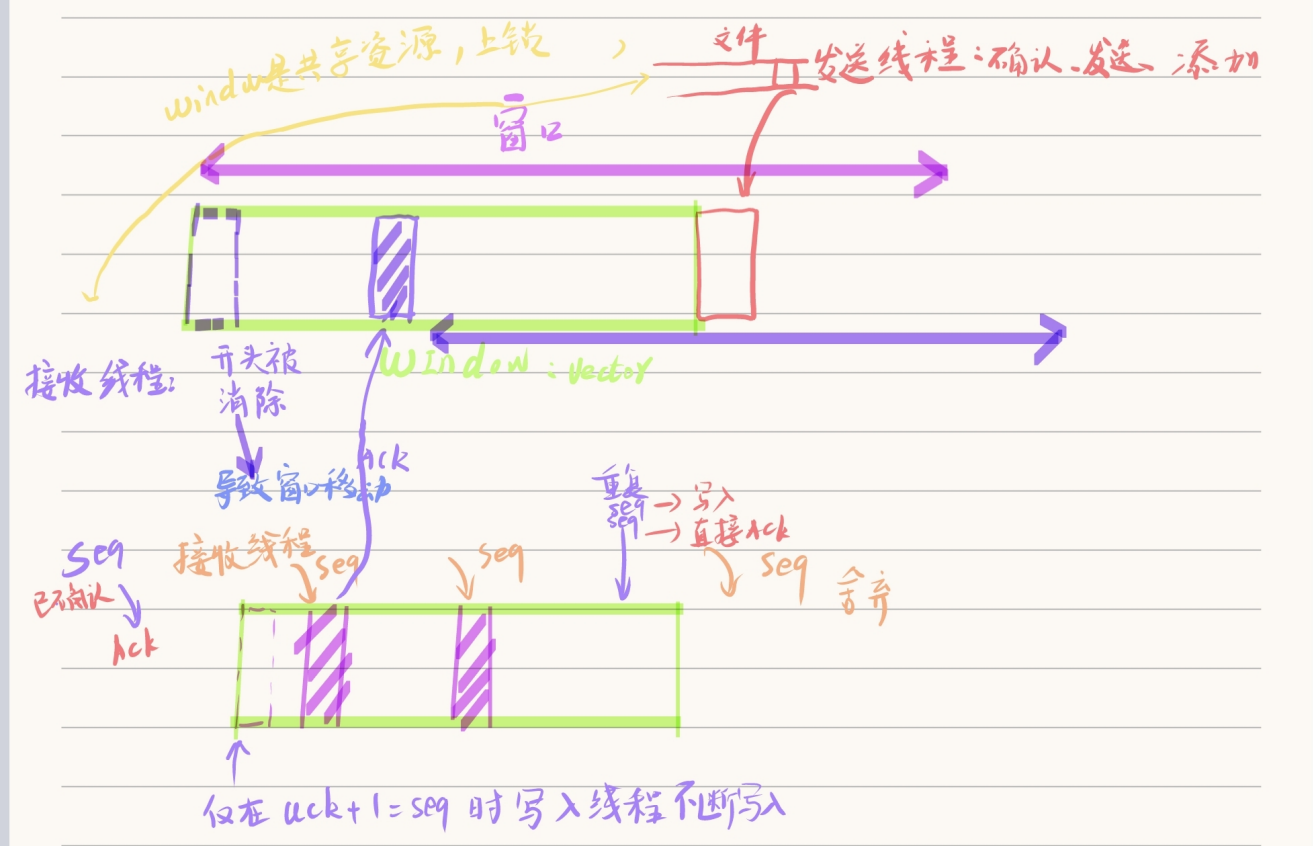
lab3-3实验报告



杜怡兴-2112847

```
CLIENT_PORT 60000
SERVER_PORT 61000

 ,-,  ,---.  ,---.  ,---.  ,---.  ,---.  ,---.  ,-,
 | |  | ,-.|  | ,-.|  | ,-.|  | ,-.|  | ,-.|  | ,-.|  | |
 | |  | | |  | | |  | | |  | | |  | | |  | | |  | |
 | |  | `-'|  | `-'|  | `-'|  | `-'|  | `-'|  | `-'|  | |
 `-'  `---'  `---'  `---'  `---'  `---'  `---'  `-'
                        V
Enter filename: 1.jpg
[Received Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 0, Flags: [SYN]
[ Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 0, Flags: [SYN,ACK]
[Received Packet]: validateChecksum: true, SeqNum: 1, AckNum: 0, DataLen: 0, Flags: [ACK]
[ Packet]: validateChecksum: true, SeqNum: 0, AckNum: 1, DataLen: 0, Flags: [ACK]
[Received Packet]: validateChecksum: true, SeqNum: 2, AckNum: 0, DataLen: 8172, Flags: [ACK]
