

**计算机网络实验报告**

**实验3-3**

****

专 业 信息安全

学 号 2113662

姓 名 张丛

班 级 信安一班

1. **实验目的**

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，发送窗口和接收窗口采用相同大小，支持选择确认，完成给定测试文件的传输。

1. **实验要求**

➢ 协议设计：数据包格式，发送端和接收端交互，详细完整

➢ 流水线协议：多个序列号

➢ 选择确认：SR

➢ 日志输出：收到/发送数据包的序号、ACK、校验和等，发送端和接收端的窗口大小等情况，传输时间与吞吐率。

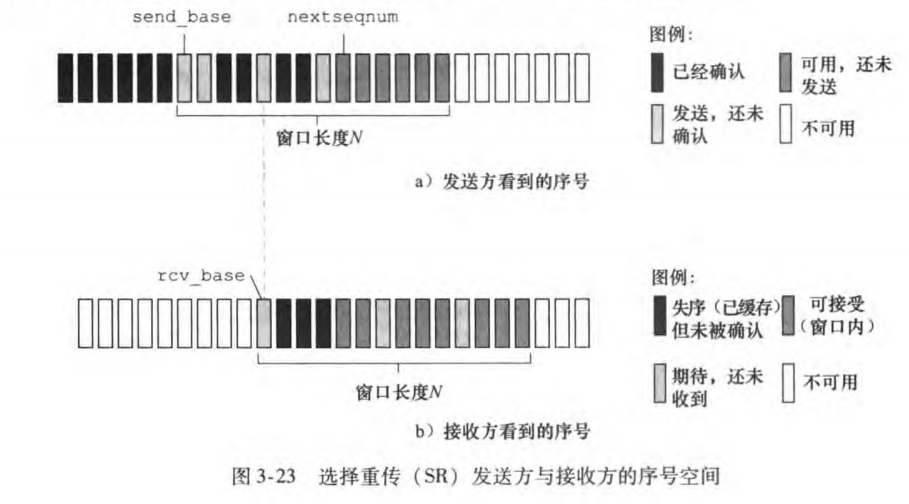
➢ 测试文件：1.jpg、2.jpg、3.jpg、helloworld.txt

1. **实验原理**

##### 累积确认和选择确认的区别：

累积确认：对序号为n的分组的确认采取累积确认的方式，表明接收方已正确接收到序号为n的以前且包括n在内的所有分组。

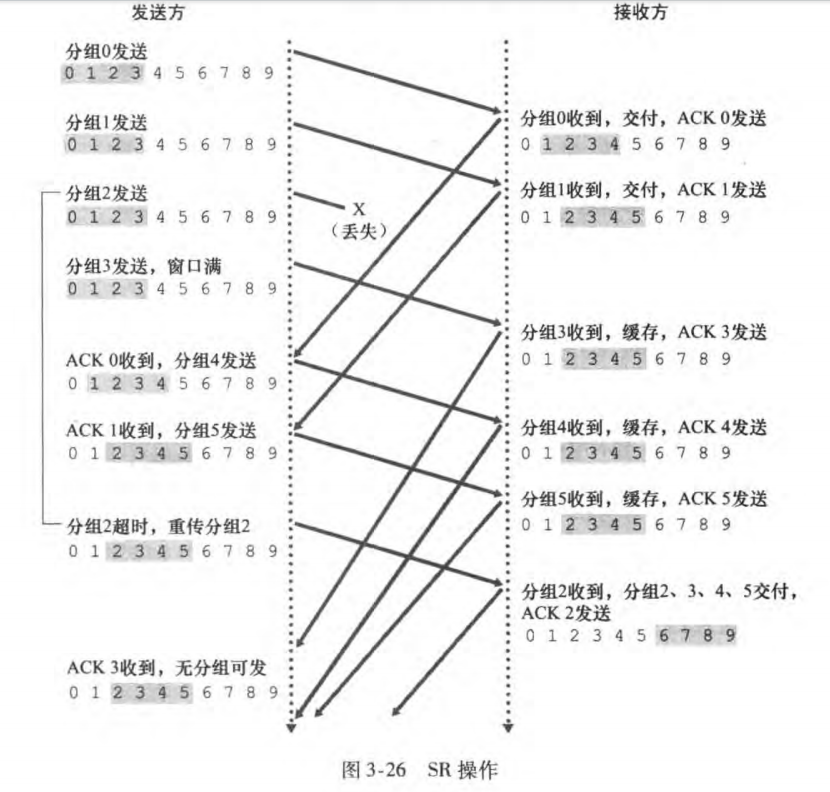
选择确认：允许接收方确认成功接收的多个、不连续的字节范围。



##### 选择重传：

发送方仅重传那些它怀疑在接收方出错（即丢失或受损）的分组而避免了不必要的重传。

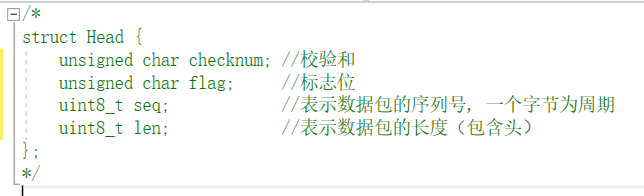
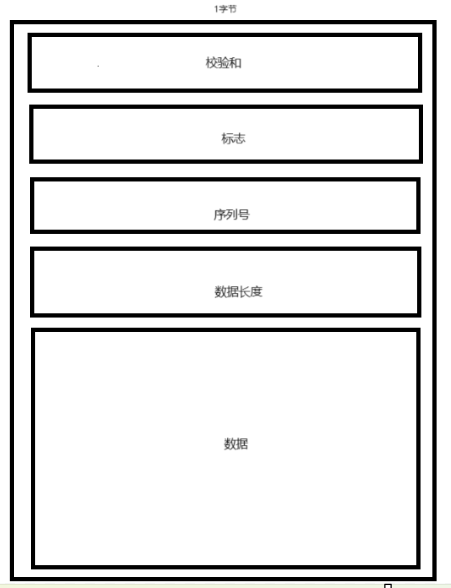
接收方将确认一个正确接收的分组而不管其是否按序，失序的分组将被缓存直到所有丢失分组（即序号更小的分组）皆被收到为止，这时才可以将一批分组按序交付给上层。



