南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络实验报告

基于滑动窗口的流量控制机制

张洋 2111460

年级：2021 级

专业：信息安全

指导教师：吴英

2023 年 12 月 15 日

目录

1 实验要求

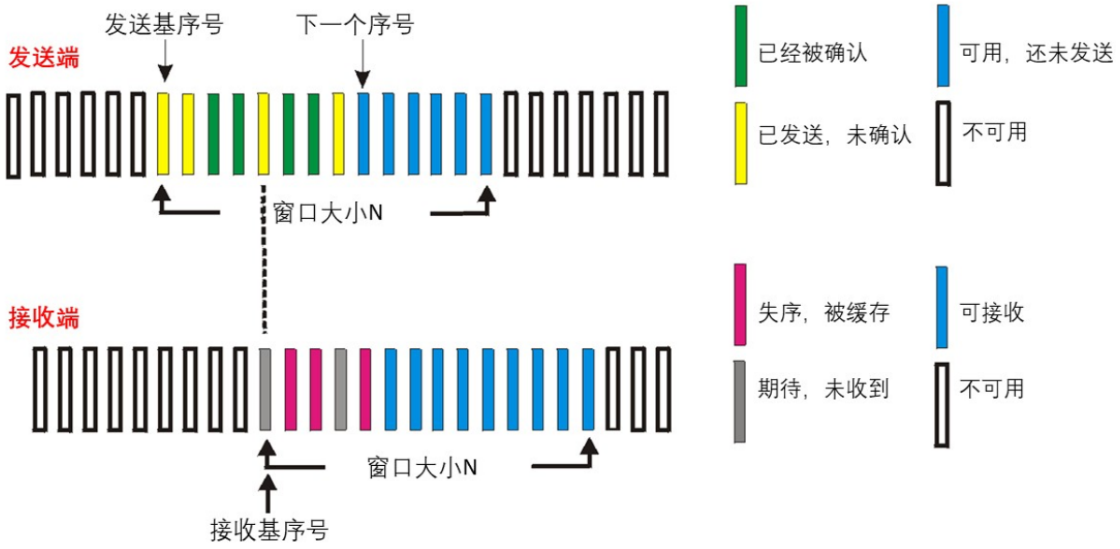
在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，发送窗口和接收窗口采用相同大小，支持选择确认，完成给定测试文件的传输。

- 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
- 发送缓冲区、接收缓冲区
- 选择确认：Selective Repeat
- 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：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

2 前期准备

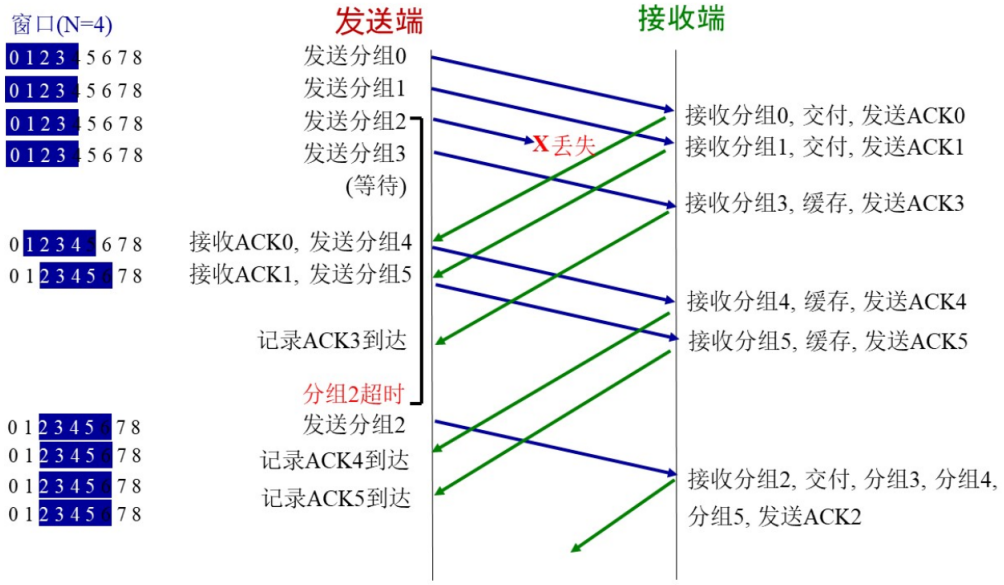
2.1 SR 协议（选择重传协议）

SR 协议在 GBN 协议的基础上进行了改进，它通过让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。选择重传协议只重传真正丢失的分组。



