ExpectedFlags) {
    int tryCount = MAXTRY;
    while (true) {
        if(tryCount == 0) break;
        // 发送和接收逻辑
    }
}
```

7.接收方回复的ACK丢包不影响正确性

在实现的滑动窗口协议中，接收方的ACK回复丢失并不会影响整体传输的正确性，这归功于协议的设计和滑动窗口机制。**逻辑如下：**

- **累积确认的作用**：在滑动窗口协议中，每个ACK包含一个序列号，表示接收方已成功接收并期望接收的下一个数据包的序列号。例如，如果接收方发送了ACK8，即使之前的ACK3和ACK4丢失，发送方仍能够理解接收方已成功接收序列号为8及之前的所有数据包。因此，发送方可以根据ACK8移除其滑动窗口中序列号小于等于8的所有数据包。
- **丢失最后一个ACK的处理**：如果发送方没有收到对最后一个数据包的确认（例如ACK8丢失），它会在超时后重传该数据包。重传机制确保即使最后一个ACK丢失，发送方也能够最终收到确认，从而正确地移动其窗口，并完成数据传输。

8.其他实现

1.传输时间和平均吞吐率显示

```
    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::seconds>(end_time - start_time);
    cout << "time counter ended , File transfer duration:" << duration.count() << " s " << endl;
    cout << "file transfer size : " << ackLen << " B " << endl;
    if (duration.count() != 0)cout << "file transfer rate : " << (ackLen / 1024 /
duration.count()) << " k/s " << endl;
    cout << "File transfer completed." << endl;
```

最小可行版本

在实现完整版代码之前，我先实现了一个最小可行版本，下面就是接收方的全部代码，**不超过60行**，发送方有100行。

receiver.cpp > ...

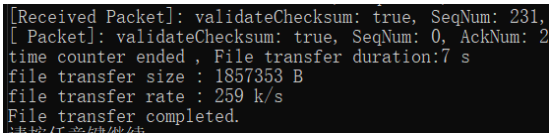
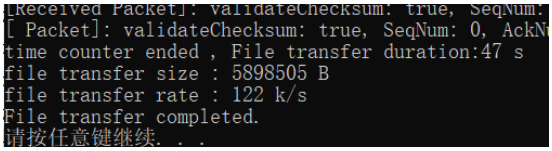
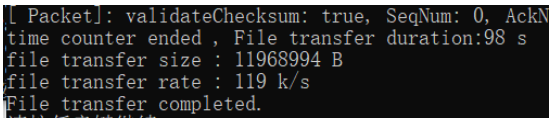
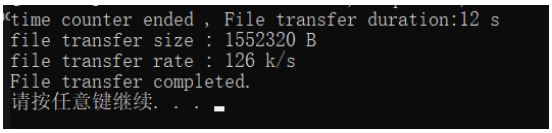
```cpp
#include "protocol.h"

WSADATA wsa;
SOCKET serverSocket;
struct sockaddr_in serverAddr, remoteAddr;
Packet sentPacket, receivedPacket;
int ack = -1, remoteAddrSize = sizeof(remoteAddr);
char fileName[256];
FILE* outFile;

void send() {
    sendto(serverSocket, (char*)&sentPacket, sizeof(Packet), 0, (struct sockaddr*)&remoteAddr, remoteAdd
}

int receive() {
    return recvfrom(serverSocket, (char*)&receivedPacket, sizeof(Packet), 0, (struct sockaddr*)&remoteAd
}

void init() {
    WSAStartup(MAKEWORD(2, 2), &wsa);
    serverSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(SERVER_PORT);
    bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
}

