lab3-3实验报告

姓名：吴静
学号：2113285
专业：信息安全

一、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。

- 选择确认：针对丢失的某一分组进行重传

二、实验原理

1. 选择确认

由上一次实验可知，累积确认的弊端是在某个数据包丢包后，该数据包后所有已经发送的数据包都要重新进行发送，这会导致批量重传，传输效率不高，因此引入了选择确认。

选择确认`SACK`：

- 接收端：对每一个正确到达的包都发送确认，对位于当前接收窗口内的数据包设置缓存，缓存乱序到达的数据包
- 发送端：只重传没有接收到确认的包，即丢失或者出差错的数据包，而收到确认的包不进行重传

接收方通过这种方式告诉发送方哪些数据包已经正确接收，来成功使发送方只发送没有正确到达的数据包。

2. 滑动窗口

上一次实验中已经使用到了滑动窗口中的回退N帧协议：

- 当窗口中的一个数据包没有正常发送，窗口内的N个数据包都要重新进行发送

本次实验用到的是选择重传协议`SR`：

- 当发送方收到数据包的`ack`确认包，如果该帧在窗口内，则SR发送方将被确认的数据包标记为已接受
- 窗口左边界向前移动到未确认数据包的最小序号处，即越过所有已确认的包

在每次发送数据包时，窗口右边界向右移动一位。

3. 超时事件

回退N帧协议：

- 一整个窗口设置一个定时器，每次发送窗口内的第一个包的时候设置定时器的值，当定时器超时了，则进行窗口内所有包的重新发送。

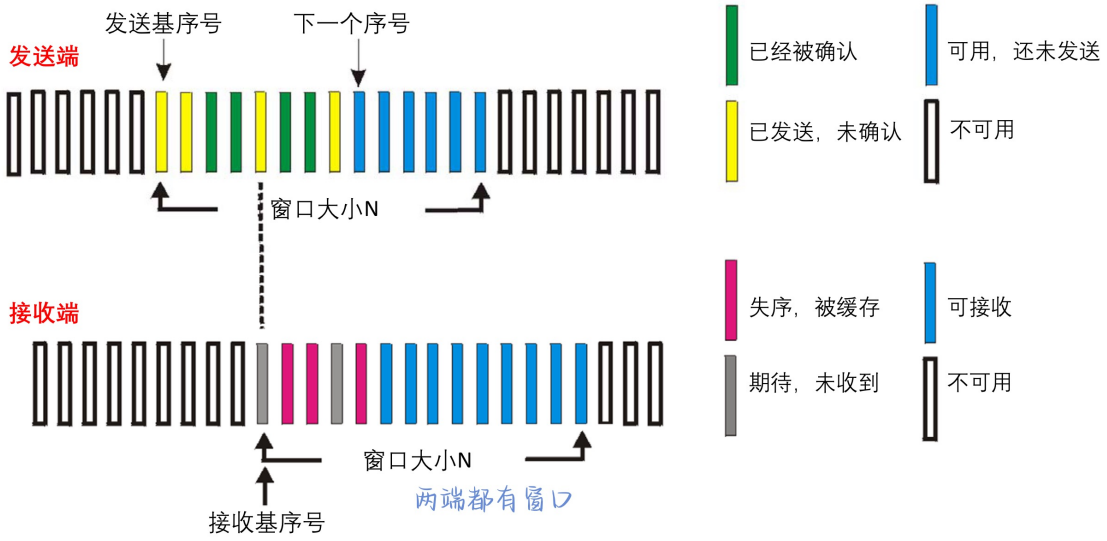
选择重传协议：

- **为每一个包设置一个定时器**，当该包超时了且没有被确认，则**单独对该数据包进行重新发送。**

4. 选择重传协议

（1）窗口

- 选择重传：*Selective Repeat (SR)*
  - 接收端独立确认每个正确接收的分组，必要时缓存分组，对高层按序交付
  - 每个分组独立定时，发送端只重传未收到ACK的分组，



如图所示，这里发送端和接收端各有四种数据包。

- 发送端：
  - 已经被确认：发送了某一数据包并且收到确认，标记该数据包为“已经被确认”
  - 已发送，未确认：发送了某一个数据包但是没有收到确认
  - 可用，还未发送：在窗口内可以进行发送，但是还没有被发送端发送出去
  - 不可用：在窗口外不可以被发送

  窗口左边界指向当前期望收到确认的数据包，并且当移动至已经被确认的数据包时会继续向右移动；

  当收到的数据包的确认号不是当前期望收到的确认号，将收到的数据包的确认号对应的数据包标记为“已经被确认”
- 接收端：
  - 失序，被缓存：收到了不是当前期望，但是序号在窗口内的数据包，进行缓存，当收到前面的数据包时，就可以将缓存内的数据包取出存储，而不是让发送端重新发送
  - 期待，未收到：当前需要存储的数据包数据，但是还没有收到确认
  - 可接受：在窗口内，还没有收到确认的数据包
  - 不可用：在窗口外，暂时不需要收到的数据包