选择重传的接收窗口与发送窗口一样大。选择重传协议允许与接收窗口一样多的分组失序到达，并保存这些失序到达的分组，直到连续的一组分组被交付给应用层。因为发送窗口与接收窗口是相同的，所以发送出来的所有分组都可以失序到达，而且会被保留直到交付为止。在一个可靠的协议中，接收方永远不会把分组失序地交给应用层。在他们被交付给应用层之前，先要等待那些更早发出来的分组到达。

2.2 SR 交互示例



2.2.1 发送方的事件与动作

- 从上层收到数据。当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。如果序号位于发送方的窗口内，则将数据打包并发送。
- 超时。定时器再次被用来防止丢失分组。现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。
- 收到 ACK。如果收到 ACK，倘若该分组序号在窗口内，则 SR 发送方将那个被确认的分组标记为已接收。若该分组的序号等于 send_base，则窗口基序号向前移动到具有最小序号的未确认分组处。如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

2.2.2 接收方的事件与动作

- 序号在 [rcv_base, rcv_base+N-1] 内的分组被正确接收。在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。如果该分组以前没收到过，则缓存该分组。如果该分组的序号等于接收端的基序号 (rcv_base)，则该分组以及以前缓存的序号连续的 (起始于 rcv_base 的) 分组交付给上层。然后，接收窗口按向前移动分组的编号向上交付这些分组。
- 序号在 [rcv_base-N, rcv_base-1] 内的分组被正确收到。在此情况下，必须产生一个 ACK，即使该分组是接收方以前确认过的分组。
- 其他情况。忽略该分组。

3 实验代码及解释

3.1-3.5 中的内容沿用实验 3-1 中的代码，此次实验中实现的内容可以直接**点此跳转**到 3.6 实现 SR 的地方。

3.1 数据报格式

我设计的数据报格式如下图所示：

0-15	16-31	32-47	48-63
Checksum	seq	ack	flag
length	Source_ip		Des_ip
Des_ip	Source_port	Des_port	data
data	...	...	...

```
struct Message
{
    //校验和
    unsigned short checkNum;//2字节
    //序号 Seq num
    unsigned short SeqNum;//2字节
    //确认号 Ack num
    unsigned short AckNum;//2字节
    //标志
    unsigned short flag;//2字节
    //数据大小
    unsigned short size;//2字节
    //源IP、目的IP
    unsigned int SrcIP, DestIP;//4字节、4字节
    //源端口号、目的端口号
    unsigned short SrcPort, DestPort;//2字节、2字节

    BYTE msgData[MaxMsgSize];
    Message();
    bool checksum();
    void setchecksum();
};
```

报文头部

flag 段：

SYN: 在 TCP 连接建立过程中（三次握手），SYN 标志被设置以表示初始的序列号。发送 SYN 的一方想要建立一个连接，并在这个 SYN 包中包含它自己的初始序列号。

ACK: ACK 标志被设置表示这个 TCP 包是一个确认包，并且确认号字段是有效的。接收方通过发送 ACK 包，告知发送方它已经接收到了哪些数据。

FIN: FIN 标志被设置表示发送方没有更多的数据需要发送，即它想要关闭连接的一方。这是 TCP 连接终止过程（四次挥手）的一部分。

RST: RST 标志被设置表示 TCP 连接应该被立即重置。这通常在发生错误或者需要立即中断一个连接时使用。

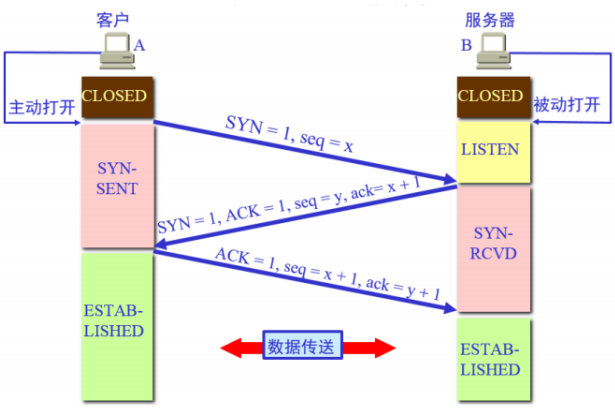
```
const unsigned short SYN = 0x1;
const unsigned short ACK = 0x2;
const unsigned short FIN = 0x4;
```

```
4 const unsigned short RST = 0x8;
```

flag 标志位

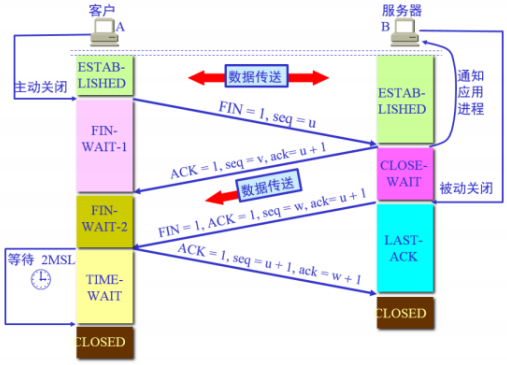
3.2 建立连接

建立连接阶段主要是三次握手以及在第三次握手后增加的一个确认机制，第一次由客户端向服务端发送请求，发送 SYN，服务端确认数据报无误后返回 SYN_ACK 表示确认握手请求，最后再由客户端发送 ACK 表示三次握手结束。