int main() {
    printReceiver(), init();
    cout << "Enter filename: ", cin >> fileName;
    outFile = fopen(fileName, "wb");
    do {
        receive();
    } while (receivedPacket.flags != SYN || receivedPacket.seqNum != ack + 1);
    sentPacket = Packet(0, ++ack, 0, SYN | ACK, "");
    send();
    while (true) {
        receive();
        if (receivedPacket.flags == (FIN | ACK)) {
            sentPacket = Packet(0, ++ack, 0, ACK, "");
            send();
            break;
        }
        if (receivedPacket.seqNum == ack + 1) {
            fwrite(receivedPacket.message, 1, receivedPacket.dataLen, outFile);
            sentPacket = Packet(0, ++ack, 0, ACK, "");
            send();
        }
        else {
            sentPacket = Packet(0, ack, 0, ACK, "");
            send();
        }
    }
    cout << "File transfer completed." << endl;
    fclose(outFile), system("pause");
    return 0;
}
```

忽略了打印吞吐率，快速重传等等比较细枝末节的内容。

但是实现了：GBN的差错校验，超时重传，可视化，线程，使用 mutex 实现线程安全，接收方回复的ACK丢包不影响正确性...
主要得益于把公共函数封装到protocol.h头文件里，重要函数都封装。
最小可行版本放入了**压缩包**。

实验结果--更多结果放在lab3-4了

传输时间和平均吞吐率显示

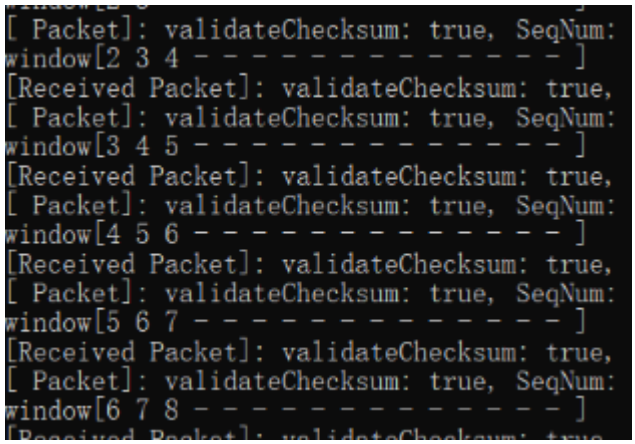
文件名	发送窗口大小	文件大小	接收时间	吞吐率	延时	截图
文件1.jpg	16	1.77MB	7s	259 k/s	1ms	
文件2.jpg	16	5.62MB	47s	122 k/s	1ms	
文件3.jpg	16	11.4MB	98s	119k/s	1ms	
文件 helloworld.txt	16	1.47MB	12s	126 k/s	1ms	

分析：

- **文件大小与传输时间关系**：
  - 文件大小较小的文件（如文件1.jpg和文件helloworld.txt）通常在较短的时间内传输完成，而较大的文件（如文件3.jpg）需要更长的时间。

我观察到的点

1. **延迟对窗口的影响**：**当网络延迟为零时**，接收方能够快速回复ACK，这导致发送方的滑动窗口（`window`）往往不会被完全填满。甚至，**窗口中可能只有一个数据包。**这表明在低延迟网络中，数据包的传输和确认可以非常迅速进行。



2. **滑动窗口大小（`N`）的设定**：在实现Go-Back-N协议时，我发现**不断增加滑动窗口的大小并不总是有益的。**由于接收方在丢失任何数据包时都不会确认窗口中该数据包之后的包，这可能**导致发送方收到大量具有重复序列号的数据包。**

实验中的坑

1.**路由器更改IP:port可能导致无法传输，重启可以解决**

实验总结

本次实验，成功实现了多个关键功能

1. **数据设计**：定义了数据包结构，包括校验和、序列号、确认号等，以支持协议的基本操作。
2. **时序设计**：实现了四次握手和四次挥手过程，确保了连接的可靠建立和终止。
3. **动态调整RTO**：根据网络条件动态调整重传超时时间，提高了数据传输的效率。
4. **差错校验**：通过校验和机制检测数据传输中的错误，保证数据的完整性。
5. **超时重传**：实现了超时重传机制，确保了数据包在丢失或延迟时能够被重新发送。
6. **滑动窗口机制**：通过滑动窗口控制发送方的数据流，优化了网络资源的使用。
7. **线程安全**：使用互斥锁保证了多线程环境中数据访问的安全性。
8. **快速重传机制**：当连续收到多个重复的ACK时触发快速重传，提高了响应速度。
9. **接收方的处理逻辑**：接收方能够正确处理按序和乱序到达的数据包，并发送相应的ACK。
10. **处理最后一次挥手的问题**：通过控制重试次数，解决了最后一次挥手时ACK丢包的问题。
11. **ACK丢包的处理**：证明了即使接收方的ACK丢失，也不会影响传输的正确性。

实验心得

- **理解逻辑本质的重要性**：**最小可行版本双方代码各自只有80行左右**，深入理解协议的逻辑本质是简化代码的关键。
- **代码封装**：将复杂的逻辑封装成函数才有了最小可行版本。
- **逐步开发的方法**：面对复杂的问题时，先实现一个最小可行版本，然后逐步扩展和优化。