[ Packet]: validateChecksum: true, SeqNum: 0, AckNum: 2, DataLen: 0, Flags: [ACK]
window[2 - - - - - - - - - - - - - - - - ]
ack: 1
[Received Packet]: validateChecksum: true, SeqNum: 3, AckNum: 0, DataLen: 8172, Flags: [ACK]
[ Packet]: validateChecksum: true, SeqNum: 0, AckNum: 3, DataLen: 0, Flags: [ACK]
window[3 - - - - - - - - - - - - - - - - ]
ack: 2
[Received Packet]: validateChecksum: true, SeqNum: 4, AckNum: 0, DataLen: 8172, Flags: [ACK]
[ Packet]: validateChecksum: true, SeqNum: 0, AckNum: 4, DataLen: 0, Flags: [ACK]
window[4 - - - - - - - - - - - - - - - - ]
```

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

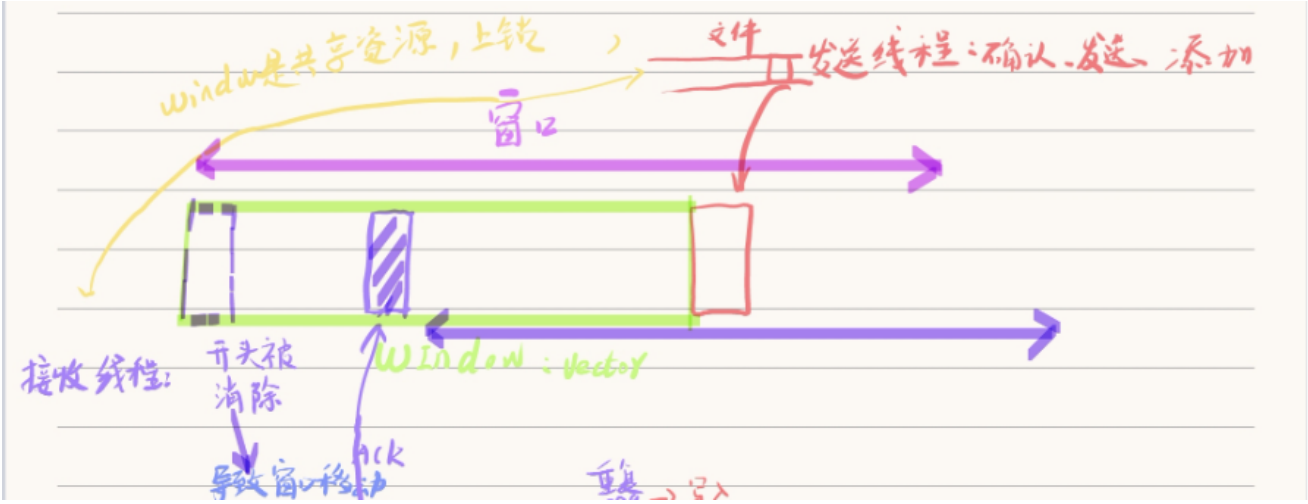
```
g++ receiver.cpp -o receiver -lws2_32
g++ sender.cpp -o sender -lws2_32
```