  当收到了期望的数据包时，窗口向右移动，同时对数据包内的信息进行存储，同时将缓存中的当前需要的数据包的信息取出进行存储，窗口继续向右移动直到当前期望的数据包状态为“没有收到”；

  当收到的数据包是乱序时，进行数据包的缓存，同时将该数据包标记为“失序，被缓存”

(2) SR发送端

1. 接收上层数据

   如果发送窗口中右可用的序号，则发送该数据包分组

2. 超时重传

   为每一个帧设置一个定时器，当某一个帧定时器超时并且该帧仍然没有被确认时，重新发送该数据帧

3. 接收`ack`确认

   如果收到`ack`，加入该帧在窗口内，则将被确认的帧标记为“已接受”；如果该帧序号是窗口最左边对应的序号，则窗口左边界向前移动直到下一个未确认分组的序号

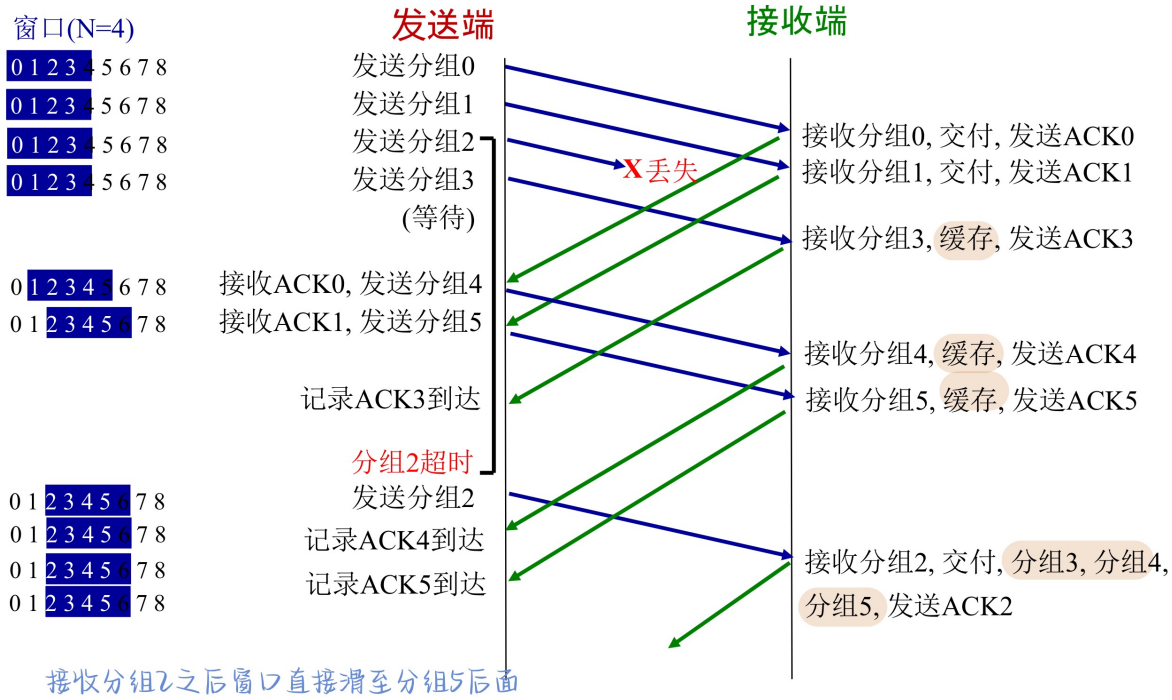
(3) SR接收端

主要工作为接收分组并缓存。

当收到一个正确接收的数据包（校验和正确）：

- 如果该数据包在当前窗口前面大小为N的窗口内，则直接发送该数据包的确认
  - 这个数据包收到的原因：由于延迟，在当前存储的过程中，发送到的确认序号没有到达发送端，发送端中该数据包对应的序号就已经超时进行重传了，所以接收端可能会重复接受已经进行缓存的数据，且该数据位于接收窗口前面
- 如果该数据包在当前窗口内，发送该数据包的确认
  - 失序分组进行缓存
  - 按序到达的分组进行存储（交付给上层），窗口向前滑动

(4) 交互



数据交互部分，为每一个包设置定时器，当该数据包对应的定时器超时，则对该数据包进行重传；同时对每一个收到确认的数据包进行标记，以便不再重复进行传送。

三、协议设计

本次实验中，为了方便对每个数据包的信息进行标识，将序列号从`u_char`类型扩大为了`int`类型，以方便取模。

发送端

发送端增加了三个全局变量数组：

- `clock_t timepkt[32];`
  - 代表**当前窗口内**每个包的计时器
  - 设置数组大小为32是因为窗口大小最大为32
  - 实际使用的部分是窗口大小Windows
  - 每个包的序列号取模Windows后就是他在数组里面对应存储的位置
  - 由于当前窗口内每个包取模Windows都不一样，在下一个窗口内该数组又可以进行重复使用
- `u_char recvack[32] = { 0 };`
  - 代表**当前窗口内**每个包是否接收到确认
  - 设置为1就是收到确认
  - 设置为0就是没有收到确认
- `bool time_work[32] = { 0 };`
  - 由于定时器不太好进行初始化，所以设置一个代表"该定时器是否有效"的数据
  - 在进行超时判断时，只有当定时器有效时，才会进行超时的判断以及最后的重发
- `Packet packet_all[32];`
  - 存储当前窗口内已经发送的数据包，达到缓存的效果
  - 在超时重传时可以直接从该数组中取出