1. **实验内容**

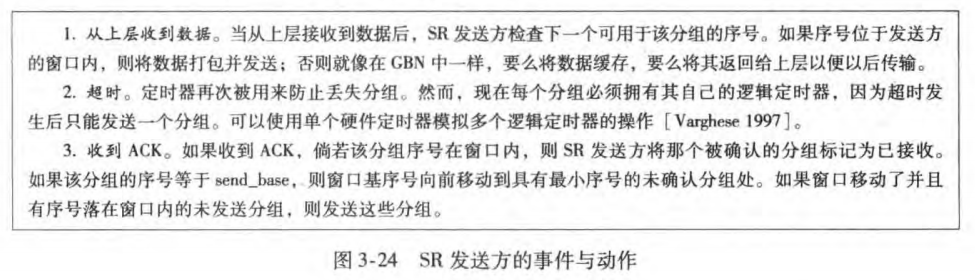
在实验3-1中的重复内容（如三次握手、两次挥手、差错检测等等）不再赘述。

### 协议设计

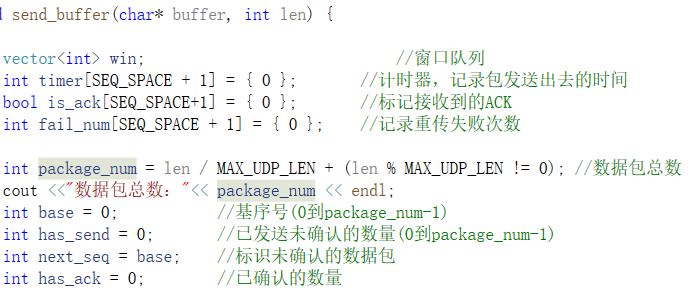


在实验3-2中，我们已经实现了停等机制改成基于滑动窗口的流量控制机制，在本次实验3-3中，滑动窗口的基本思路没有太大变化，但需要根据选择重传来调整。

### 发送端(Client)：

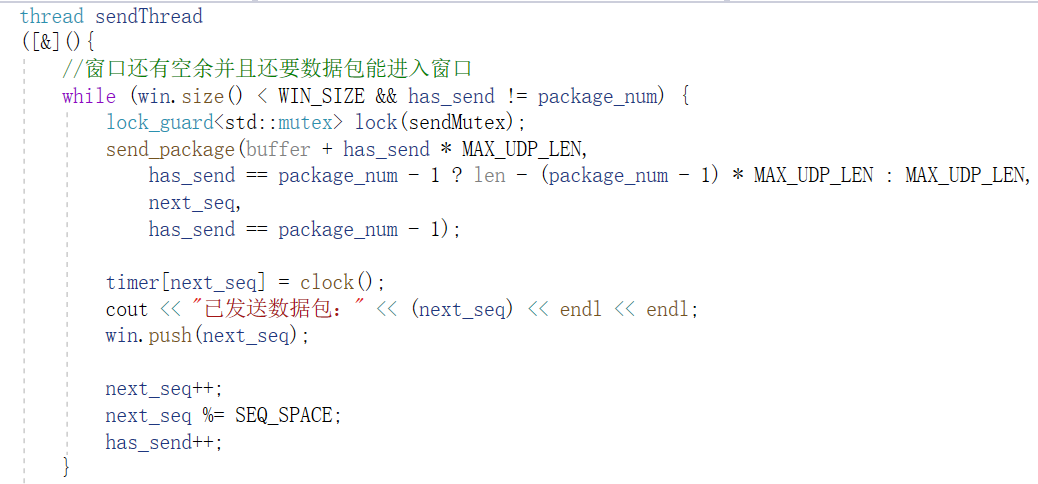


在3-3中，设置**窗口**如下：



相较于实验3-2，主要多了is\_ack[]用于**标记**数据包。

然后创建发送数据包的**线程**：



win.size()来控制窗口长度，has\_ack来判断是否还有未发送的包，

send\_package来发送数据包，timer[]记录时间。

在发送端（Client）的发送线程（sendThread）中，还设置了**超时重传**：



迭代窗口中的数据包，检查未标记的数据包是否超时，对所有的未标记的超时数据包进行重传。

使用vector<int>::iterator it 进行迭代。

重传会刷新计时器，但不会改变数据包在窗口的顺序。

在发送线程（sendThread）之外，进行接收ACK并处理。即多线程同时收发。

1. char recv[3];
2. if (recvfrom(m\_ClientSocket, recv, 3, 0, (sockaddr\*)&m\_ServerAddress, &len\_addr) != SOCKET\_ERROR && check\_sum(recv, 3) == 0 && recv[1] == ACK)
3. {
4. SUCCESS\_RECV++;
5. //大序ACK
6. if ((win.front() > SEQ\_SPACE - WIN\_SIZE) && ((int)recv[2] < WIN\_SIZE)) {