3.3 断开连接

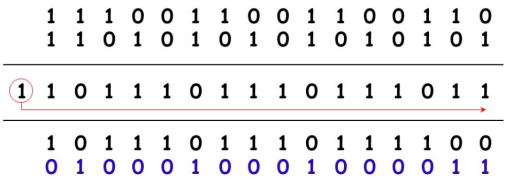
断开连接阶段主要是四次挥手以及对四次挥手的改进。先由客户端发起请求，发送 FIN，随后客户端先后发送 ACK 以及 FIN_ACK 来作为确认请求发送给客户端，最后客户端在发送 ACK 后等待两个 MSL 后中断连接。



3.4 差错检验

校验和的计算是差错检验的核心途径，具体的校验和计算方法就是传入一个 char* 指针指向需要计算校验和的起始地址以及他的字节长度，根据字节长度计算需要计算多少次校验和。

校验和的具体计算流程为对每一个 16 位数进行二进制相加，如果产生进位就把最高位加到最后一位，等到所有的计算结束后再进行取反。下图是校验和计算的示例。




```
u_short checksum(u_short* mes, int size) {
    int count = (size + 1) / 2;
    u_short* buf = (u_short*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, mes, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

校验和

3.5 日志输出

在客户端会输出一个传输日志，记录了传输总时长和吞吐率。

```
//计算传输时间和吞吐率
    cout << "============================输出日志================================" <<
        endl;
    int endTime = clock();
    cout << "总体传输时间为: " << (endTime - startTime) / CLOCKS_PER_SEC << " s" <<
        endl;
    cout << "吞吐率: " << ((float)fileSize) / ((endTime - startTime) /
        CLOCKS_PER_SEC) << " byte/s" << endl << endl;
```

传输日志

3.6 选择重传（SR）

3.6.1 总体流程

- 客户端：先发送文件名和文件大小，文件名通过报文数据段传递，文件大小通过报文的 size 字段传递。然后依据 SR 协议依次发送文件内的数据部分，分为最大装载报文和剩余部分的报文。
- 服务器端：收到文件名和文件大小，根据文件大小计算要收到多少个报文依据 SR 协议等待接收，当收到了所有报文即结束接收。

3.6.2 发送端

发送端修改对滑动窗口以及发送线程的操作，具体代码及分析如下：

与之前一致由于发送端需要发送数据的同时处理接收来自接收端的确认信息，所以本实验中利用双线程来实现发送端的传输部分：主线程只负责发送滑动窗口内的数据，另一个线程则需要处理应答报文并负责窗口的滑动，具体更改分析如下：

主线程（发送滑动窗口内的数据）：

发送数据前一部分判断结束标志的部分未作更改，此处不再说明。发送数据前需要判断当前准备发送数据的序列号 nextSeqNum 是否在滑动窗口中且没有收到过 ack，如果是则将数据打包成数据报文，并调用 makePacket 函数构建对应的数据包保存到缓冲区 sendPkt 中，之后使用 sendto 函数发送给服务端。在这个过程中窗口中的每一个数据包都需要利用计时器进行超时重传的判断。

- 每个数据包的计时。对于每个数据包创建结构体，sendTime 记录发送数据包的时间，active 记录计时器当前的状态。

```
struct PacketTimer {
    clock_t sendTime;
    bool active;
};
```

- 在发送包时，记录发送时间并将计时器设为活动状态：

```
packetTimers[nextSeqNum].sendTime = clock(); // 记录发送时间
packetTimers[nextSeqNum].active = true; // 设置计时器为活动状态
```

- 如果发生超时进入 time_out 标签，需要遍历滑动窗口中的所有数据包，判断窗口内每个数据包是否超时，如果 clock() - sendTime 大于设置的超时时间，则需要重传并重新启动计时。

```
packetTimers[i].sendTime = clock(); // 重置计时器
```

发送线程不停循环发送数据和判断超时，直到全部数据发送完毕再退出。

为了保护共享数据，在更新缓冲区时必须使用互斥锁来避免多个线程同时修改数据结构可能引发的冲突。

