南開大學

计算机网络实验报告

实验3：基于UDP服务设计可靠传输协议并编程实现（3-2）

姓名:陈翊炀

学号:2113999

专业:信息安全

一、实验要求

> 实验3-2：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，支持累积确认，完成给定测试文件的传输。
>
> (1)实现单向数据传输(一端发数据，一端返回确认)。
>
> (2)对于每个任务要求给出详细的协议设计。
>
> (3)完成给定测试文件的传输，显示传输时间和平均吞吐率。
>
> (4)性能测试指标：吞吐率、延时，给出图形结果并进行分析。
>
> (5)完成详细的实验报告(每个任务完成一份，主要包含自己的协议设计、实现方法、遇到的问题、实验结果，不要抄写太多的背景知识)。
>
> (6)编写的程序应该结构清晰，具有较好的可读性。
>
> (7)提交程序源码、可执行文件和实验报告。

二、实验过程

协议设计

通过课程学习，我们了解到UDP是一种无连接的传输层协议，它提供了面向事务的简单而不可靠的信息传输服务。为了实现基于UDP服务的可靠传输协议，我们需要设计相应的字段来支持可靠传输功能。
下图展示了本次实验自主设计的数据包头部的字段分布。

32 0

FLAG 标志位	HEAD	ACK	SYN	FIN
Seq 序列号				
Ack 序列号				
Len 数据部分长度				
CheckSum 校验和				
Window 窗口				

在设计数据包头结构时，为了实现可靠传输协议RDT3.0，本次实验引入了不同的标志位。
其中，FIN标志用于表示该数据包是断开连接请求，SYN标志用于表示建立连接请求，ACK标志则表示应答数据包。
此外，头部(HEAD)标志指示HEAD数据包，即在传输开始时会先发送的第一个包，它包含了与文件有关的一系列信息。
具体实现的代码如下：

```
class Packet {
public:
    uint32_t FLAG;  // 标志位:HEAD, ACK,SYN,FIN
    uint32_t seq;  // 序列号
    uint32_t ack;  // 确认号
    uint32_t len;  // 数据部分长度
    uint32_t checksum;  // 校验和
    uint32_t window;  // 窗口
    char data[1024];  // 数据长度
    //下略
};
```

对于前三个标志位,设置了以下几种函数来进行设置,来应对不同的情况:

void Packet::setACK() ;

void Packet::setSYN();

void Packet::setSYNACK();

void Packet::setFIN();

void Packet::setFINACK();

其余成员函数的内容如下:

```
void Packet::setHEAD(int seq, int fileSize, char* fileName)
{
    // 设置HEAD位为1
    this->FLAG = 0x8;

    // 对于head包,包含一部分文件的信息,如文件大小,文件名等
    this->len = fileSize;
    this->seq = seq;
    memcpy(this->data, fileName, strlen(fileName) + 1);
}

void Packet::fillData(int seq, int size, char* data) {
    // 将文件数据填入数据包data变量
    this->seq = seq;
    this->len = size;
    memcpy(this->data, data, size);
}
```

- # 三次握手/四次挥手

在上一次实验已经实现,不再赘述

- # 文件传输过程设计

基于滑动窗口的流量控制机制

一、滑动窗口协议概述

滑动窗口协议是基于停止等待协议的优化版本。

停等协议的性能因为需要等待 ack 之后才能发送下一个帧,在传送的很长时间内信道一直

在等待状态。滑动窗口则利用缓冲思想，允许连续发送 (未收到 ack 之前) 多个帧，以加强信道利用滑动窗口的概念：可以看做是缓冲帧的一个容器，将处理好的帧发送到缓冲到窗口，可以发送时就可以直接发送，借此优化性能。一个帧对应一个窗口。
基于滑动窗口协议，我们可以设计出**回退 N 帧**协议，进而还可以设计出选择重传协议。

二、回退 N 帧 (GBN) 协议

GBN 是滑动窗口中的一种，其中发送窗口 > 1，接收窗口 =1 因发送错误后需要退回到最后正确连续帧位置开始重发，故而得名。
控制方法：
· 发送端：在将发送窗口内的数据连续发送
· 接收端：收到一个之后向接收端发送累计确认的 ack
· 发送端：收到 ack 后窗口后移发送后面的数据
累计确认：累计确认允许接收端一段时间内发送一次 ack 而不是每一个帧都需要发送 ack。该确认方式确认代表其前面的帧都以正确接收到。
差错控制：发送帧丢失、ack 丢失、ack 迟到等处理方法基本和停等协议相同，不同的是采用累计确认恢复的方式，当前面的帧出错之后后面帧无论是否发送成功都要重传
优点：信道利用率高（利用窗口有增加发送端占用，并且减少 ack 回复次数）
缺点：累计确认使得该方法只接收正确顺序的帧，而不接受乱序的帧，错误重传浪费严重
下图展示了 GBN 协议可能出现的实际传输情况：