1. 传输单个数据包

```
int send_package(SOCKET& client, SOCKADDR_IN& server_addr, int& serveraddr_len, char* data_content, int datasize, int& seq) {
    //传输单个数据包：每个数据包=头部+数据
    //printf("%d\n", (u_char)seq);
    Packet* sendpkt = new Packet();
    //要发送的内容，包括头和数据部分

    //初始化数据头：
    sendpkt->set_datasize(datasize);//初始化数据长度
    sendpkt->clear_sum();//对序列号进行清零
    sendpkt->set_ack(seq);//初始化序列号seq，注意此时seq是u_char类型
    //sendpkt->print_pkt();
    //此时数据部分不为0，所以要等数据部分初始化后再开始计算校验和

    //初始化数据：
    sendpkt->set_datacontent(data_content);

```

```
    //初始化数据头的校验和:
    sendpkt->set_sum(cksum((u_short*)sendpkt, sendpkt->get_size()));

    //检验发送数据包
    printTime();
    printf("检查数据包内容：\n");
    sendpkt->printpacketmessage();

    //缓存发送数据包
    memcpy(&packet_all[seq % Windows], sendpkt, sendpkt->get_size());

    //发送数据包
    if (sendto(client, (char*)sendpkt, sendpkt->get_size(), 0,
(SOCKADDR*)&server_addr, serveraddr_len) == -1) {
        printf("[Failed send]\nPacket\n\n");
        return -1;
    }
    printTime();
    printf("已成功发送数据包：");
    sendpkt->printpacketmessage();

    return 1;
}
```

本次实验相比于上次实验在"传输单个数据包"部分只加了一句话：

```
//缓存发送数据包
    memcpy(&packet_all[seq % Windows], sendpkt, sendpkt->get_size());
```