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。

多线程实现思路



发送方`receiverThread`逻辑

1. **收到数据包后的处理**：当接收到数据包时，`receiverThread` 遍历 `window` 并**仅移除序列号与接收到的确认号（ACK）相匹配的数据包。**这是因为在SR协议中，**接收方独立确认每个数据包**，而非像Go-Back-N协议那样使用累积确认。

```
if (receivedPacket.flags == ACK) {
    for (auto it = window.begin(); it != window.end(); ) {
        if (it->seqNum == receivedPacket.ackNum) {
            it = window.erase(it);
            break;
        } else {
            ++it;
        }
    }
}
```

2. **超时和快速重传**：如果接收线程在超时后未收到确认，或者连续收到多个相同的ACK（触发快速重传机制），它将重新发送 `window` 中的所有数据包。

```
if (recvResult < 0 || (samePacketCount > 5 && FastRetransmission)) {
    for (auto& packet : window) {
        sentPacket = packet;
        send();
    }
}
```

3. **线程安全的同步机制**：使用 `std::lock_guard<std::mutex>` 来同步对 `window` 的访问，确保在多线程环境中数据的一致性和安全性。

```
lock_guard<mutex> lock(windowMutex);
```

4. **线程退出条件**：当文件读取完毕（`feof(inFile)`）并且滑动窗口中没有剩余的数据包时（`window.size() == 0`），线程终止。这表示所有数据已被成功发送并确认。

```
if (feof(inFile) && window.size() == 0) break;
```

这些实现细节确保了在SR协议下，每个数据包都被单独确认，并且在发生数据包丢失或延迟时能够及时重传，从而提高了数据传输的可靠性和效率。

优雅地实现选择确认

在我的实验中，`set<Packet> packetsSet` 被用来优雅地实现选择确认（Selective Acknowledgment）机制。

1. **Packet类的比较运算符重载**：为了在 `set` 中自动按序列号排序，我对 `Packet` 数据包类重载了 `operator<`。这确保了 `packetsSet` 中的数据包会根据其 `seqNum` 成员变量自动排序。

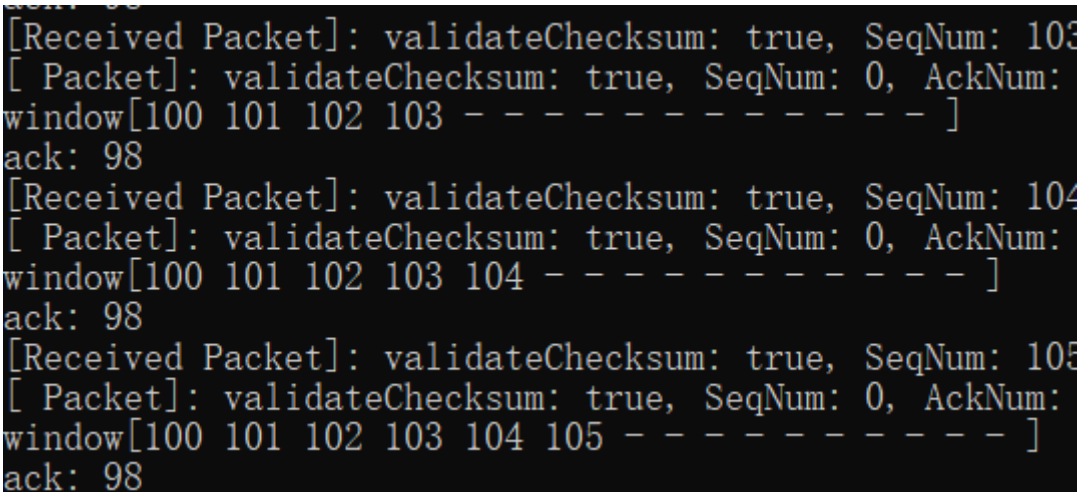
```
struct Packet {
    // 数据包结构定义
    bool operator<(const Packet& other) const {
        return seqNum < other.seqNum;
    }
};
```