```
        packetDataLen = min(MAX_DATA_SIZE, len - sendIndex * MAX_DATA_SIZE);
        mutexLock.lock();//互斥锁保护共享资源
        //[3-3]添加判断条件只发送没有收到ack的包
        if (inWindows(nextSeqNum) && nextSeqNum < packetNum &&
            ackReceived[nextSeqNum] == false) {
            memcpy(data_buffer, fileBuffer + nextSeqNum * MAX_DATA_SIZE,
                packetDataLen);
            sendPkt = makePacket(nextSeqNum, data_buffer, packetDataLen);
            memcpy(pkt_buffer, &sendPkt, sizeof(Packet));
            packetTimers[nextSeqNum].sendTime = clock(); // 记录发送时间
            packetTimers[nextSeqNum].active = true; // 设置计时器为活动状态
            nextSeqNum = (nextSeqNum + 1) % MAX_SEQ;
```

```
            //ShowPacket(&sendPkt[(int)waitingNum(nextSeqNum)]);
            sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR*)&addr, addrLen);
            cout<<"已发送seq = " << nextSeqNum - 1 << "的包"<<endl;
            sendIndex++;
        }
        mutexLock.unlock();
    time_out:
        mutexLock.lock();
        for (int i = base; i != nextSeqNum; i++) {
            if (packetTimers[i].active && (clock() - packetTimers[i].sendTime >=
                MAX_TIME)) {
                // 如果包超时，重发这个包
                packetDataLen = min(MAX_DATA_SIZE, len - i * MAX_DATA_SIZE);
                memcpy(data_buffer, fileBuffer + i * MAX_DATA_SIZE, packetDataLen);
                sendPkt = makePacket(i, data_buffer, packetDataLen);
                memcpy(pkt_buffer, &sendPkt, sizeof(Packet));
                sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR*)&addr,
                    addrLen);
                cout << "超时，正在重新发送seq = " << i << "的包" << endl;
                packetTimers[i].sendTime = clock(); // 重置计时器
                ShowPacket(&sendPkt);
            }
        }
        mutexLock.unlock();
```

另一线程（处理应答报文并负责窗口的滑动）：

该线程始终在非阻塞模式下通过 if (recvfrom(*clientSock, recvBuffer, sizeof(Packet), 0, (SOCKADDR *) addrSrv, addrLen) > 0) 判断是否有报文到达。如果有则需要判断其中的 ack 值：

- 如果收到的 ack 等于 base，设置 ackReceived 数组中的值为 true，停止对应包的计时器，并通过 while 循环查找到第一个 ackReceived 为 false 的地方，把 base 移动到这个地方。
- 如果收到的 ack 不等于 base，且在窗口内，只需要设置 ackReceived 数组中的值为 true，停止对应包的计时器。

```
//判断其中的ack值是否在窗口内
if (base < (recvPacket.head.ack + 1) && recvPacket.head.ack < (base+windowSize)) {
    cout << "接收应答: ";
    ShowPacket(&recvPacket);
    //[3-3]如果当前收到的ack等于base
    if(base == recvPacket.head.ack){
        ackReceived[recvPacket.head.ack] = true;
        packetTimers[recvPacket.head.ack].active = false; // 停止对应包的计时器
        while(ackReceived[base]){
            base++;
            cout << "[窗口滑动]base:" << base << " nextSeq:" << nextSeqNum << "
                endWindow:"
                            << base + windowSize << endl;
        }
```

```
    }
    else{
        ackReceived[recvPacket.head.ack] = true;
        packetTimers[recvPacket.head.ack].active = false; // 停止对应包的计时器
    }
}
```

3.6.3 接收端

服务端对非顺序到达的数据包取下数据段复制缓存中，同时会回复一个对应发送数据包的 ACK 报文，接收到 base 对应的报文后滑动窗口，直到全部接收，具体代码及分析如下：

与之前实验同理，服务端并不知道传输文件何时结束，所以时钟需要判断接收到的报文 flag 中 END 位是否被置为 1，如果是则说明文件传输完毕。

如果到达的报文是正常的数据报文且 recvPkt.head.seq == base_stage，即收到了窗口中最小的序列号，设置 receivedPkt 数组中对应的位置为 true，然后判断是不是第一次收到此报文，如果是第一次收到则将其缓存。通过 while 循环查找到第一个 receivedPkt 为 false 的地方，把 base 移动到这个地方，即下一个想要收到的最小序列号。最后给客户端发送一个对应的 ack。

如果到达的报文是正常的数据报文，序列号小于 base_stage + windowSize 且 recvPkt.head.seq != base_stage，即收到了除窗口中最小的序列号外的其他序列号，设置 receivedPkt 数组中对应的位置为 true，然后判断是不是第一次收到此报文，如果是第一次收到则将其缓存。最后给客户端发送一个对应的 ack。

其他情况发送给客户端一个 ack = base_stage 的报文。

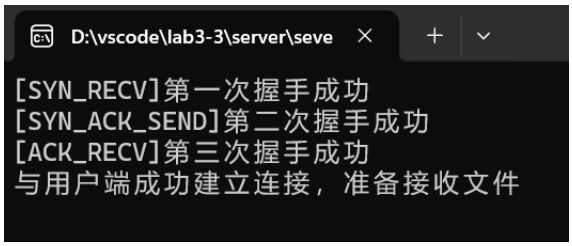
```
u_long recvmessage(char* fileBuffer, SOCKET& socket, SOCKADDR_IN& addr) {
  u_long fileLen = 0;//初始化变量fileLen，用于跟踪接收到的文件长度
  //获取地址结构体的大小
  int addrLen = sizeof(addr);
  u_int expectedSeq = base_stage;
  int dataLen;