这句话的意思是缓存每次发送的数据包，以便在超时重传时直接从该数组中取出。

2. 发送文件

```
int send(SOCKET& client, SOCKADDR_IN& server_addr, int& serveraddr_len, char*
data_content, int datasize) {
    //先算要发多少包：num=len/MAXSIZE+是否有余数（有余数就要多一个包）
    int package_num = datasize / MAXSIZE + (datasize % MAXSIZE == 0 ? 0 : 1);

    cout << datasize << endl;
    //确认号（序列号）从0开始
    //int seqnum = 0;

    printf("------即将开始发送当前文件------\n\n");

    //做好前期准备工作:
    int base = -1;
    //base指向被确认的最后一个数据包
    //base+1就是发送窗口的第一个数据包
    //(nextseqnum-(base+1)就是当前发送了但是没有被确认的数据包的数量
    int nextseqnum = 0;
    //nextseqnum指向即将发送的数据包
```

```
    //clock_t start;
    //设置一个定时器
    while (base != package_num - 1) {
        //在base不等于最后一个包的时候，持续进行发送以及判断

        if ((nextseqnum - 1) - (base + 1) < Windows -1 && nextseqnum != package_num) {
            //增加两个条件：一是发送位置不能等于包的数量，二是发送处和确认处的差值大小必须小于窗口大小（发送缓冲区）
            //base+1指向已发送但是没有被确认的第一个数据包

            //循环发送所有分开后的数据包
            printf("即将发送当前文件中的第%d号数据包：\n", nextseqnum);

            int len = 0;//每次要发送数据包的长度，前面都为MAXSIZE，最后一次发送剩下的
            if (nextseqnum == package_num - 1)
                len = datasize - (package_num - 1) * MAXSIZE;
            else
                len = MAXSIZE;

            //发送每一个数据包
            //int seqnum = nextseqnum % 256;
            if (send_package(client, server_addr, serveraddr_len, data_content + nextseqnum * MAXSIZE, len, nextseqnum) == -1) {
                //如果发送失败的话
                printf("[send package Failed]\n\n");
                //重新发送该数据包
                nextseqnum--;
                continue;
            }
            printf("当前文件中的第%d号数据包发送成功！\n\n", nextseqnum);
            printf("packages:%d\n", package_num);
            printf("windows:%d\n", (nextseqnum - (base + 1)));

            //设置定时器：发送数据包的时候进行初始化
            timepkt[nextseqnum % Windows] = clock();
            //设置该定时器为有效的
            time_work[nextseqnum % Windows] = 1;

            //如果发送成功，发送窗口右端也要向前移动
            nextseqnum++;

            //防止发送方发送的太快所以这里睡眠10ms
            Sleep(10);
        }

        /*此时仍然默认为阻塞模式，需要设置为非阻塞模式*/
        u_long mode = 1;
        ioctlsocket(client, FIONBIO, &mode);

        Header header;
        char* recv_buffer = new char[sizeof(header)];
```

```
67          //开始接收ACK
68
69          if (recvfrom(client, recv_buffer, sizeof(header), 0,
(SOCKADDR*)&server_addr, &serveraddr_len) != -1) {//这里一定要写上条件！因为if的判
断只有0和非0！
70              //如果收到了返回的数据包，首先要进行差错检测和ACK的确认
71              //累积确认
72
73              memcpy(&header, recv_buffer, sizeof(header));
74              if (header.get_tag() == ACK && cksum((u_short*)&header,
sizeof(header)) == 0) {
75                  //返回了确认包，首先要检查校验和之类的
76
77                  if (header.get_ack() >= base + 1)
78                  {
79                      //将该包对应的ack设置为1，表示收到了ack确认
80                      recvack[header.get_ack() % Windows] = 1;
81
82                      printf("发送的数据包已经被确认:\n");
83                      header.print_header();
84
85                      //重置定时器
86                      //timepkt[header.get_ack() % Windows] = 0;
87                      //令定时器失效
88                      time_work[header.get_ack() % Windows] = 0;
89                  }
90
91                  //窗口向前移动
92                  int temp = base+1;
93                  while (true) {
94                      if (recvack[temp%windows] == 1) {
95                          recvack[temp%windows] = 0;
96                          temp++;
97                          base++;
98                          printf("base:%d,nextseqnum:%d\n\n", base, nextseqnum);
99                          continue;
100                     }
101                     break;
102                 }
103
104
105             }
106
107         }
108         else {
109             //检测超时重传
110             for (int i = 0; i < windows; i++) {
111                 int index = (base + 1 + i) % Windows;//当前需要确认的包的序列号对应
的index
112                 if (clock() - timepkt[index] > MAX_TIME&&time_work[index]==1)
{
113                     //超时了且定时器有效，重新发送
```

```
                    printf("重新发送%d号数据包\n", base + i + 1);

                    //从缓存中取出，并且丢进发送缓冲区，进行重新发送
                    char* send_buffer = new char[sizeof(packet_all[index])];
                    memcpy(send_buffer, &(packet_all[index]), packet_all[index].get_size());
                    sendto(client, send_buffer, packet_all[index].get_size(), 0, (SOCKADDR*)&server_addr, serveraddr_len);

                    //发送了，需要重置定时器，定时器一直处于有效状态
                    timepkt[index] = clock();

                }
            }
        }
        delete[]recv_buffer;
        mode = 0;
        ioctlsocket(client, FIONBIO, &mode);
    }

    clock_t now;
    //for循环结束，发送数据包结束，接着开始发送结束标志：over
    //初始化要发送的结束包：
    Header header;
    header.set_tag(OVER);
    header.set_datasize((u_short)0);
    header.set_ack((u_char)0);
    header.clear_sum();
    header.set_sum(cksum((u_short*)&header, sizeof(header)));

    //初始化要发送的数据：
    char* send_buffer = new char[sizeof(header)];
    memcpy(send_buffer, &header, sizeof(header));
    if (sendto(client, send_buffer, sizeof(header), 0, (SOCKADDR*)&server_addr, serveraddr_len) == -1) {
        printf("[Failed send]\nOVER\n\n");
        return -1;
    }
    printTime();
    printf("[send]\nOVER\n\n");

    //存储当前时间
    now = clock();

    u_long mode = 1;
    ioctlsocket(client, FIONBIO, &mode);

    char* recv_buffer = new char[sizeof(header)];

    while (true)
    {
```

```
162         while (recvfrom(client, recv_buffer, sizeof(header), 0,
    (SOCKADDR*)&server_addr, &serveraddr_len) <= 0) {
163             if (clock() - now > MAX_TIME) {
164                 printf("[timeout]\nresend OVER again.....\n\n");
165                 if (sendto(client, send_buffer, sizeof(header), 0,
    (SOCKADDR*)&server_addr, serveraddr_len) == -1) {
166                     printf("[Failed send]\nOVER\n\n");
167                 }
168                 now = clock();
169             }
170         }
171 
172         mode = 0;
173         ioctlsocket(client, FIONBIO, &mode);
174 
175         memcpy(&header, recv_buffer, sizeof(header));
176         if (header.get_tag() == OVER && cksum((u_short*)&header,
    sizeof(header)) == 0) {
177             printTime();
178             printf("[recv]\nOVER\n\n");
179             printf("------对方已接受到文件------\n\n");
180             break;
181         }
182         else
183             continue;
184     }
185     return 1;
186 }
```

这个函数主要分成四个部分：

1. 发送数据包
    - 发送当前需要发送的数据包
    - 设置定时器：时间为当前时间
    - 将计时器设置为有效
    - 发送窗口右端向右移动
2. 接收确认
    - 如果收到的包无误（校验和为0+标志位是`ACK`）且在当前窗口内，则将该包标记为”已收到确认“，即将`recvack`数组中该确认序号对应的位置1；
    - 由于收到了确认，定时器失效
    - 进行窗口的移动：从窗口左边界开始，如果左边界对应的数据包是”已收到确认“的数据包，则左边界右移，不断进行判断知道当前窗口指向的数据包未收到确认
3. 超时重传
    - 从当前窗口左边界开始查找，如果当前对应的数据包对应的计时器满足下列两个条件：
        - 超时
        - 该定时器有效