2. **自然排序的有序集合**：当数据包加入 `packetsSet` 后，它们会根据序列号自然排序。这意味着**即使数据包乱序到达，它们仍会在 `packetsSet` 中被正确排序。**
3. **处理乱序数据包**：当接收方的**滑动窗口头部的数据包（即下一个期望的数据包）没有到来时**，`processPacketsThread` **不会处理** `packetsSet`，因为此时无法保证数据的顺序性。这保证了即使在乱序到达的情况下，数据的顺序性不会被破坏。
4. **连续数据包的处理**：当接收方滑动窗口**头部的数据包到来时**（即 `packetsSet` 中的最小序列号等于 `ack + 1`），`processPacketsThread` 会**从头部开始连续消去数据包，直至遇到不连续的序列号。**这样，即使数据包乱序到达，也能保证按正确顺序写入文件。

```
void processPacketsThread() {
    while (!isFinished || !packetsSet.empty()) {
        if (!packetsSet.empty() && packetsSet.begin()->seqNum == ack + 1) {
            // 数据包写入文件并更新ack
        }
    }
}
```

这种实现方式不仅保证了接收方可以处理乱序到达的数据包，同时也确保了数据包能够按正确的顺序写入文件。此外，它减少了数据包处理的复杂度，使得整个系统的效率和可靠性得到了提高。

接收方滑动窗口的可视化



1. **输出滑动窗口内所有序列号**：通过遍历 `window` 集合，打印出窗口中所有数据包的序列号（`seqNum`）。

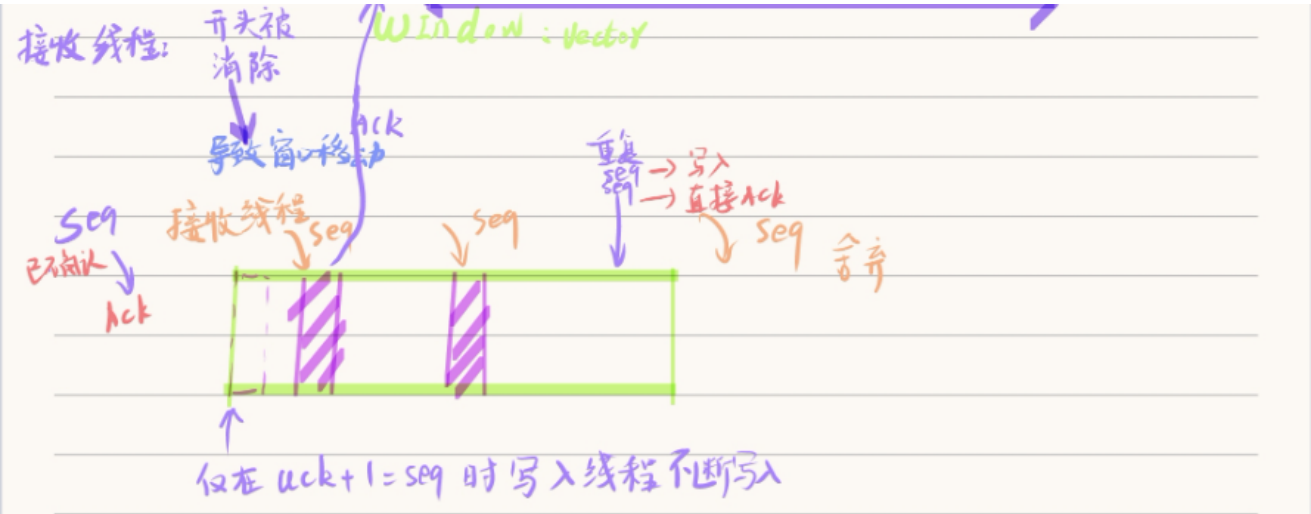
```
for (auto& packet : window) {
    cout << packet.seqNum << " ";
}
```