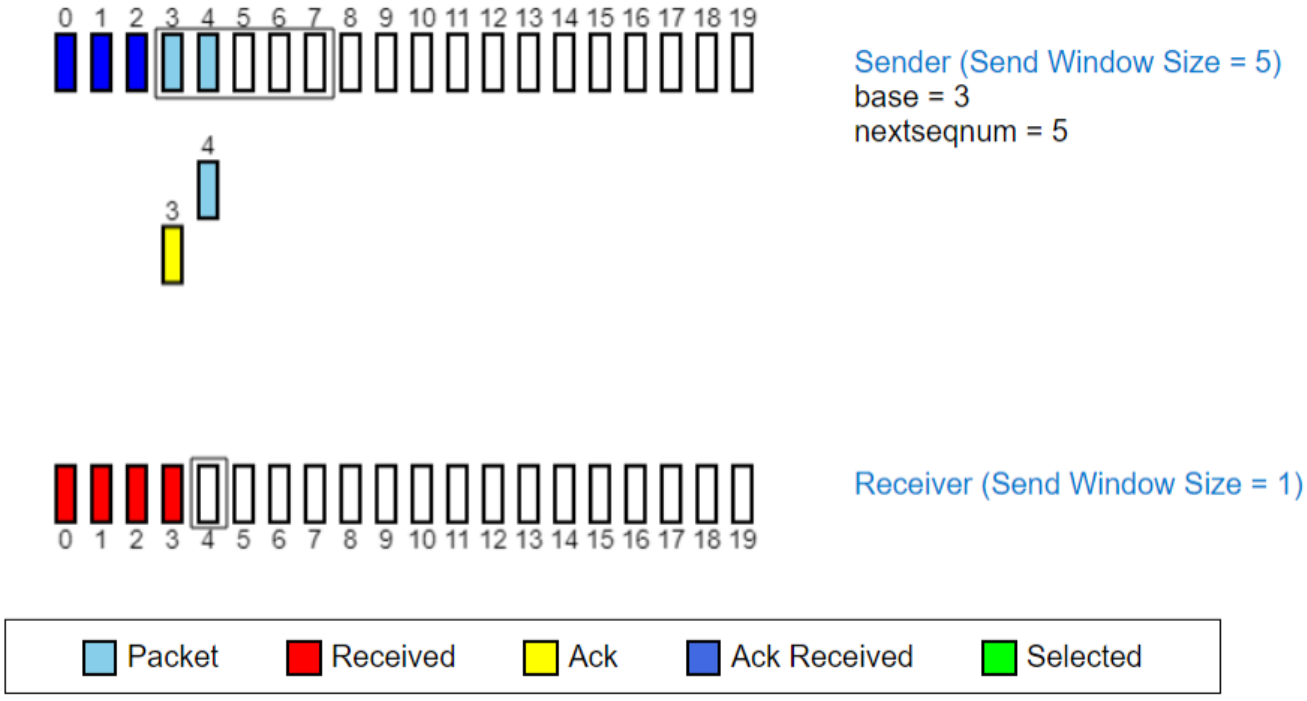


图 1 GBN 协议可能的实际传输情况

本次实验在 GBN 的基础上又实现了选择重传，因此会在下一小节具体讲解代码实现。

三、选择重传 (SR) 协议

（一）选择重传协议简述

SR 协议可以说是 GBN 的增强版本，在 GBN 的基础上改回每一个帧都要确认的机制，解决了累计确认只接收顺序帧的弊端只需要重发错误帧。其中发送窗口 > 1，接收窗口 > 1，接收窗口 > 发送窗口 (本次实验中接收窗口 = 发送窗口)。

控制方法:

- 发送端:将窗口内的数据连续发送
- 接收端:收到一个帧就将该帧缓存到窗口中并回复一个 ack
- 接收端:接收到顺序帧后将数据提交给上层并接收窗口后移(若接收到的帧不是连续的顺序帧时接收窗口不移动)
- 发送端:接收到顺序帧的 ack 后发送窗口后移(同理发送窗口接收到的 ack 不连续也不移动)

差错控制:发送帧丢失、ack 丢失、ack 迟到三类处理方式仍然和停等协议相同,不同的是 SR 向上层提交的是多个连续帧,停等只提交一个帧(不连续的帧要等接收或重传完成后才会提交)

下图展示了 SR 协议可能出现的实际传输情况:

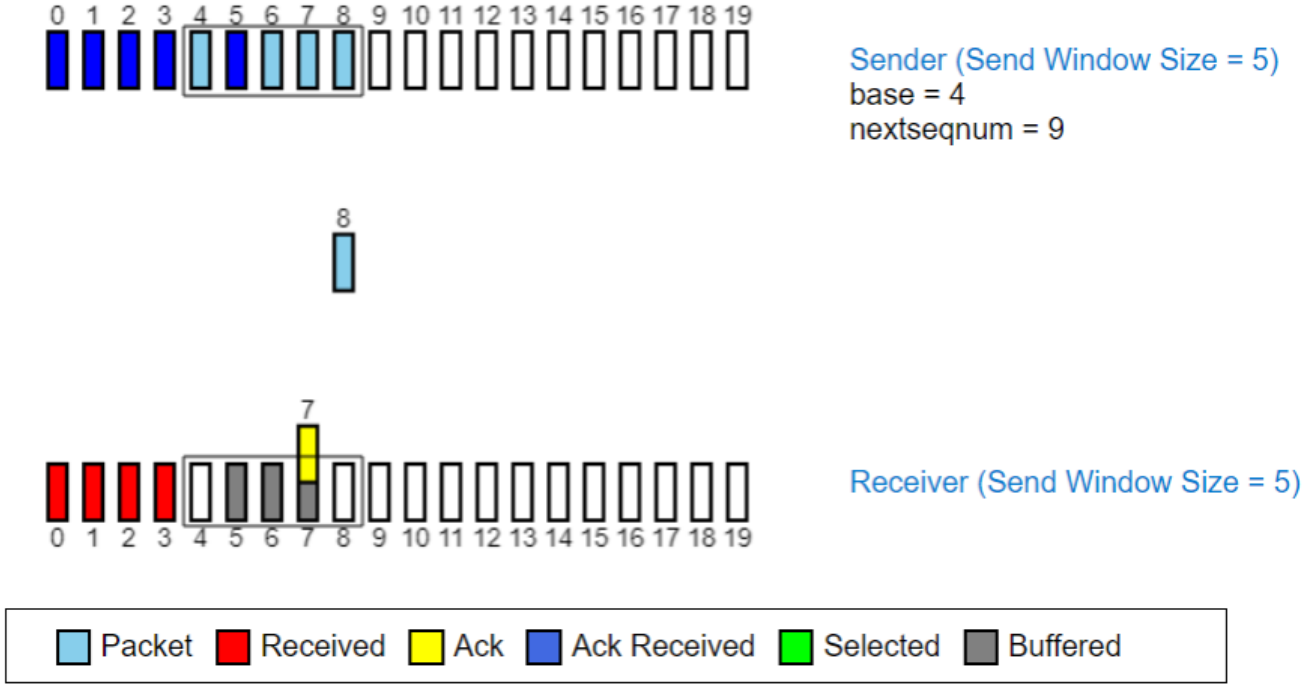


图 2: SR 协议可能的实际传输情况

（二）代码实现

选择重传的具体代码实现中，发送端的实现较为复杂。

首先是基于滑动窗口发送数据包的机制的实现：

```
//....省略中间的代码
#define PACKET_LENGTH 1024
#define BUFFER_SIZE sizeof(Packet)  // 缓冲区大小
#define TIME_OUT 0.2 * CLOCKS_PER_SEC  // 超时重传
#define WINDOW_SIZE 27  // 滑动窗口大小
char** selectiveRepeatBuffer;  // 选择重传缓冲区
unsigned int sendBase;  // 窗口基序号，指向已发送还未被确认的最小分组序号
unsigned int nextSeqNum;  // 指向下一个可用但还未发送的分组序号
//...
while (sendBase < packetNum) //还没发完时
{
    while (nextSeqNum < sendBase + WINDOW_SIZE && nextSeqNum < packetNum)
    {  //还在窗口内持续发送数据包
        if (nextSeqNum == packetNum - 1)
        {  // 如果是最后一个包，处理包的大小
            sendPkt->fillData(nextSeqNum, fileSize - nextSeqNum * PACKET_LENGTH, fileBuffer +
nextSeqNum * PACKET_LENGTH);
            sendPkt->checksum = checksum((uint32_t*)sendPkt);
        }
        else
        {  //正常大小的包(1024
            sendPkt->fillData(nextSeqNum, PACKET_LENGTH, fileBuffer + nextSeqNum *
PACKET_LENGTH);
            sendPkt->checksum = checksum((uint32_t*)sendPkt);
        }
        memcpy(selectiveRepeatBuffer[nextSeqNum - sendBase], sendPkt, sizeof(Packet));  // 已经
发送的包存入缓冲区
        sendPacket(sendPkt);
        timerID[nextSeqNum - sendBase] = SetTimer(NULL, 0, TIME_OUT, (TIMERPROC)resendPacket);

        // 给包设置定时器，用于监控当前发送的数据包是否在规定时间内得到了确认，没有则触发resendPacket超
时重传
        nextSeqNum++;
        Sleep(10);//包与包之间时延
    }

    // 当前发送窗口已经用完则进入接收ACK阶段
    err = recvfrom(socketClient, (char*)recvPkt, BUFFER_SIZE, 0, (SOCKADDR*)&(socketAddr),
&len);
    if (err > 0)
    {
        printRcvPktMsg(recvPkt);
        ackHandler(recvPkt->ack);  // 处理ack
    }
    MSG msg;
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {  // 以查看的方式从系统中获取消息，可以不将消息从系统中移除，是非阻塞函数；当系统无消息时，返回
FALSE，继续执行后续代码。
        if (msg.message == WM_TIMER) {  // 定时器消息
```

```
            DispatchMessage(&msg);
        }
    }
}
```

这里使用了两个指针，**sendBase** 和 **nextSeqNum**，分别指向**已发送还未确认**的**最小分组序号**和**下一个可用还未发送的分组序号**。

1. **sendBase 指针：**
   - `sendBase` 变量代表窗口的基序号，指向已发送但尚未收到确认的最小分组序号。这意味着，`sendBase` 指向的分组以及之前的所有分组都已经发送，并且已经在窗口内等待接收端的确认。
   - 当接收到 ACK 数据包时，`sendBase` 将随着最小确认序号的更新而向前移动，使其指向未被确认的最小分组序号，从而维护了一个动态的发送窗口。

2. **nextSeqNum 指针：**
   - `nextSeqNum` 变量代表下一个可用但还未发送的分组序号。在每次循环迭代中，如果 `nextSeqNum` 小于 `sendBase + WINDOW_SIZE` 且小于 `packetNum`（未发送的分组总数），则表示当前窗口内仍有空闲位置，可以继续发送数据包。
   - `nextSeqNum` 的不断增加确保了发送窗口内有足够的空间来发送新的数据包，同时也反映了发送端的活动窗口。