  char* pkt_buffer = new char[sizeof(Packet)];
  Packet recvPkt, sendPkt = makePacket(base_stage);
  clock_t start;
  bool clockStart = false;
  while (true) {
      memset(pkt_buffer, 0, sizeof(Packet));
      recvfrom(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR*)&addr, &addrLen);
      memcpy(&recvPkt, pkt_buffer, sizeof(Packet));
      //判断接收到的报文flag中END位是否被置为1
      //如果是则说明文件传输完毕
      if (recvPkt.head.flag & END && CheckPacketSum((u_short*)&recvPkt,
          sizeof(PacketHead)) == 0) {
          cout << "传输完毕" << endl;
          PacketHead endPacket;
          endPacket.flag |= ACK;
          endPacket.checkSum = CheckPacketSum((u_short*)&endPacket,
              sizeof(PacketHead));
```

```
            memcpy(pkt_buffer, &endPacket, sizeof(PacketHead));
            sendto(socket, pkt_buffer, sizeof(PacketHead), 0, (SOCKADDR*)&addr,
                addrLen);
            //[3-3]把缓存的包都填入fileBuffer
            if(fileLen % MAX_DATA_SIZE){
                for(int i = 0 ; i < fileLen / MAX_DATA_SIZE ; i++)
                    memcpy(fileBuffer + i * MAX_DATA_SIZE, filetmp[i], MAX_DATA_SIZE);
                memcpy(fileBuffer + fileLen - dataLen, filetmp[fileLen /
                    MAX_DATA_SIZE], fileLen % MAX_DATA_SIZE);
                return fileLen;
            }
            else{
                for(int i = 0 ; i < fileLen / MAX_DATA_SIZE ; i++)
                    memcpy(fileBuffer + i * MAX_DATA_SIZE, filetmp[i], MAX_DATA_SIZE);
                return fileLen;
            }
        }
        //[3-3]判断收到的seq是不是等于base，相等则滑动窗口
        if (recvPkt.head.seq == base_stage && CheckPacketSum((u_short*)&recvPkt,
            sizeof(Packet)) == 0) {
            sendPkt = makePacket(recvPkt.head.seq);
            if(receivedPkt[recvPkt.head.seq] == false){
                //保存报文
                dataLen = recvPkt.head.bufSize;
                memcpy(filetmp[recvPkt.head.seq], recvPkt.data, dataLen);