2. **表示剩余窗口大小**：为了表示滑动窗口的剩余空间，我输出一系列的 - 符号，数量等于滑动窗口的最大大小 `M` 减去当前窗口中的数据包数量。这样，我不仅可以看到窗口中当前的数据包，还能一目了然地了解还有多少空间可以用于接收新的数据包。

```
for (int i = 0; i < M - window.size(); i++) {
    cout << "- ";
}
```

接收方receivePacketsThread逻辑

在实验中，`receivePacketsThread` 函数的逻辑是关键的一环，特别是在实现选择重传（SR）协议的环境中。



1. **维护两个集合**：
   - `set<Packet> packetsSet`：用于**有序存储收到的数据包**。这个集合按数据包的序列号排序，**待processPacketsThread线程读取和处理。**
   - `set<int> ackedPacketsSet`：用于**跟踪已经确认过的数据包的序列号，防止对同一个数据包进行重复处理*。**
2. **处理终止信号**：当接收到标志为 `(FIN | ACK)` 的数据包时，表示数据传输即将结束。此时，设置 `isFinished` 标志为 `true` 并退出循环，以结束线程的执行。

```
if (receivedPacket.flags == (FIN | ACK)) {
    isFinished = true;
    break;
}
```

3. **避免重复确认**：如果收到的数据包的序列号已存在于 `ackedPacketsSet` 中，表明该数据包之前已经被确认过。在这种情况下，**只需再次发送ACK响应，而不需要再次处理该数据包。**

```
if (ackedPacketsSet.find(receivedPacket.seqNum) != ackedPacketsSet.end()) {
    sentPacket = Packet(0, receivedPacket.seqNum, 0, ACK, "");
    send();
}
```

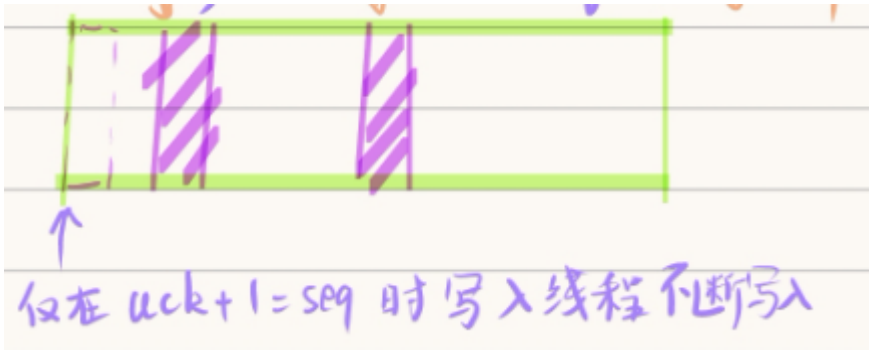
4. **处理接收窗口内的数据包**：对于那些序列号在接收方**当前确认号ack 和 `ack + M`（滑动窗口大小）之间**的数据包，将它们添加到 `packetsSet` 和 `ackedPacketsSet`，并发送ACK响应。这确保了按序和乱序到达的数据包都能被妥善处理。

```
else if (receivedPacket.seqNum > ack && receivedPacket.seqNum <= ack + M) {
    packetsSet.insert(receivedPacket), ackedPacketsSet.insert(receivedPacket.seqNum);
    sentPacket = Packet(0, receivedPacket.seqNum, 0, ACK, "");
    send();
}
```

这些实现细节确保了在SR协议下，每个数据包都被独立处理，并且在发生数据包丢失或延迟时能够及时重传，从而提高了数据传输的可靠性和效率。

processPacketsThread的逻辑

在选择重传（SR）协议的实现中，`processPacketsThread` 函数扮演着核心角色，负责处理接收到的数据包。



线程持续运行条件：只要 `isFinished` 标志为 `false` 或 `packetsSet` 非空，**这意味着还有数据包待处理或还在接收数据包**，线程就会继续运行。这保证了只要数据传输过程未结束，线程就不会停止。