则从缓存中取出该数据包，进行该数据包的重传；同时由于重新发送了该数据包，所以重置定时器

    - 查找窗口内所有发送的数据包的定时器

4. 所有数据包发送结束，发送`OVER`标识
    - 所有数据包结束后，发送`OVER`数据包，表示数据包发送结束

注意：这里在每次发送完数据包后，会睡眠10ms，这是由于发送端发送数据太快可能会导致接收端缓冲区中数据太多，接收端每一个都进行确认后返回确认包，在后面的数据包被确认的时间增加，返回`ack`时更容易引起超时，也就更容易进行重传，造成接收端缓冲区中的数据更多，造成恶性循环；

- 弊端：发送的时间会被`Sleep(10)`所限制
- 改进思路：增大超时时间`MAX_TIME`，降低超时概率

后续将对增加`Sleep(10)`的前后两种情况进行对比

接收端

接收端增加了两个全局变量数组：

1. `char datadata[32][1024] = { 0 };`
    - 这个数组中的每一个元素都表示存储的数据包的携带的数据
    - 1024是每一个数据包携带数据的最大大小
    - 设置这个数组以帮助缓存乱序数据包
2. `int isdata[32] = { 0 };`
    - 表示`datadata`数组对应元素中是否存储着有效数据（当前窗口中断 对应数据）
    - 为了方便查找，设置`isdata`数组，当该数组元素设置为1时，则表示`datadata`数组中有需要的数据，可以从中取出

3. 接收数据包

```
int recvdata(SOCKET& server, SOCKADDR_IN& client_addr, int& clientaddr_len, char* data) {
    printf("--------开始接收当前文件--------\n\n");
    Packet* recvpkt = new Packet();//接收传过来的数据包（带数据的那种）
    Header header;//发送的确认头（不带数据的那种）

    //int seq = -1;//seq是上一个已经确认的序列号
    int seq_predict = 0;//seq_predict是期待收到的序列号（确认号）

    int file_len = 0;
    //目前已经保存的文件的长度--用于标识data应该从哪里开始存

    char* recv_buffer = new char[MAXSIZE + sizeof(recvpkt->get_header())];
    //接收缓存区
    char* send_buffer = new char[sizeof(header)];
    //发送缓存区

    while (true) {
```

```
        //循环接受所有的数据包，退出条件是数据包接收完毕

        int length = recvfrom(server, recv_buffer, sizeof(recvpkt->get_header()) + MAXSIZE, 0, (SOCKADDR*)&client_addr, &clientaddr_len);
        if (length == -1) {
            printf("[Failed recv]\n");
        }

        //设置丢包
        if ((rand() % 100) + 1 < random)
        {
            printf("该包不做存储，进行人工丢弃\n\n");
            continue;
        }

        printf("延时%dmseconds\n\n", delay);
        Sleep(delay);

        memcpy(recvpkt, recv_buffer, sizeof(recvpkt->get_header()) + MAXSIZE);
        recvpkt->printpacketmessage();

        //判断是否结束，如果已经是最后一个包，则退出接收
        if (recvpkt->get_tag() == OVER&& cksum((u_short*)recvpkt, sizeof(recvpkt->get_header())) == 0) {
            printTime();
            printf("[recv]\nOVER\n\n");
            break;
        }

        //返回该包的ack
        header.set_tag(ACK);
        header.set_datasize(0);
        header.clear_sum();
        header.set_ack(recvpkt->get_ack());
        header.set_sum(cksum((u_short*)&header, sizeof(header)));

        memcpy(send_buffer, &header, sizeof(header));

        if (sendto(server, send_buffer, sizeof(header), 0, (SOCKADDR*)&client_addr, clientaddr_len) == -1) {
            printf("[Failed send]\n\n");
        }
        printTime();
        printf("已发送确认：\n");
        header.print_header();
        printf("\n");

        if(recvpkt->get_ack()>=seq_predict)
        {
            //对接收到的数据包进行缓存
            memcpy(datadata[(recvpkt->get_ack()) % Windows], recvpkt->get_data_content(), recvpkt->get_datasize());
```

```
            isdata[(recvpkt->get_ack()) % Windows] = recvpkt->get_datasize();
            printf("%d号数据包缓存成功：\n", recvpkt->get_ack());
            recvpkt->printpacketmessage();
        }

        while (true) {
            if (isdata[seq_predict % Windows] != 0) {
                //取出来
                memcpy(data + file_len, datadata[seq_predict %Windows],
isdata[seq_predict %Windows]);
                //data+file_len表示数据应该从哪里开始存储
                //recvpkt->get_data_content()表示应该存储的数据
                //recvpkt->get_datasize()表示存储数据的长度
                //更新已存储文件长度
                file_len += isdata[seq_predict %Windows];
                printf("存储第%d号数据包\n", seq_predict);

                //已经存储过，该数组后的存储的数据失效
                isdata[seq_predict % Windows] = 0;
                //清空该数据
                memset(datadata[seq_predict % Windows], 0, 1024);
                //期望的序列号+1
                seq_predict++;

                continue;
            }
            break;
        }

    }
    //文件接收完毕，发送OVER
    header.clear_sum();
    header.set_tag(OVER);
    header.set_datasize(0);
    header.set_sum((cksum((u_short*)&header, sizeof(header))));
    memcpy(send_buffer, &header, sizeof(header));
    if (sendto(server, send_buffer, sizeof(header), 0,
(SOCKADDR*)&client_addr, clientaddr_len) == -1) {
        printf("[Failed send]\n\n");
        return -1;
    }
    printTime();
    printf("[send]\nOVER\n\n");
    printf("--------成功接收当前文件--------\n\n");
    return file_len;//返回读取的字节数，为了之后的存储数据
}
```