7. is\_ack[(int)recv[2]] = 1;
8. }
9. if ((win.front() < (int)recv[2])) {
10. if ((win.front() < WIN\_SIZE) && ((int)recv[2] > SEQ\_SPACE - WIN\_SIZE))
11. {
12. break;
13. }
14. is\_ack[(int)recv[2]] = 1;
15. }
16. cout << "接收到ACK：" << (int)recv[2] << endl;
17. cout << "窗口队列头的序号：" << win.front() << endl;
18. cout << "窗口队列的大小：" << win.size() << endl << endl;
19. //接收到期望ACK
20. if (win.front() == (int)recv[2]) {
21. is\_ack[(int)recv[2]] = 1;
22. //更新窗口
23. int index = (int)recv[2];
24. while (is\_ack[index])
25. {
26. is\_ack[index] = 0;
27. base++;
28. has\_ack++;
29. win.pop();
30. index++;
31. index %= SEQ\_SPACE;
32. }
33. }
34. //对于ACK<win.front()的情况不用理会
35. }

分三种处理，接收到大序、小序和期望的ACK。

（期望ACK就是窗口头的ACK，大序和小序基于期望ACK来判断。）

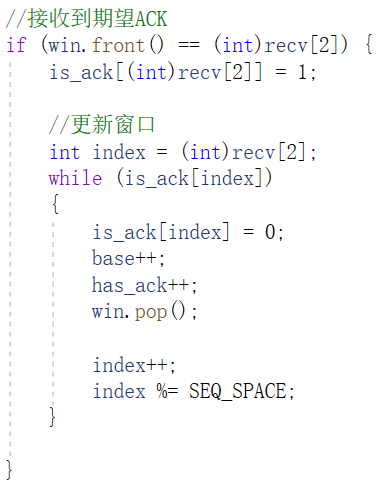
对于**大序ACK**:



就是将此数据包**标记**为已经正确接收。

被标记后，数据包将不会进入超时重传的检测。

对于**期望ACK**:



就是在一个while里面进行迭代，移动窗口，移动到具有最小序号的未确认分组处。

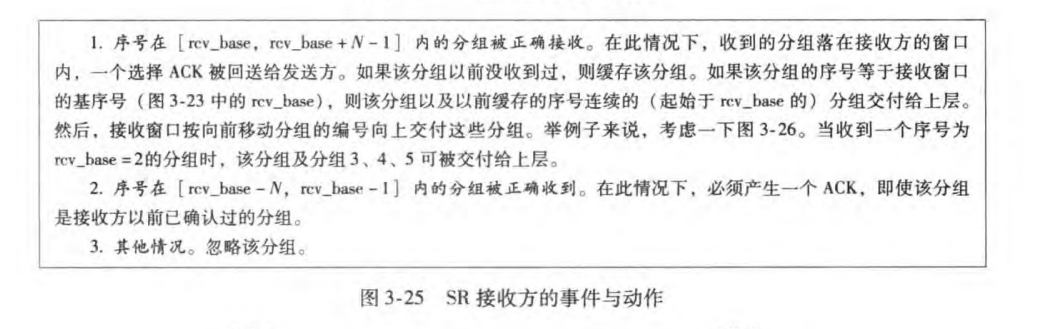
index用于迭代分组序号（数据包序号）。

窗口移动时，还需要恢复is\_ack[]状态（即置为0）以便下一个序号空间使用。

此时窗口长度减小，在sendThread线程内会进行判断并发送新数据包。

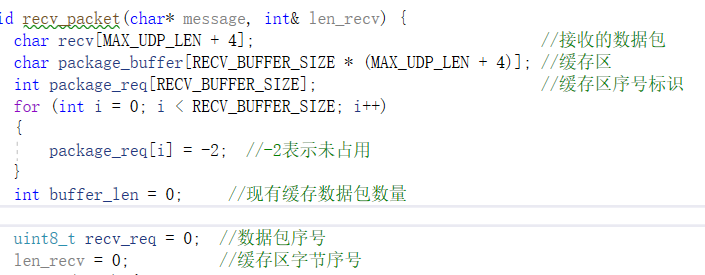
对于**小序ACK，**就是什么都不用做（因为已经确认了，还不在窗口内）。

### 接收端（Server）:



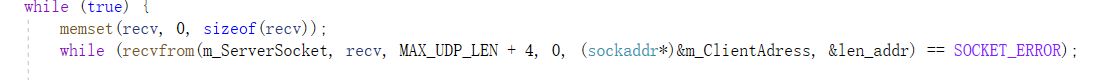
此次实验，接收端的难点在于对缓存区的处理。

设立**缓存区**：



package\_buffer用于缓存数据包，package\_req用于标识缓存的数据包序号和位置。

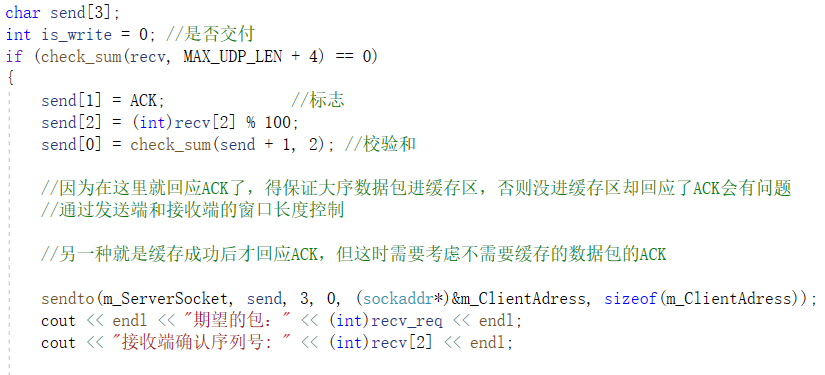
首先还是不断接收数据包：



处理数据包：

首先，对应接收到的数据包，不管序号大小，都**回应ACK**。

即便是，对于已经确认过的，甚至不在窗口内的，也需要回应（即base-N到base-1的数据包），因为要考虑之前的回应ACK丢失的问题：



这里有一个问题，就是，你既然回应了ACK，对于你接收端需要缓存的数据包，你接收端就必须缓存。这需要控制窗口长度和缓存区大小，分别在发送端和接收端实现。

还有一种方法，就是，对于需要缓存的数据包，接收端缓存后再发送ACK,这样不必过分担心缓存区溢出的问题，但需要对“不需要缓存的数据包”进行额外处理。

若接收到的是，**没收到过的数据包**，需要缓存：

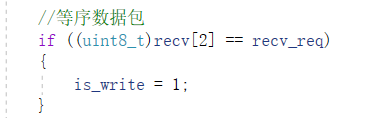


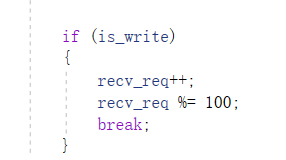
流程是：

* 判断是否是大序（考虑序号空间）
* 是否缓存过
* 缓存区满了没
* 没满就要将数据包存进package\_buffer,将序号存进package\_req。