1.
```
while (!isFinished || !packetsSet.empty()) {
    // 线程处理逻辑
}
```

2. **处理接收窗口开头的数据包**：如果 `packetsSet` 不为空，并且集合中的第一个数据包（即**序列号最小的数据包）的序列号正好是** `ack + 1`，此时，线程将该数据包的内容写入文件，并更新 `ack`，packetsSet.erase(packetsSet.begin())相当于**移动窗口。**

```
if (!packetsSet.empty() && packetsSet.begin()->seqNum == ack + 1) {
    const Packet& packet = *packetsSet.begin();
    fwrite(packet.message, 1, packet.dataLen, outFile);
    ack++;
    packetsSet.erase(packetsSet.begin());
}
```

3. **锁机制**：为了保证对共享资源 `packetsSet` 的线程安全访问，使用了 `std::lock_guard<std::mutex>` 锁机制。这确保了在处理数据包时，其他线程不会同时修改 `packetsSet`。

```
lock_guard<mutex> lock(packetsSetMutex);
```

4. **与 `receivePacketsThread` 线程的协作**：`processPacketsThread` 通过检查 `isFinished` 标志与 `receivePacketsThread` 线程协作。当 `receivePacketsThread` 设置 `isFinished` 为 `true`，表明数据传输结束，`processPacketsThread` 也将随之结束。

这样的实现确保了接收方能够按序处理每个数据包，并且在收到数据包的同时能够有效地管理和更新接收窗口。

对于可能的错误都能正确处理

在实验中，接收方的实现考虑到了多种可能的错误情况，并采取了相应的措施来确保数据传输的正确性和可靠性：

1. **处理重复数据包**：通过维护一个已确认数据包的序列号集合 `ackedPacketsSet`，接收方能够有效地识别并处理重复接收的数据包。**即使接收方连续收到两个窗口内的同一数据包，也不会导致数据被重复写入文件。**这是因为 `ackedPacketsSet.find(receivedPacket.seqNum) != ackedPacketsSet.end()` 检查确保**只处理尚未确认的新数据包。**通过 `ackedPacketsSet.insert(receivedPacket.seqNum)` 的操作，接收方实现了对数据包的去重效果。

```
if (ackedPacketsSet.find(receivedPacket.seqNum) != ackedPacketsSet.end()) {
    sentPacket = Packet(0, receivedPacket.seqNum, 0, ACK, "");
    send();
}
```

2. **处理接收方ACK丢失情况**：即使**接收方发送的ACK丢失**，发送方可能会**重发已经被接收方确认的数据包。**但**由于接收方知道这些数据包已经被确认过，它不会重复处理它们。**这种机制确保了即使在ACK丢包的情况下，接收方也能正确地维护数据流的状态，不会导致数据重复或乱序。

```
else if (receivedPacket.seqNum > ack && receivedPacket.seqNum <= ack + M) {
    packetsSet.insert(receivedPacket), ackedPacketsSet.insert(receivedPacket.seqNum);
}
```

这些措施确保了即使在网络不稳定或出现错误的情况下，接收方仍能正确地处理数据包，保证数据传输的顺序性和完整性。

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
    }
}
```

其他实现

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;
```