这个函数主要干了三件事：

1. 接收数据包
    - 设置阻塞模式的`recvfrom`函数，不断接收缓冲区中传过来的数据包

- 对传输过来的数据包进行检测，如果不是`OVER`包且校验和正确，则发送接收到的数据包的确认包

2. 缓存数据包
   - 对接收到的数据包进行缓存：当接收到的数据包的序列号在当前期待接收到的序列号后面，说明是当前仍未接收到的数据包，对该数据包进行缓存
     - 将接收到的数据包的有效数据部分（即文件对应的数据部分）存储至`datadata`数组中
     - 设置`isdata`数组中对应元素为1，表示`datadata`数组中数据有效
3. 存储数据包中数据（顺序接收到的数据包中的数据+缓存中的数据包中的数据）
   - 如果当前期望收到的数据包对应的`isdata`数组的值为1，表示收到了对应的数据包
     - 将对应`datadata`中的数据取出来存储再接收到的文件中
     - 更新文件长度
     - 将`isdata`中对应的值设置为0，表示该序列号中的数据已经无效
     - 清空`datadata`中该序列号对应的数
   - 循环进行判断直到当前期望收到的数据包对应的`isdata`数组中的值为0结束判断
4. 数据包全部接收完毕后，发送`OVER`数据包
   - 发送标志位为`OVER`的数据包，表示前面发送的数据包全部接收完毕