若收到过的**小序数据包**，已经回应了ACK，无需额外处理。

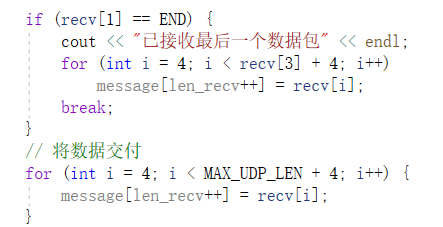
若接收端**期望的数据包**：



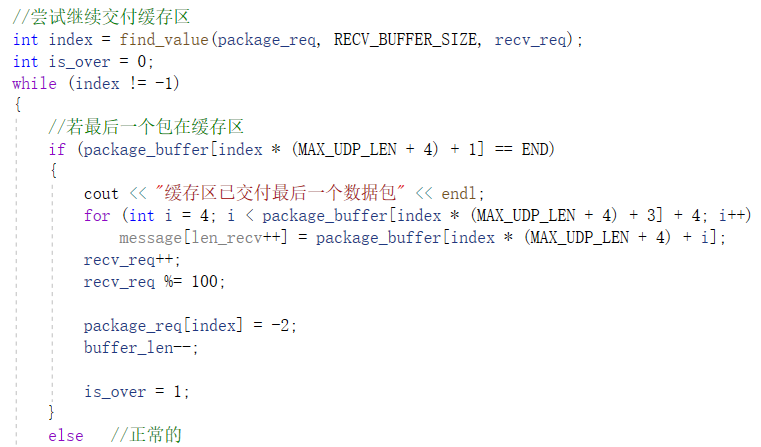


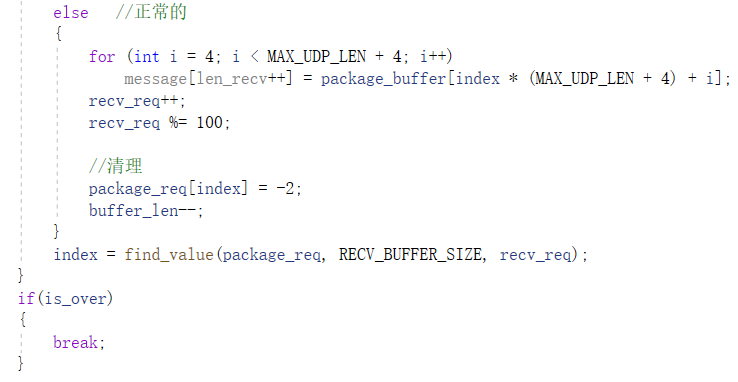
不需要缓存，直接进行交付。

**交付**数据如下：



并且需要**交付，连续**的缓存过的分组（如果有）：

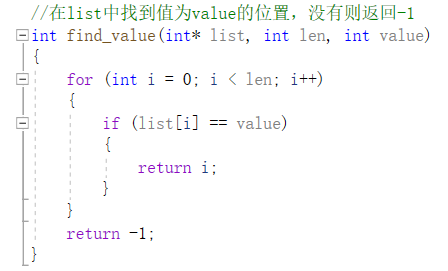




因为最后一个包也可能在缓存区，所以进行了判断。

缓存区交付后，package\_req[]置为初始状态-2表示没有占用。

其中find\_value的作用是在数组中找到value的位置，找不到则返回-1.