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;
```

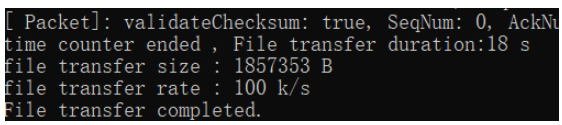
快速重传机制

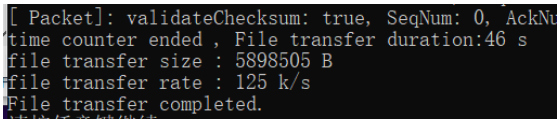
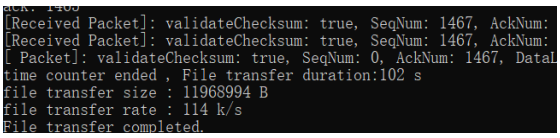
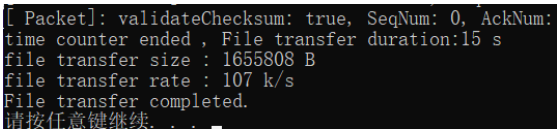
通过跟踪最后一个接收到的ACK号（`lastPacketAck`）和相同ACK号的计数（`samePacketCount`），我实现了快速重传机制。当连续收到超过一定数量的相同ACK时，触发快速重传。