3. **数据包发送与缓存：**
   - 在发送数据包时，使用 `sendPkt` 结构体进行填充，并将其拷贝到 `selectiveRepeatBuffer` 缓冲区中。这个缓冲区用于存储已发送但未被确认的数据包，以备发生超时需要进行重传。
   - 缓存的数据包通过 `memcpy` 函数复制到 `selectiveRepeatBuffer` 中，保留了每个数据包的内容，用于后续的超时重传。

4. **接收 ACK 阶段：**
   - 当当前窗口内的所有分组都已经发送，或者接收到了 NAK，发送端进入接收 ACK 阶段。这时通过 `recvfrom` 函数接收来自接收端的 ACK 数据包。
   - 调用 `ackHandler` 处理接收到的 ACK 数据包，更新 `sendBase`，并执行相应的窗口移动，以便发送端继续发送新的数据包。

接受ack阶段的具体代码如下：

```
 void ackHandler(unsigned int ack)
{
    // cout << endl << "ack " << ack << endl << endl;
    if (ack >= sendBase && ack < sendBase + WINDOW_SIZE) // 如果ack在窗口内
    {
        KillTimer(NULL, timerID[ack - sendBase]);//取消对应的定时器
        timerID[ack - sendBase] = 0;  // timerID置零

        if (ack == sendBase)
        {   // 如果 ack = sendBase,那么sendBase移动到具有最小序号的未确认分组处
            for (int i = 0; i < WINDOW_SIZE; i++)
            {
                if (timerID[i]) break;  // 遇到有计时器的停下来
                sendBase++;  // sendBase后移
            }
            int offset = sendBase - ack;
            for (int i = 0; i < WINDOW_SIZE - offset; i++)
            {
```

```
                //统一进行平移操作
                timerID[i] = timerID[i + offset];
                timerID[i + offset] = 0;
                memcpy(selectiveRepeatBuffer[i], selectiveRepeatBuffer[i + offset],
sizeof(Packet));
            }
            for (int i = WINDOW_SIZE - offset; i < WINDOW_SIZE; i++) {
                timerID[i] = 0;  // 清除多余的计时器
            }
        }
    }
}
```

这段代码是用于处理接收到的 ACK（确认）的函数 `ackHandler`，主要用于处理选择重传协议中的确认情况。以下是对代码的详细分析：

1. **确认号在窗口内：**
   - 通过条件判断 `ack >= sendBase && ack < sendBase + WINDOW_SIZE` 确保确认号在发送窗口内。

2. **取消对应的定时器：**
   - 如果确认号在窗口内，就通过 `KillTimer` 函数取消与该确认号对应的定时器，即 `timerID[ack - sendBase]`。
   - 将取消的定时器 ID 置零，表示该定时器已经被取消。

3. **移动发送窗口：**
   - 特别地，如果确认号等于 `sendBase`，表示收到的确认对应的是窗口内的最小序号，说明可以将 `sendBase` 移动到未确认分组的最小序号处。
   - 函数通过循环遍历窗口内的所有分组，找到第一个有计时器的位置，即 `timerID[i]` 不为零的位置。这个位置之前的分组都已经收到了确认，可以移动窗口。
   - `sendBase` 向后移动，将 `sendBase` 的值更新到找到的第一个有计时器的位置。
   - 计算移动的偏移量 `offset`，表示移动了多少个位置。

4. **进行统一平移操作：**
   - 通过循环，将未确认的分组整体进行平移操作，以确保窗口内的数据结构保持一致性。
   - 将窗口内的定时器 ID `timerID` 和数据包缓冲区 `selectiveRepeatBuffer` 进行统一平移，以保持窗口内的数据结构。

5. **清除多余的计时器：**
   - 最后，将多余的位置上的定时器 ID 置零，表示这些位置的定时器已经不再需要。

其次是超时重传机制的实现。

不同于 GBN 协议，SR协议的发送端在传输数据包时，需要为每个数据包设置一个独立的计数器，这里采用的方法是每次发送数据包时，都调用 SetTimer 函数来建立一个计时器，该函数会自动计时，到时会发送一个 **WM_TIMER** 消息到消息队列，通过 **PeekMessage** 函数去循环查看消息队列有无超时消息，如果有，就会将该消息派发给对应的处理函数，重传数据包

这部分源代码如下：

```
#define TIME_OUT 0.2 * CLOCKS_PER_SEC // 超时重传
/*
 * 计时器 SetTimer 的返回值,即 Timer 的 ID。
 * 当窗口句柄为 NULL 时,系统会随机分配 ID,因此如果该数组某个元素不为 0,
 * 就代表当前对应的分组有 Timer 在计时。
 * 所以这个数组有两个重要的作用:
 * (1) 数组元素的值对应计时器的 ID
 * (2) 标识一个分组有没有对应的计时器,如果有(即数组元素不为 0),说明该分组未被 ACK。
 */
int timerID[WINDOW_SIZE];
```

超时重传代码如下:

```
void resendPacket(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime)
{   // 重传函数
    // cout << endl << "resend" << " Timer ID " << idEvent << endl << endl;
    unsigned int seq = 0;
    for (int i = 0; i < WINDOW_SIZE; i++)
    {  // 找到是哪个Timer超时了
        if (timerID[i] == idEvent && timerID[i] != 0)
        {
            seq = i + sendBase;
            break;
        }
    }
    cout << "No." << seq << " 数据包对应的计时器超时,重新发送" << endl;

    Packet* resendPkt = new Packet;
    memcpy(resendPkt, selectiveRepeatBuffer[seq - sendBase], sizeof(Packet));  // 从缓冲区直接取出来
    sendPacket(resendPkt);
    printSndPktMsg(resendPkt);
    cout << endl;

}
```