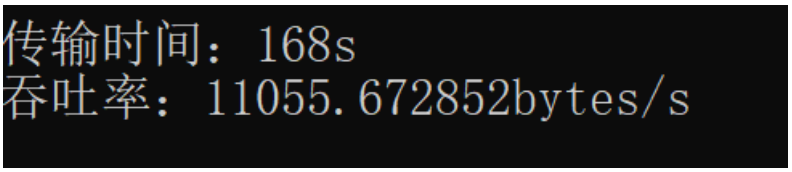
四、实验结果

1. 四个文件的传输验证

条件：丢包率为5%，延时10ms，发送窗口大小为20

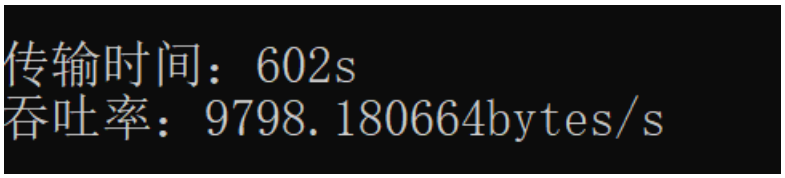
四个文件传输结果如下：

- `1.jpg`
  - 增加`Sleep(10)`前

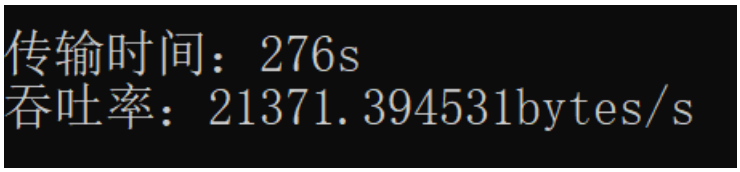


  - 增加`Sleep(10)`后

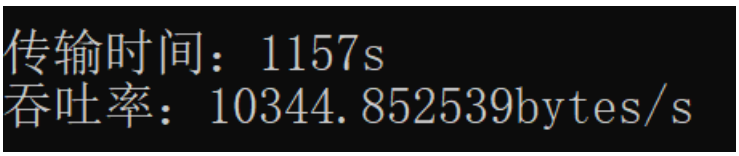
- `2.jpg`
  - 增加`Sleep(10)`前



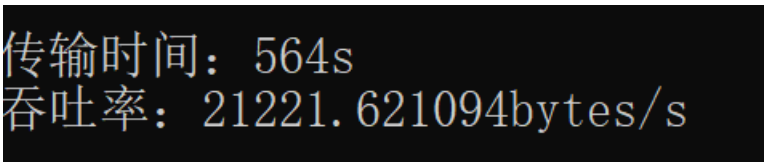
  - 增加`Sleep(10)`后



- `3.jpg`
  - 增加`Sleep(10)`前

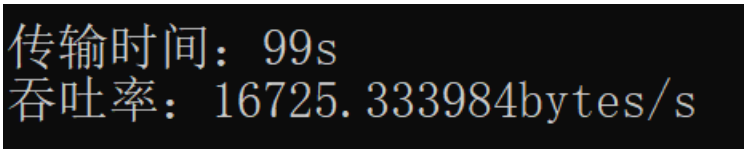


  - 增加`Sleep(10)`后



- `helloworld.txt`
  - 增加`Sleep(10)`前

  - 增加`Sleep(10)`后



传输结果：

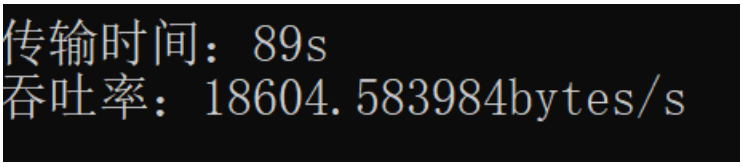
名称	修改日期	类型	大小
x64	2023/12/13 20:54	文件夹	
1.jpg	2023/12/15 0:19	JPG 文件	1,814 KB
2.jpg	2023/12/15 0:25	JPG 文件	5,761 KB
3.jpg	2023/12/15 0:39	JPG 文件	11,689 KB
helloworld.txt	2023/12/15 0:53	文本文档	1,617 KB
Server.cpp	2023/12/14 23:17	C++ 源文件	15 KB
Server.vcxproj	2023/12/13 19:59	VC++ Project	7 KB
Server.vcxproj.filters	2023/12/13 19:59	VC++ Project Filter...	1 KB
Server.vcxproj.user	2023/12/13 19:59	Per-User Project O...	1 KB

可以发现，图片和文本文件均传输成功，实验正确。

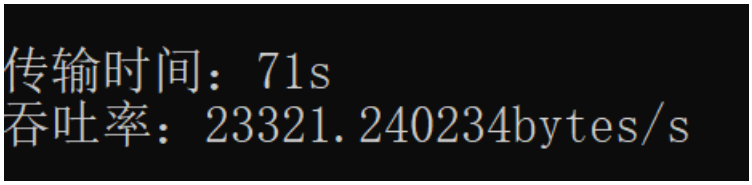
2. 不同窗口大小的传输差别

以`helloworld.txt`为例，将分别对窗口大小为4，8，16，20，32为例进行传输验证来观察差别，结果如下：

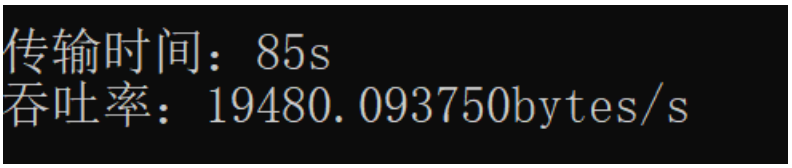
- 窗口大小为4



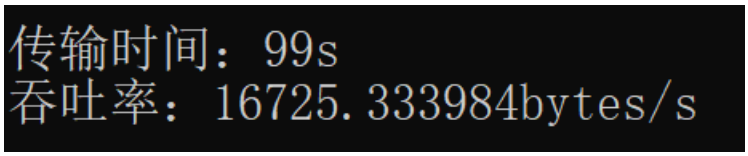
- 窗口大小为8



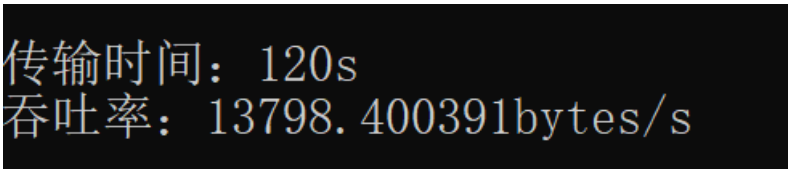
- 窗口大小为16



- 窗口大小为20



- 窗口大小为32



总结如下：

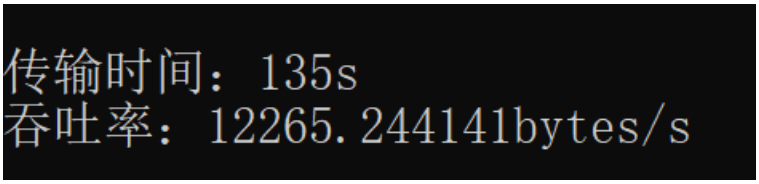
窗口大小	传输时间（s）	吞吐率（bytes/s）
4	89	18604.583984
8	71	23321.240234
16	85	19480.093750
20	99	16725.333984
32	120	13798.400391

可以看到，除了窗口大小为4的情况，从窗口大小为8到窗口大小为32，传输时间逐渐增加，吞吐率逐渐降低，推测原因如下：

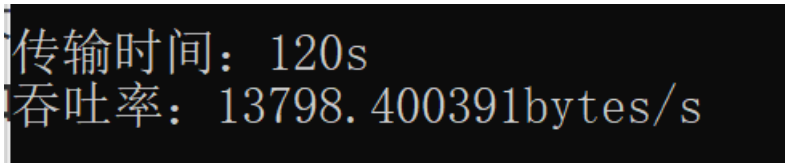
- 超时时间过短，导致许多数据包提前超时，进行超时重传后使得接收端需要确认的数据包增加，增加接收端的负载
- 窗口增大，发送的数据包增多，由于超时时间过短，重传的概率增加，重传的数据包增加，造成吞吐率降低，时间增长