```
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
}
```

实验结果

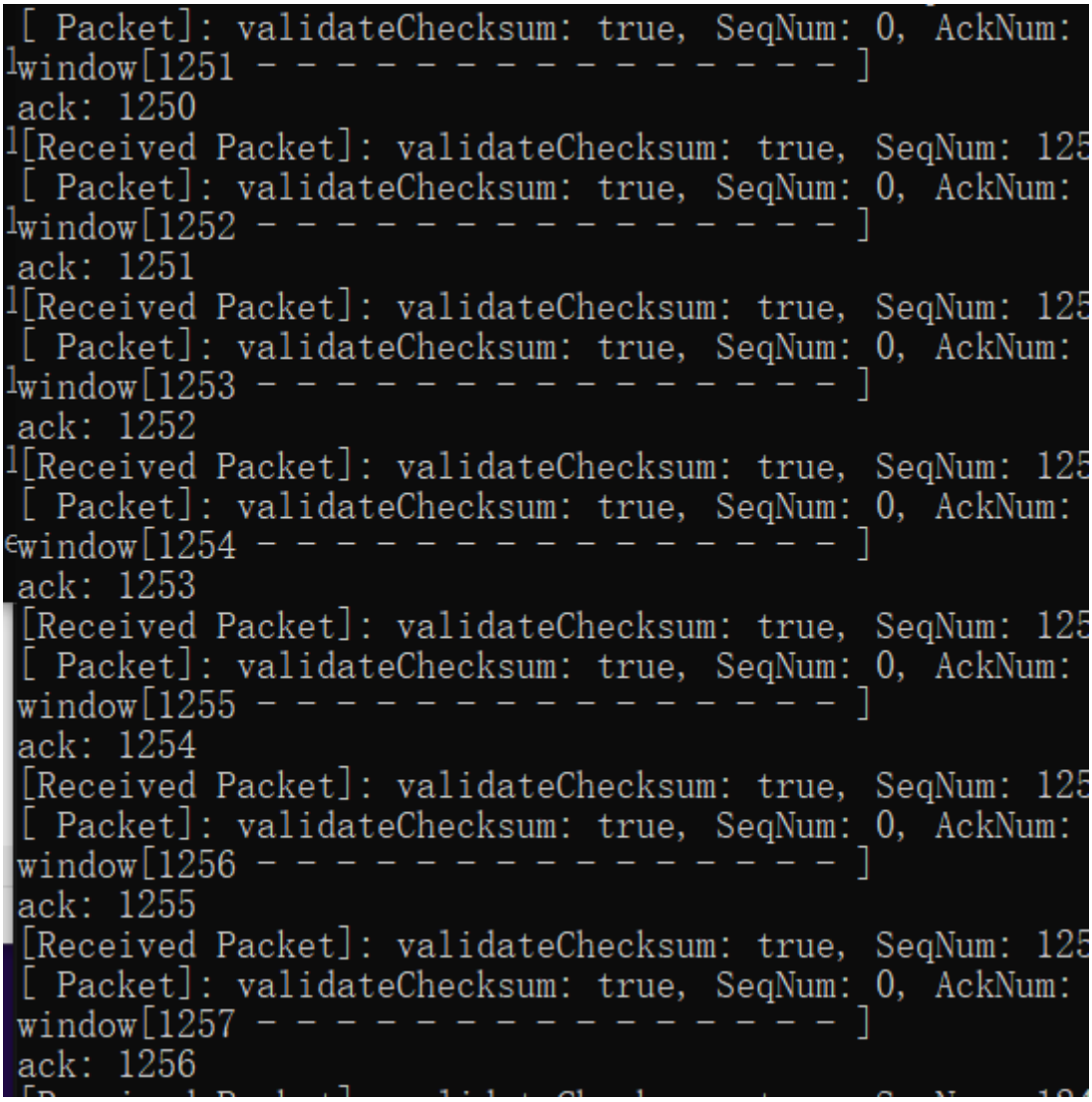
传输时间和平均吞吐率显示

文件名	N	M	文件大小	接收时间	吞吐率	延时	截图
文件1.jpg	32	16	1.77MB	18s	100 k/s	1ms	

文件名	N	M	文件大小	接收时间	吞吐率	延时	截图
文件2.jpg	32	16	5.62MB	46s	125 k/s	1ms	
文件3.jpg	32	16	11.4MB	102s	114k/s	1ms	
文件 helloworld.txt	32	16	1.47MB	15s	107 k/s	1ms	

在我的实验中，窗口大小对于数据传输的性能有影响。

1. **接收窗口与发送窗口的不同行为**：在大多数情况下，**接收方的滑动窗口并未完全填满**，而**发送方的窗口则可能被填满。**这主要是因为接收窗口中的数据包**一旦到达就可以立即被写入文件**，而**发送窗口中的数据包必须等到收到对应的ACK才能从窗口中移除。**这导致接收窗口能够更快地处理和释放数据包，而发送窗口则可能因等待ACK而持续保持满状态。



2. **延时为零时窗口的表现**：当网络延时为零时，发送窗口和接收窗口通常都不会完全填满。这是因为在**无延迟的网络环境中**，ACK可以非常迅速地回传给发送方，使得**发送方能够快速地移动其滑动窗口并发送新的数据包。**同样，接收方也能够快速接收和处理数据包，从而保持窗口的流动性。

3. **窗口未满时的发送速度**：**当双方的窗口都没有填满时，数据的发送速度通常较快。**因为发送方不需要频繁等待ACK来移动其窗口，而接收方也能够持续接收新的数据包并快速处理。

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

1. **数据设计**：设计了数据包结构，包括必要的信息如校验和、序列号、确认号等，确保数据传输的完整性和可靠性。
2. **时序设计**：实现了四次握手和四次挥手过程，确保了TCP连接的可靠建立和终止。
3. **动态调整RTO**：根据网络状况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**：实现了差错校验机制，确保数据在传输过程中的正确性。
5. **超时重传**：在数据包丢失或延迟的情况下实现了超时重传机制，保障了数据传输的可靠性。
6. **优雅地实现选择确认**：利用set的自然排序特性，实现了SR协议中的选择确认机制。
7. **处理可能的错误**：有效应对了数据重复和ACK丢失的情况，保证了数据传输的正确性。
8. **解决最后一次挥手问题**：确保了连接终止时的正确性，避免了可能的挂起状态。
9. **传输时间和平均吞吐率显示**：为了评估性能，实现了传输时间和吞吐率的计算与显示。
10. **RTO动态调整**：根据ACK的接收情况动态调整重传超时时间，提高了协议的效率。
11. **快速重传机制**：在连续收到多个重复ACK的情况下触发快速重传，提高了响应速度。

实验心得

通过这次实验，我深刻体会到了对**数据结构和C++语言**熟练掌握的重要性。例如，通过自定义set的排序功能，我能够更简单、更直观地实现了SR协议的选择确认机制，这大大简化了实验的实现过程。