重传函数 `resendPacket`用于处理定时器超时触发的重传事件。以下是对代码的详细分析:

1. **函数签名:**
   - 函数声明为 `void resendPacket(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime)`,这是一个定时器回调函数,会在之前的发送函数中设定的数据包定时器超时时被调用。

2. **定时器超时处理:**
   - 函数首先通过循环遍历 `timerID` 数组,找到触发定时器超时的序列号 `seq`。
   - 通过 `timerID[i] == idEvent && timerID[i] != 0` 条件判断,确定是哪个定时器超时了,找到对应的序列号。

3. **输出信息:**
   - 输出一条信息,表示哪个数据包的计时器超时了,需要进行重传。例如,输出 "No.seq 数据包对应的计时器超时,重新发送"。

4. **重传数据包:**

- 创建一个新的 `Packet` 对象 `resendPkt`，通过 `memcpy` 从选择性重传缓冲区 `selectiveRepeatBuffer` 中复制出需要重传的数据包。
- 调用 `sendPacket` 函数重新发送这个数据包。
- 输出发送的数据包信息，通过 `printSndPktMsg` 函数输出。

在接收端的实现中，我们只需要维护一个 **recvBase** 变量，指向**期待收到但尚未收到的最小分组序号**。与接收端实现有关的基于滑动窗口机制的具体代码如下：

```
#define DATA_AREA_SIZE 1024
#define BUFFER_SIZE sizeof ( Packet ) // 缓冲区大小
#define WINDOW_SIZE 16 // 滑动窗口大小

unsigned int packetNum ; //发送数据包的数量
unsigned int fileSize ; // 文件大小
unsigned int recvSize ; // 累积收到的文件位置
unsigned int recvBase ; // 期待收到但还未收到的最小分组序号
bool isCached [WINDOW_SIZE] ; // 标识窗口内的分组有没有被缓存
char* fileBuffer ; // 读入文件缓冲区
char** receiveBuffer ; // 接收缓冲
```

```
  void receiveFile()
{
    //int nextSeqNum = 0; // 下一个要发送的序号
    int expectedSeqNum = 0; // 期望收到的下一个序号
    int lastAckedSeqNum = -1; // 最后一个确认的序号
    Packet* recvPkt = new Packet;
    Packet* sendPkt = new Packet;
    int err = 0;
    int flag = 1;
    int len = sizeof(SOCKADDR);
    int packetNum = 0;
    float total = 0;  // 总接受的包
    float loss = 0;   // 丢包
    recvSize = 0;
    recvBase = 0;//期待收到但还未收到的最小分组序号
    receiveBuffer = new char* [WINDOW_SIZE];
    for (int i = 0; i < WINDOW_SIZE; i++)receiveBuffer[i] = new char[PKT_LENGTH];
    //接受head包
    while (flag)
    {
        // 等待接受HEAD状态
            err = recvfrom(serverSocket, (char*)recvPkt, BUFFER_SIZE, 0, (SOCKADDR*)&
(ServerAddr), &len);
            if (err > 0) {

                if (isCorrupt(recvPkt)) {  // 检测出数据包损坏
                    printTime();
                    cout << "收到损坏数据包" << endl;
                    //state = 2;
                    //break;
                }
                printRcvPktMsg(recvPkt);
                printTime();
```

```
                if (recvPkt->FLAG & 0x8)
                {  // HEAD=1
                    fileSize = recvPkt->len;
                    fileBuffer = new char[fileSize];
                    fileName = new char[128];
                    memcpy(fileName, recvPkt->data, strlen(recvPkt->data) + 1);
                    packetNum = fileSize % PKT_LENGTH ? fileSize / PKT_LENGTH + 1 : fileSize /
PKT_LENGTH;

                    printTime();
                    cout << "收到文件头数据包,文件名: " << fileName;
                    cout << ", 文件大小: " << fileSize << " Bytes,总共需要接收 " << packetNum <<
" 个数据包";
                    cout << ", 等待发送文件数据包..." << endl << endl;

                    expectedSeqNum++;
                    flag = 0;
                    //state = 2;
                }
                else {
                    printTime();
                    cout << "非文件头数据包,等待发送端重传..." << endl;
                }
            }

    }
    // --------------循环接收文件数据包,基于滑动窗口机制-------------------------
      while (recvBase < packetNum)
    {
        err = recvfrom(serverSocket, (char*)recvPkt, BUFFER_SIZE, 0, (SOCKADDR*)&(ServerAddr),
&len);
        if (err > 0)
        {
            if (isCorrupt(recvPkt))
            {  // 检测出数据包损坏
                printTime();
                cout << "收到一个损坏的数据包,不做任何处理" << endl << endl;
            }
            printRcvPktMsg(recvPkt);
            printTime();
            cout << "收到第 " << recvPkt->seq << " 号数据包" << endl;
            if (randomNumber(randomEngine) >= 0.05)
            {  // 自主进行丢包测试
                unsigned int seq = recvPkt->seq;
                if (seq >= recvBase && seq < recvBase + WINDOW_SIZE)
                {   // 窗口内的分组被接收
                    //发送ack
                    Packet* sendPkt = new Packet;
                    unsigned int ack = recvPkt->seq;
                    sendPkt->setACK(ack);
                    sendto(serverSocket, (char*)sendPkt, BUFFER_SIZE, 0,
(SOCKADDR*)&ServerAddr, sizeof(SOCKADDR));
                    if (!isCached[seq - recvBase])
                    {
                        cout << "首次收到该数据包,将其缓存  ";
```

```
                        memcpy(receiveBuffer[seq - recvBase], recvPkt->data, PKT_LENGTH);
                        isCached[seq - recvBase] = 1;
                    }
                    else {
                        cout << "已经收到过该数据包...";
                    }

                    if (seq == recvBase)
                    {   // 该分组序号等于基序号
                        for (int i = 0; i < WINDOW_SIZE; i++)
                        {  // 窗口移动
                            if (!isCached[i]) break;  // 遇到没有缓存的,停下来
                            recvBase++;
                        }
                        int offset = recvBase - seq;
                        cout << "该数据包序号等于目前接收基序号,窗口移动 " << offset << " 个单位
";
                        for (int i = 0; i < offset; i++)
                        {  // 分组交付上层
                            memcpy(fileBuffer + (seq + i) * PKT_LENGTH, receiveBuffer[i],
recvPkt->len);
                        }
                        for (int i = 0; i < WINDOW_SIZE - offset; i++)
                        {
                            //同步平移
                            isCached[i] = isCached[i + offset];
                            isCached[i + offset] = 0;
                            memcpy(receiveBuffer[i], receiveBuffer[i + offset], PKT_LENGTH);
                        }
                        for (int i = WINDOW_SIZE - offset; i < WINDOW_SIZE; i++)
                        {
                            isCached[i] = 0; //清零已处理的数据包标记
                        }
                    }
                    printWindow();
                    cout << endl;
                }
                else if (seq >= recvBase - WINDOW_SIZE && seq < recvBase)
                {
                    printWindow();
                    cout << "  该数据包分组序列号在["<< recvBase - WINDOW_SIZE <<","<< recvBase
- 1<<"],发送ACK且不进行其他操作..." << endl;
                    unsigned int ack = recvPkt->seq;
                    sendPkt->setACK(ack);
                    sendto(serverSocket, (char*)sendPkt, BUFFER_SIZE, 0,
(SOCKADDR*)&ServerAddr, sizeof(SOCKADDR));
                    // 接受到在这个范围内的分组则发送ACK,不做其他操作
                }
                else {
                    printWindow();
                    cout << "  分组序号 " << seq << " 不在正确范围,不做任何操作" << endl;
                }
            }
            else {
                loss++;
```

```
                cout << "主动丢包" << endl;
            }
        }
    }

    // 跳出循环说明文件接收完毕
    printTime();
    cout << "文件接收完毕..." << endl;
    float lossRate = (loss / packetNum) * 100;
    cout << "丢包率: " << lossRate << "%" << endl;
    cout << endl <<
"**********************************************************************************************
*****" << endl << endl;

    // 保存文件
    saveFile();

    // 文件保存完毕,等待发送端发送断连请求
    flag = 1;
    while (flag) {
        err = recvfrom(serverSocket, (char*)recvPkt, BUFFER_SIZE, 0, (SOCKADDR*)&(ServerAddr),
&len);
        if (err > 0) {
            if (recvPkt->FLAG & 0x1) {  // FIN=1
                flag = 0;
                disconnect();
            }
        }
    }
}
```