                fileLen += dataLen;
                receivedPkt[recvPkt.head.seq] = true;
                while(receivedPkt[base_stage]){
                    base_stage++;
                    expectedSeq++;
                    cout << "[窗口滑动]base:" << base_stage << " expectedSeq:" <<
                        expectedSeq<<endl;
                }
            }
            memcpy(pkt_buffer, &sendPkt, sizeof(Packet));
            sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR*)&addr, addrLen);
            cout<<"[send]: ack = " << recvPkt.head.seq <<endl;
            continue;
        }
        else if(recvPkt.head.seq != base_stage && recvPkt.head.seq < base_stage +
            windowSize && CheckPacketSum((u_short*)&recvPkt, sizeof(Packet)) == 0){
            sendPkt = makePacket(recvPkt.head.seq);
            if(receivedPkt[recvPkt.head.seq] == false){
                //保存报文
                dataLen = recvPkt.head.bufSize;
                memcpy(filetmp[recvPkt.head.seq], recvPkt.data, dataLen);
                fileLen += dataLen;
                receivedPkt[recvPkt.head.seq] = true;
            }
```

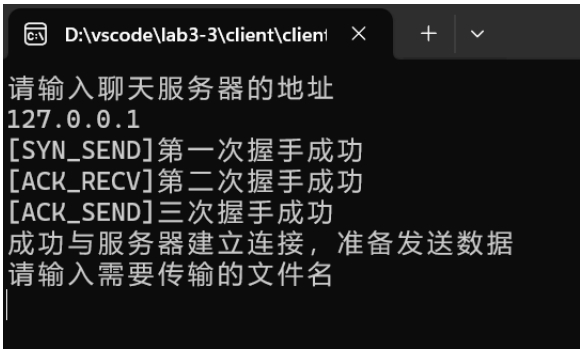
```
67              memcpy(pkt_buffer, &sendPkt, sizeof(Packet));
68              sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR*)&addr, addrLen);
69              cout<<"[send]: ack = " << recvPkt.head.seq <<endl;
70              continue;
71          }
72          cout << "wait head:" << expectedSeq << endl;
73          cout << "recv head:" << recvPkt.head.seq << endl;
74          memcpy(pkt_buffer, &sendPkt, sizeof(Packet));
75          sendto(socket, pkt_buffer, sizeof(Packet), 0, (SOCKADDR*)&addr, addrLen);
76          cout<<"[send]: ack = " << base_stage <<endl;
77      }
78  }
```

3.7 运行界面

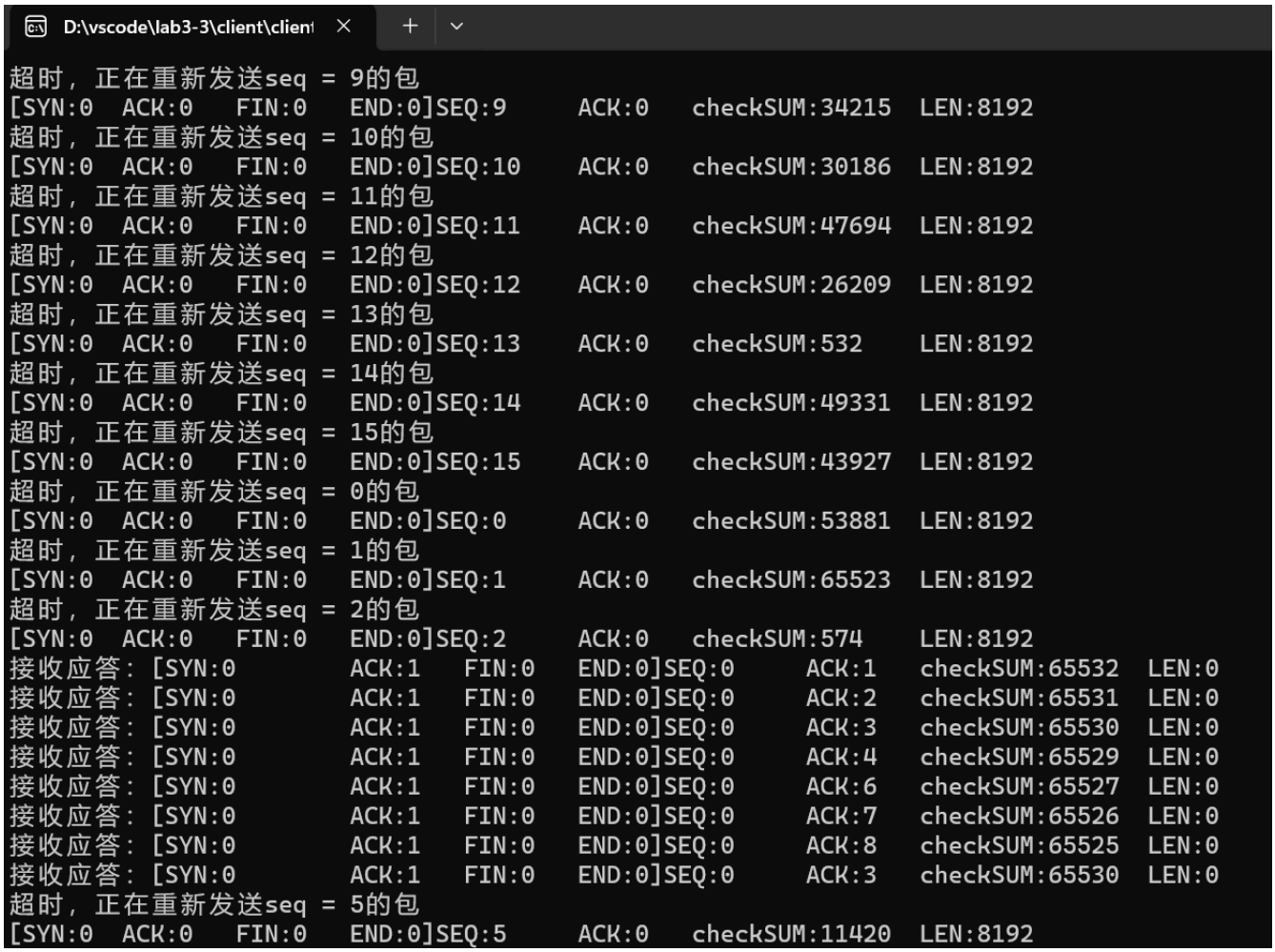
服务器端建立连接



客户端建立连接



发送端 (客户端) 日志输出



接收端 (服务器端) 日志输出