以上就是发送端和接收端相对于实验3-2的主要更改。

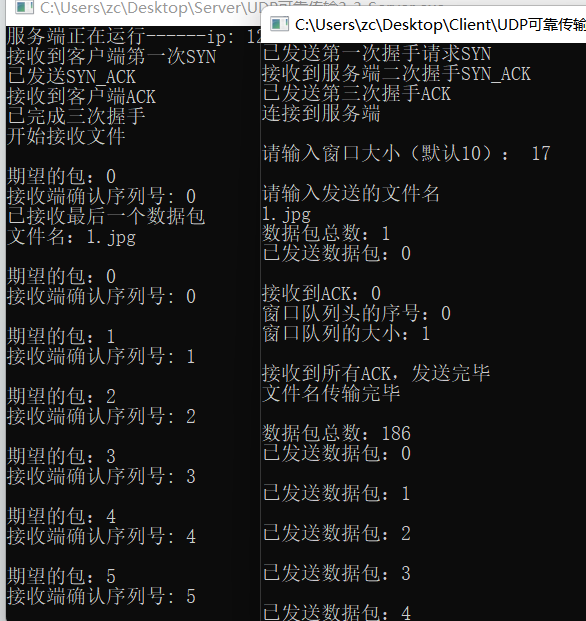
**实验效果**

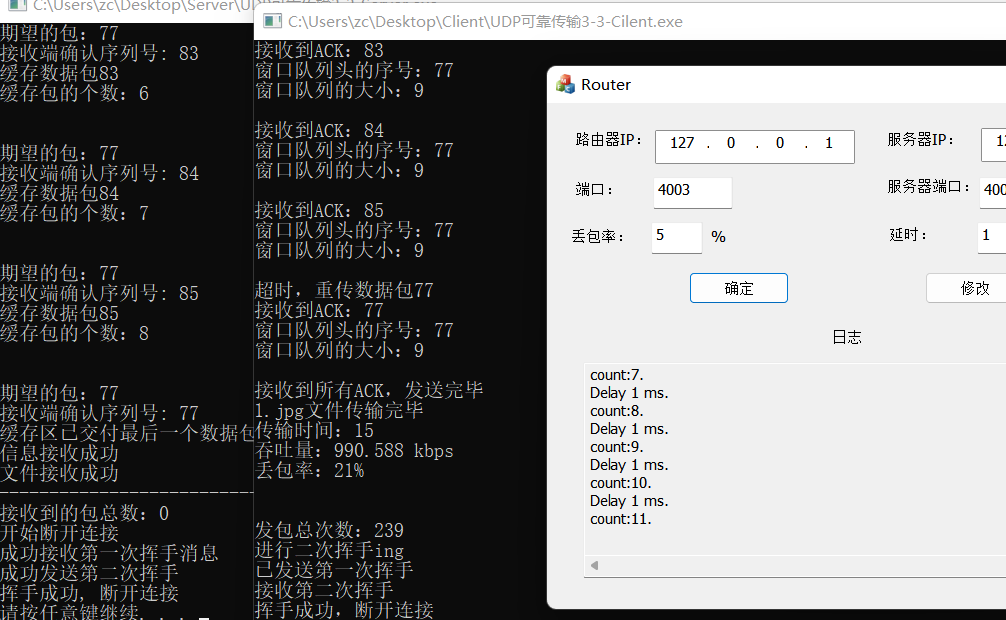
在路由器设置丢包、延时后

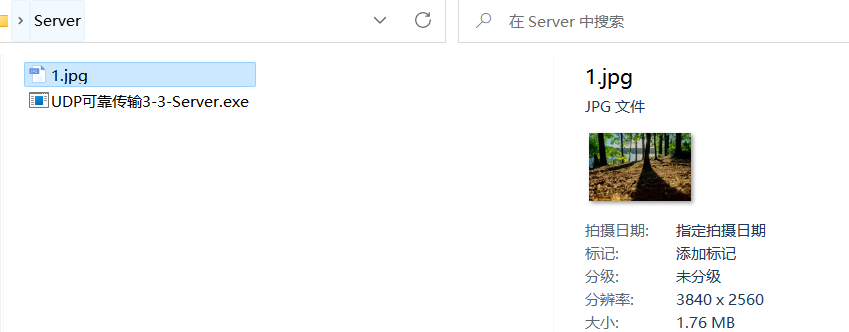


对于四个测试文件：

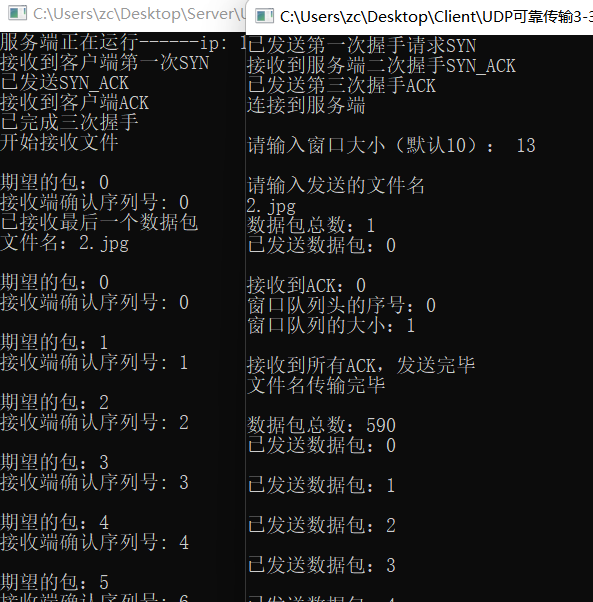
1. jpg:

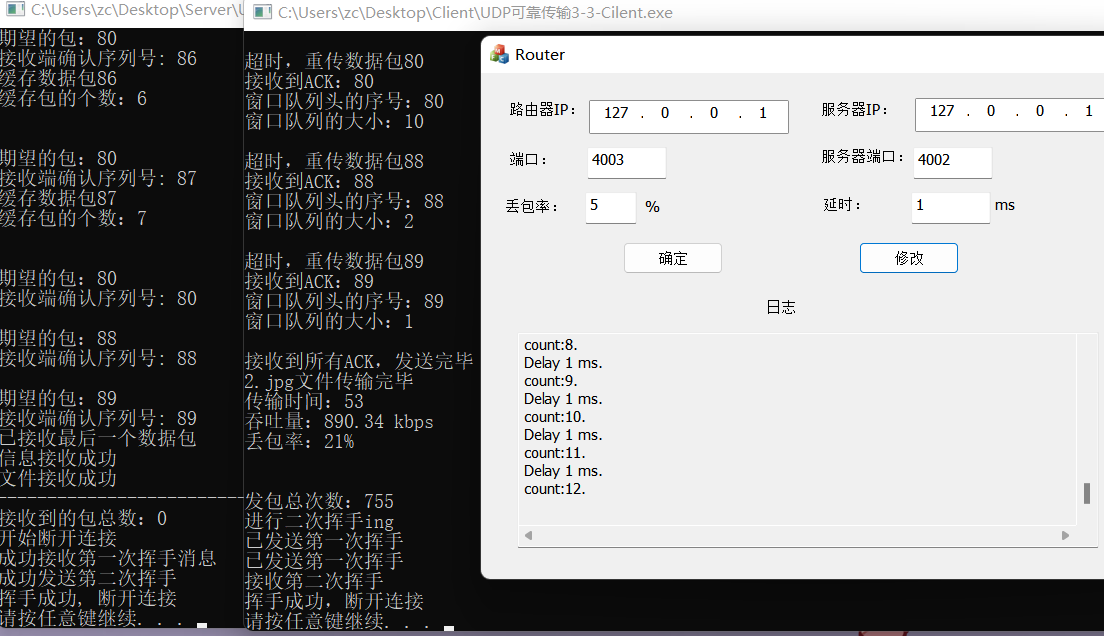


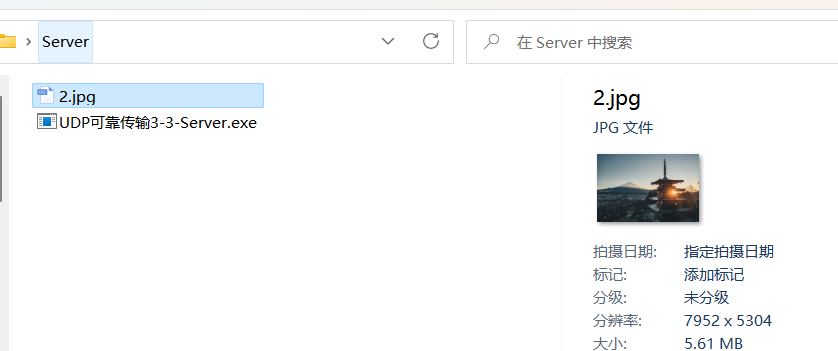




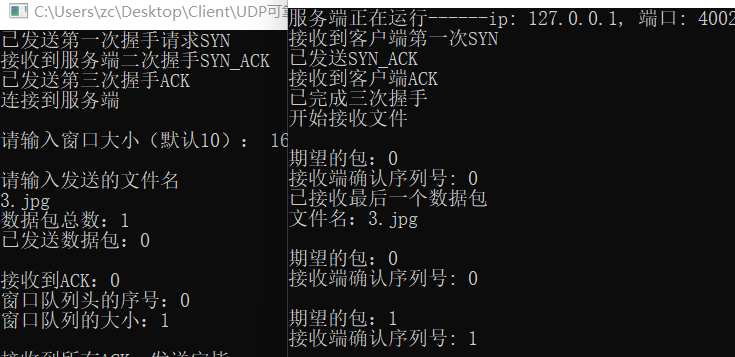
1. jpg:



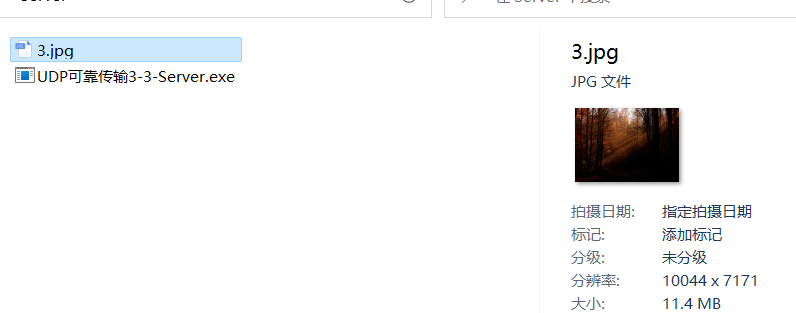




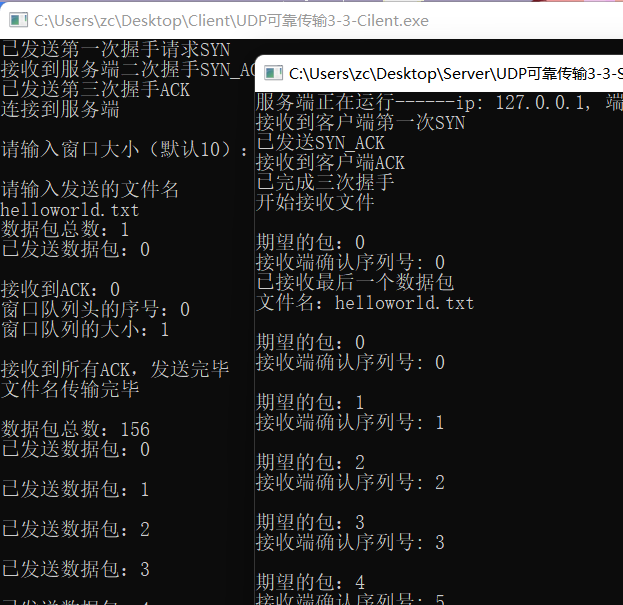
1. jpg:

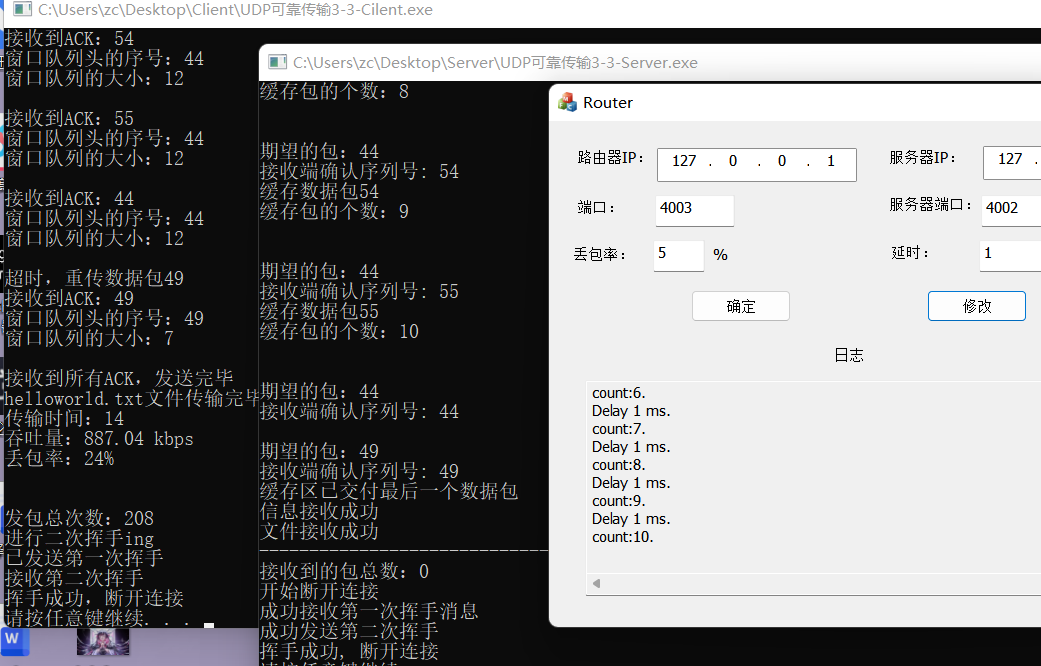


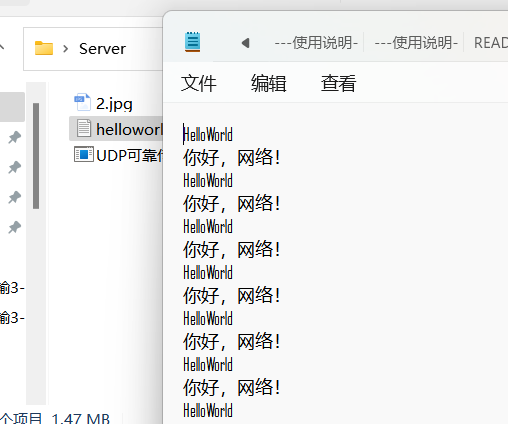




1. helloworld.txt:







1. **思考与总结**

实现了UDP可靠传输的滑动窗口机制和选择确认机制，复习到课上的理论知识点。

中途有不懂的地方和概念模糊的地方，去翻书找到了。

使用了多线程。