关键部分主要介绍如下：

1. **接收数据包：**
   - 使用 `recvfrom` 函数从套接字接收数据包，并将其存储在 `recvPkt` 结构体中。
   - 通过 `isCorrupt` 函数检测数据包是否损坏。如果数据包损坏，则不做任何处理，并输出相应的信息。

2. **丢包测试：**
   - 通过 `randomNumber` 函数生成一个随机数，如果该随机数大于等于 0.05（95%的概率），则执行正常的接收和处理流程；否则，模拟丢包情况，增加丢包计数并输出相应信息。

3. **窗口内分组处理：**
   - 在上述代码中可以看到，对于相对于当前**接收窗口不同区域**分组序号的数据包，接收端会采取不同的处理方法。具体而言：
   - 针对接收到的数据包的序列号（`seq`），根据其在窗口的位置进行不同的处理。
   - 如果 `seq` 处于 `[recvBase, recvBase + WINDOW_SIZE)` 范围内，表示数据包在当前接收窗口内，进行以下操作：
     - 发送 ACK：创建 ACK 数据包，并将其发送给发送端，以确认接收到的数据包。
     - 缓存数据包：将接收到的数据包缓存到 `receiveBuffer` 中，并标记为已缓存。
     - 窗口移动：如果接收到的数据包的序列号等于当前接收基序号 `recvBase`，则窗口移动，将已经缓存的连续分组交付给上层，更新 `recvBase`。
   - 如果 `seq` 处于 `[recvBase - WINDOW_SIZE, recvBase)` 范围内，表示接收到的是已经确认的分组的 ACK，只发送 ACK 给发送端，不进行其他操作。