```
[send]: ack = 0
[窗口滑动]base:1 expectedSeq:1
[send]: ack = 1
[窗口滑动]base:2 expectedSeq:2
[send]: ack = 2
[窗口滑动]base:3 expectedSeq:3
[send]: ack = 3
[窗口滑动]base:4 expectedSeq:4
[send]: ack = 4
[窗口滑动]base:5 expectedSeq:5
[send]: ack = 6
[send]: ack = 7
[send]: ack = 8
[send]: ack = 3
[send]: ack = 5
[窗口滑动]base:6 expectedSeq:6
[窗口滑动]base:7 expectedSeq:7
[窗口滑动]base:8 expectedSeq:8
[窗口滑动]base:9 expectedSeq:9
[send]: ack = 9
[窗口滑动]base:10 expectedSeq:10
```

发送端 (客户端) 窗口移动示例

```
接收应答: [SYN:0       ACK:1   FIN:0   END:0]SEQ:0     ACK:12  checkSUM:65521  LEN:0
接收应答: [SYN:0       ACK:1   FIN:0   END:0]SEQ:0     ACK:16  checkSUM:65517  LEN:0
接收应答: [SYN:0       ACK:1   FIN:0   END:0]SEQ:0     ACK:17  checkSUM:65516  LEN:0
接收应答: [SYN:0       ACK:1   FIN:0   END:0]SEQ:0     ACK:5   checkSUM:65528  LEN:0
[窗口滑动]base:6 nextSeq:21 endWindow:22
[窗口滑动]base:7 nextSeq:21 endWindow:23
[窗口滑动]base:8 nextSeq:21 endWindow:24
[窗口滑动]base:9 nextSeq:21 endWindow:25
[窗口滑动]base:10 nextSeq:21 endWindow:26
[窗口滑动]base:11 nextSeq:21 endWindow:27
[窗口滑动]base:12 nextSeq:21 endWindow:28
[窗口滑动]base:13 nextSeq:21 endWindow:29
[窗口滑动]base:14 nextSeq:21 endWindow:30
[窗口滑动]base:15 nextSeq:21 endWindow:31
[窗口滑动]base:16 nextSeq:21 endWindow:32
[窗口滑动]base:17 nextSeq:21 endWindow:33
[窗口滑动]base:18 nextSeq:21 endWindow:34
[窗口滑动]base:19 nextSeq:21 endWindow:35
[窗口滑动]base:20 nextSeq:21 endWindow:36
已发送seq = 21的包
已发送seq = 22的包
```

当前 base = 5, 分别接收到 ack = 12,16,17 的数据包，但是当接收到 ack = 5 的包时窗口滑动，因为之前已经接受过 ack = 6 到 20 的包了，所以直接滑动到下一个接受包为 21 的位置。

接收端 (服务器端) 窗口移动示例

```
[send]: ack = 43
[send]: ack = 38
[send]: ack = 33
[send]: ack = 29
[send]: ack = 43
[send]: ack = 45
[send]: ack = 46
[send]: ack = 47
[send]: ack = 48
[send]: ack = 49
[send]: ack = 50
[send]: ack = 35
[窗口滑动]base:36 expectedSeq:36
[窗口滑动]base:37 expectedSeq:37
[窗口滑动]base:38 expectedSeq:38
[窗口滑动]base:39 expectedSeq:39
[窗口滑动]base:40 expectedSeq:40
[窗口滑动]base:41 expectedSeq:41
[窗口滑动]base:42 expectedSeq:42
[窗口滑动]base:43 expectedSeq:43
[窗口滑动]base:44 expectedSeq:44
[窗口滑动]base:45 expectedSeq:45
[窗口滑动]base:46 expectedSeq:46
[窗口滑动]base:47 expectedSeq:47
[窗口滑动]base:48 expectedSeq:48
[窗口滑动]base:49 expectedSeq:49
[窗口滑动]base:50 expectedSeq:50
[窗口滑动]base:51 expectedSeq:51
```

当前 base = 35, 已经接收到 36-50 的数据包，当收到 seq = 35 的包时，会直接滑动窗口到 51。
客户端断开连接