因此得到结论：当前设置的超时时间过短，需要将其增大

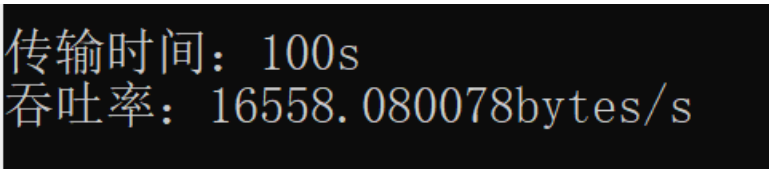
- 窗口大小为4



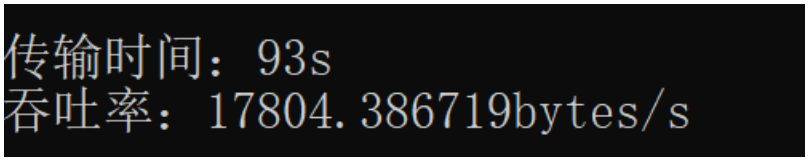
- 窗口大小为8



- 窗口大小为16



- 窗口大小为20



- 窗口大小为32

总结如下：

窗口大小	传输时间（s）	吞吐率（bytes/s）
4	135	12265.244141
8	120	13798.400391
16	100	16558.080078
20	93	17804.386719
32	87	19032.275391

可以看到，随着窗口增加，传输时间在逐步增加，吞吐率也逐渐增加，但是相比于超时时间增大前的情况，窗口大小小的传输时间增加，窗口大小大的传输时间减小：

- 相同情况下，窗口大小增大，可以发送的数据包增加，数据传输效率增加，故传输时间降低，吞吐率增加
- 相同情况下，超时时间增加
    - 对于窗口大小小的情况，可以发送的数据包少，当发生了丢包的情况时，只能等待更长的超时时间
    - 对于窗口大小大的情况，可以发送的数据包多，当发生了丢包的情况时，已经发送的数据包变多，可以被确认的数据包增加，效率变高

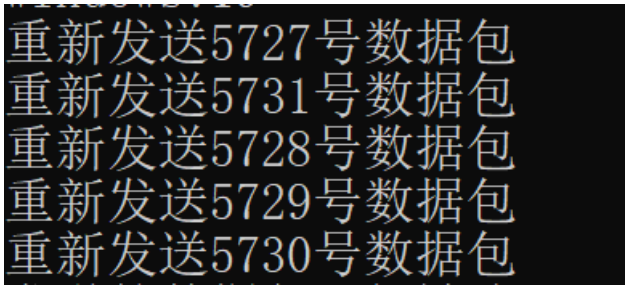
3. 丢包演示和乱序数据包示例

```
该包不做存储，进行人工丢弃

延时10mseconds

Packet size=275 bytes，tag=0，seq=5760，sum=43613，datasize=265
[2023-12-14 23:31:45]
已发送确认:
datasize:0，ack:5760，tag:2
```

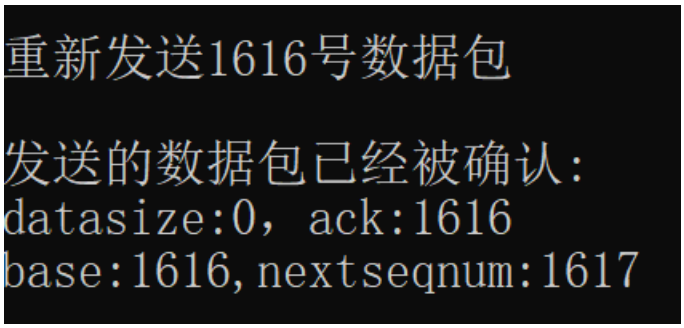
可以看到，即使在该数据包被丢包后，仍然会返回该数据包的确认，符合选择确认的实现。

4. 选择重传部分代码示例

- 增加`sleep(10)`前



  可以看到，这里重新发送了许多数据包，但是不是顺序的，说明发送端只对超时的数据包进行重传：
    - 5727和5731号数据包首先超时进行重传
    - 重传后没有收到任何的确认包
        - 由于我是单线程工作，接收端在收到数据包后会进行数据的存储，存储完成后再开始接收下一个数据包，所以在这一过程中会有很大的延时，导致发送端发送的速度远大于接收端接收的速度，故会产生很多的超时数据包
    - 5728，5729和5730在这一过程中超时，进行重传
- 增加`sleep(10)`后



可以看出，他只重传了一个数据包，数据包超时的可能性降低。

五、总结

通过这次实验，我了解了选择重传协议，选择确认，学会编写`SR`相关代码，领会感悟到了根据结果判断并调整代码参数设置，对于可靠传输协议的知识了解得更加透彻。