- 如果 `seq` 不在上述范围内，则不进行任何操作。

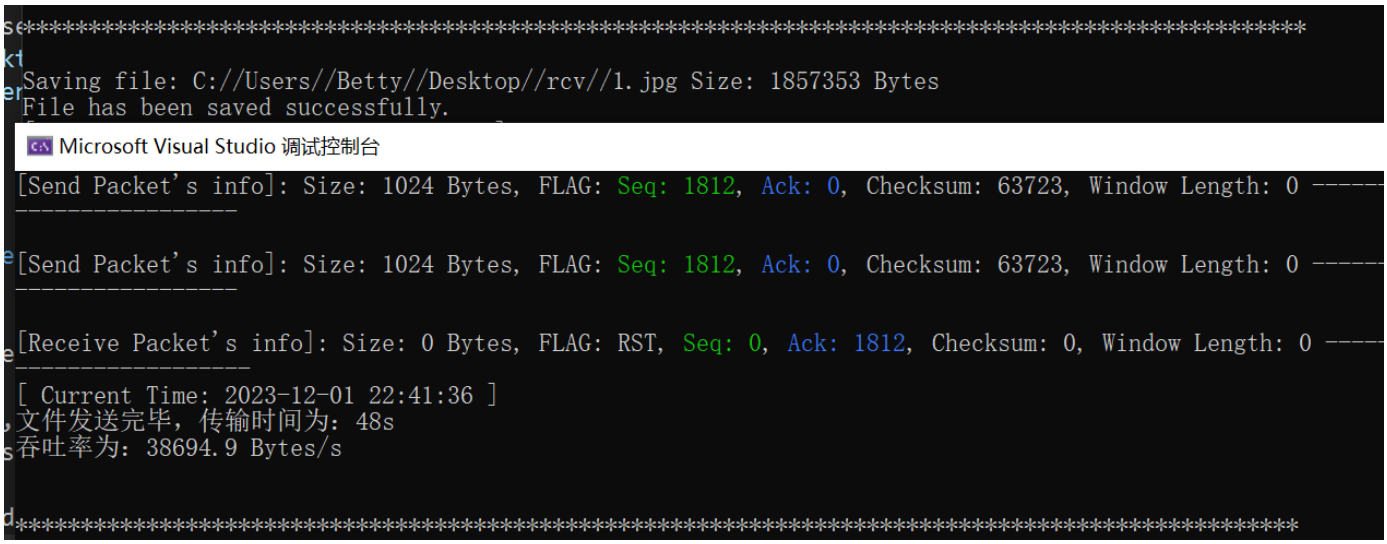
4. **输出信息：**
   - 在每一步操作后，输出相应的信息，包括接收到的数据包信息、窗口状态、丢包信息等。

三、实验实现效果展示

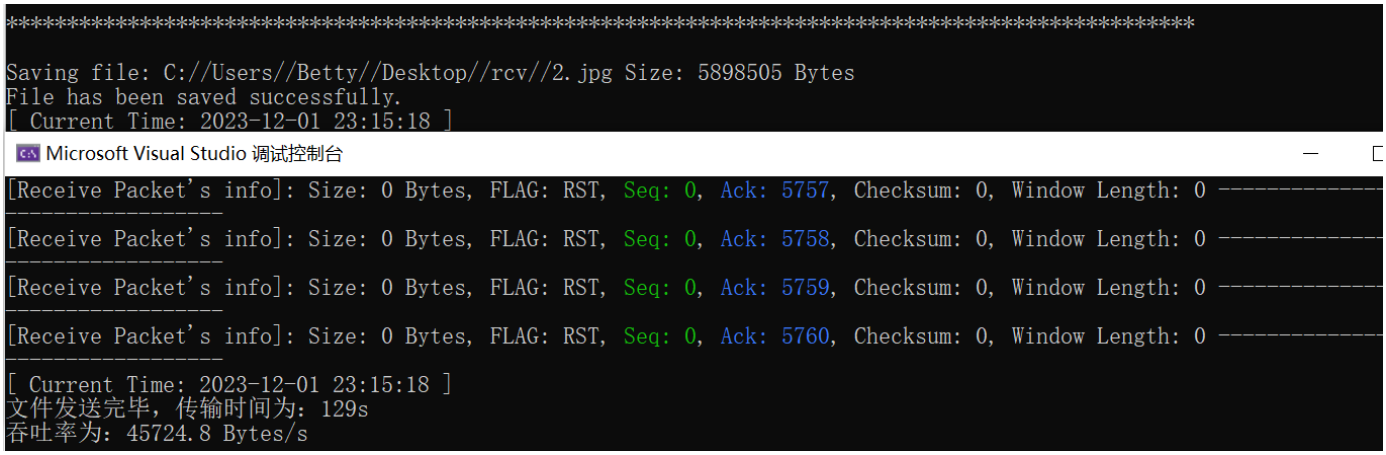
文件传输

测试文件：

1.jpg



2.jpg



3.jpg

```
**********************************************************************************************************

Saving file: C://Users//Betty//Desktop//rcv//3.jpg Size: 11968994 Bytes
File has been saved successfully.
```

Microsoft Visual Studio 调试控制台

```
No.11682 数据包对应的计时器超时，重新发送
发送第 11682 号数据包  当前发送窗口：[11682, 11697]
[Send Packet's info]: Size: 1024 Bytes, FLAG: Seq: 11682, Ack: 0, Checksum: 53853, Window Length: 0 ----------
----------

[Send Packet's info]: Size: 1024 Bytes, FLAG: Seq: 11682, Ack: 0, Checksum: 53853, Window Length: 0 ----------
----------

[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 11682, Checksum: 0, Window Length: 0 ----------
----------
[ Current Time: 2023-12-01 23:23:51 ]
文件发送完毕，传输时间为：256s
吞吐率为：46753.9 Bytes/s
```

helloworld.txt

```
**********************************************************************************************************

Saving file: C://Users//Betty//Desktop//rcv//helloworld.txt Size: 1655808 Bytes
File has been saved successfully.
```