```
文件传输完成
[FIN_SEND]第一次挥手成功
[ACK_RECV]第二次挥手成功，客户端已经断开
[FIN_RECV]第三次挥手成功，服务器断开
[ACK_SEND]第四次挥手成功，连接已关闭
文件传输完成
```

服务器端断开连接

```
传输完毕
[FIN_RECV]第一次挥手完成，用户端断开
[ACK_SEND]第二次挥手完成
[FIN_SEND]第三次挥手完成
[ACK_RECV]第四次挥手完成，链接关闭
1655808
文件复制完毕
```

3.8 性能测试

3.8.1 更改丢包率

在路由器丢包率设置为 2%，5%，10%，延时设置为 1ms，窗口大小设置为 16，设置窗口中传输的最大字节数为 8192 时，四个文件的时延和吞吐率如下表所示。

	2%		5%		10%	
文件名	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）
1.jpg	9.788945	189739.854499	11.873675	156426.127547	14.619942	127042.432863
2.jpg	41.393240	142499.234174	42.612815	138420.918684	46.491094	126873.869649
3.jpg	79.327091	150881.544364	85.039458	140746.358002	93.301678 s	128282.730349
Helloworld.txt	6.931590	238878.525706	9.527809	173786.859077	11.690835	141632.997130

3.8.2 更改延时

在路由器延时设置为 1ms，3ms，10ms，丢包率设置为 2%，窗口大小设置为 16，设置窗口中传输的最大字节数为 8192 时，四个文件的时延和吞吐率如下表所示。

	1ms		3ms		10ms	
文件名	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）
1.jpg	9.534089	194811.795862	11.268609	164825.401254	13.380382	138811.657246
2.jpg	38.192453	154441.637985	39.182917	150537.669260	41.720547	141381.295888
3.jpg	81.902620	146136.887929	82.416673	145225.396322	89.833086	133235.921562
Helloworld.txt	6.669484	248266.282669	6.990844	236853.804777	10.875782	152247.259094

3.8.3 更改窗口大小

在窗口大小设置为 8，16，32，路由器延时设置为 1ms，丢包率设置为 2%，设置窗口中传输的最大字节数为 8192 时，四个文件的时延和吞吐率如下表所示。

	8		16		32	
文件名	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）	时延（秒）	吞吐率（字节/秒）
1.jpg	9.265378	200461.654128	9.534089	194811.795862	11.682538	158985.401973
2.jpg	27.237133	216561.155684	38.192453	154441.637985	38.955868	151415.057675
3.jpg	54.706877	218784.084495	81.902620	146136.887929	80.926594	147899.391392
Helloworld.txt	7.636154	216837.952718	6.669484	248266.282669	9.453861	175146.218037

4 问题与总结

4.1 更改对每个包进行计时

每个包都有自己独立的计时器，能够更准确地控制每个包的超时重传，而不是采用单一的计时器来统一管理所有包的超时。创建一个结构体存储计时器信息，增加了系统的灵活性，允许更容易地扩展或修改超时控制的策略，以满足不同网络条件或性能需求。独立的计时器使得针对每个包的错误处理更为灵活，能够更准确地确定是哪个包超时，从而更好地应对网络波动和丢包情况。针对每一个包的超时控制可以帮助避免不必要的冗余传输，提高网络利用率和性能。

4.2 窗口大小

可以看到性能测试部分随着窗口大小的增长吞吐率并没有呈现出线性增长的趋势。一般来说，随着窗口的增大，吞吐率也会相应提高，但同时可能会面临一些潜在问题。以下是我通过查阅资料找到的一些可能的影响和应对办法：

- 吞吐率的提高：较大的发送窗口允许同时发送多个数据包，从而增加了网络的利用率，提高了吞吐率。
- 丢包率的影响：随着窗口的增大，网络中的丢包可能导致较大窗口中的多个数据包需要重传，从而降低吞吐率。
- 累计重传次数增加：较大的窗口可能导致一次丢失需要重传的数据包数量增多，因此累计重传的次数可能会相应增加。
- 网络延迟的影响：大窗口可能引入较大的网络延迟，因为发送方需要等待窗口中的所有数据包都被确认才能发送下一批数据。

对策：

- 动态调整窗口大小：可以根据网络状况动态调整发送窗口的大小。在网络较差或丢包率较高时，可以选择较小的窗口，以减少累计重传的影响。
- 拥塞控制机制：结合拥塞控制机制，及时降低发送速率，防止在拥塞情况下大窗口导致的丢包率升高。
- 选择合适的超时时间：调整累计重传的超时时间，确保在网络条件下仍然能够有效地重传丢失的数据包，而不至于过早或过晚触发。
- 快速重传机制：引入快速重传机制，及时重传丢失的数据包，而不必等待累计重传的触发时机。