Microsoft Visual Studio 调试控制台

```
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1615, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1608, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1602, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1616, Checksum: 0, Window Length: 0 ----------
----------
[ Current Time: 2023-12-01 23:27:04 ]
文件发送完毕，传输时间为：40s
吞吐率为：41395.2 Bytes/s


**********************************************************************************************************
```

正常发送：

接收端:(窗口支持同时移动多个单位

```
----------
[ Current Time: 2023-12-01 22:41:36 ]
收到第 1780 号数据包
首次收到该数据包，将其缓存  该数据包序号等于目前接收基序号，窗口移动 12 个单位  当前接收窗口：[1792, 1818]
[Receive Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1781, Ack: 0, Checksum: 63754, Window Length: 0 ----------
----------
[ Current Time: 2023-12-01 22:41:36 ]
收到第 1781 号数据包
当前接收窗口：[1792, 1818]  该数据包分组序列号在[1765,1791]，发送ACK且不进行其他操作...
[Receive Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1782, Ack: 0, Checksum: 63753, Window Length: 0 ----------
----------
```

```
----------
[ Current Time: 2023-12-01 22:41:36 ]
收到第 1805 号数据包
首次收到该数据包，将其缓存  该数据包序号等于目前接收基序号，窗口移动 1 个单位  当前接收窗口：[1806, 1832]
[Receive Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1780, Ack: 0, Checksum: 63755, Window Length: 0 --------
----------
[ Current Time: 2023-12-01 22:41:36 ]
收到第 1780 号数据包
当前接收窗口：[1806, 1832]  该数据包分组序列号在[1779,1805]，发送ACK且不进行其他操作...
[Receive Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1806, Ack: 0, Checksum: 63729, Window Length: 0 --------
----------
```

发送端：

发送：

```
发送第 1811 号数据包  当前发送窗口: [1792, 1818]
[Send Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1811, Ack: 0, Checksum: 63724, Window Length: 0 ----------
----------

发送第 1812 号数据包  当前发送窗口: [1792, 1818]
[Send Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1812, Ack: 0, Checksum: 63723, Window Length: 0 ----------
----------

发送第 1813 号数据包  当前发送窗口: [1792, 1818]
[Send Packet's info]: Size: 841 Bytes, FLAG: Seq: 1813, Ack: 0, Checksum: 63722, Window Length: 0 ----------
----------
```

处理ack:

```
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1782, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1770, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1783, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1771, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1792, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1793, Checksum: 0, Window Length: 0 ----------
----------
[Receive Packet's info]: Size: 0 Bytes, FLAG: RST, Seq: 0, Ack: 1794, Checksum: 0, Window Length: 0 ----------
----------
```

接收端主动丢包：

```
----------
[ Current Time: 2023-12-01 22:41:36 ]
收到第 1812 号数据包
主动丢包
[Receive Packet's info]: Size: 841 Bytes, FLAG: Seq: 1813, Ack: 0, Checksum: 63722, Window Length: 0 ----------
----------
```

发送端超时重传：

```
No.1806 数据包对应的计时器超时，重新发送
发送第 1806 号数据包  当前发送窗口: [1799, 1825]
[Send Packet's info]: Size: 1024 Bytes, FLAG: Seq: 1806, Ack: 0, Checksum: 63729, Window Length: 0 ----------
----------
```

传输结束

```
Saving file: C://Users//Betty//Desktop//rcv//helloworld.txt Size: 1655808 Bytes
File has been saved successfully.
[ Current Time: 2023-12-01 23:27:04 ]
接收到客户端的断开连接请求，开始第二次挥手，向客户端发送ACK=1的数据包...
[ Current Time: 2023-12-01 23:27:04 ]
开始第三次挥手，向客户端发送FIN,ACK=1的数据包...
[ Current Time: 2023-12-01 23:27:04 ]
收到了来自客户端第四次挥手的ACK数据包...
[ Current Time: 2023-12-01 23:27:04 ]
四次挥手结束，确认已断开连接...
[ Current Time: 2023-12-01 23:27:04 ]
程序退出...
```

```
文件发送完毕，传输时间为：40s
吞吐率为：41395.2 Bytes/s


**********************************************************************************************************

[ Current Time: 2023-12-01 23:27:04 ]
开始第一次挥手，向服务器发送FIN=1的数据包...
[ Current Time: 2023-12-01 23:27:04 ]
收到了来自服务器第二次挥手ACK数据包...
[ Current Time: 2023-12-01 23:27:04 ]
收到了来自服务器第三次挥手FIN&ACK数据包，开始第四次挥手，向服务器发送ACK=1的数据包...
[ Current Time: 2023-12-01 23:27:04 ]
模拟TIME_WAIT
[ Current Time: 2023-12-01 23:27:06 ]
四次挥手结束，确认已断开连接...
[ Current Time: 2023-12-01 23:27:06 ]
程序退出...
```

四、总结

在本次实验之中，我深入地了解了滑动窗口协议、后退 N 帧协议以及选择重传协议，并且对于选择重传协议进行了编程实现。在编写代码的过程中，对于线程以及消息队列的相关知识进行了深入了解，并且进一步理解和体会了计算机网络中传输协议的设计与实现。