

计算机网络第四次实验报告

网络空间安全学院 物联网工程 2111673 岳志鑫

一、实验目的

基于 UDP 服务设计可靠传输协议并编程实现（3-2）

二、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，支持累积确认，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
- 累计确认：Go Back N
- 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）

三、实验内容

1. 协议设计

（1）建立连接：

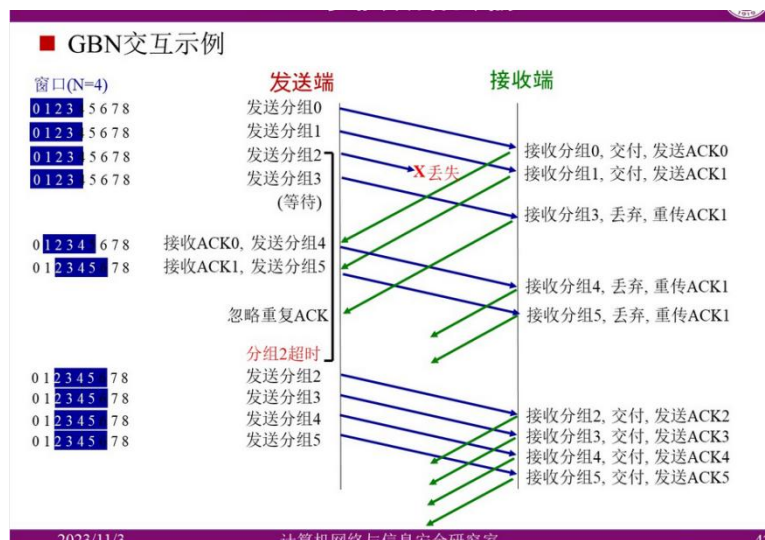
通过握手过程建立连接。发送方发送连接请求数据包，接收方收到连接请求，发送连接确认数据包。

（2）差错检测：

在数据包中添加校验和字段，用于检测数据传输过程中的错误。发送方发送数据包，记录序列号和计算校验和。接收方收到数据包，进行序列号和校验和的检测，如果数据包正确就发送确认，如果数据包错误就重新发送请求。

（3）累计确认：

当接收方成功收到一个数据包时，它发送一个确认（ACK）给发送方。发送方接收到这个确认后就默认这个序号之前的数据包都已传输成功，这样的机制有效地减少了网络上的确认消息数量，从而减小了网络负载。如果发送方出现了丢包的问题，那么接收方就会一直发送这个数据包的序号的 ACK，就不管之后接收到的数据包，当发送方接收到这个序号的 ACK 时就会从这个序号的数据包开始重新传送直到传送成功。



(4) 超时重传:

发送方发送数据包, 启动定时器。如果发送方收到确认则停止定时器, 如果发送方在规定时间内未收到确认, 则进行超时重传, 重新发送上一个数据包。

(5) 滑动窗口

一个窗口的大小是由一个后沿序号和一个前沿序号来控制的。发送窗口的大小是由接收窗口的大小来决定的, 窗口大小不能大于接收方的接收缓冲区中剩余空间的大小, 避免发送大量数据而缓冲区积累满后导致的丢包问题。发送窗口只有收到对方对于本段发送窗口内字节的 ACK 确认, 才会移动发送窗口的后沿。而接收窗口只有在前面所有的段都确认的情况下才会移动后沿。当前面还有字节未接收但收到后面字节的情况下, 窗口不会移动, 并不对后序字节确认。超时后发送端会对上一个接收序号后面的数据包进行重传

(5) 流量控制:

采用滑动窗口机制, 传输的数据包会首先存储在滑动窗口对应的缓冲区中, 如果缓冲区满了, 则滑动窗口也会清零, 发送端就不会发送数据, 等待缓冲区中的所有数据都写入后清空缓冲区, 再将滑动窗口开启, 发送端就会继续发送数据包, 保证了不会出现网络拥塞等问题。

2. 核心代码分析

(1) 数据报结构

```
struct Head
{
    u_short checksum;//校验和 16 位
    u_short datasize;//所包含数据长度 16 位
    unsigned char flag;//八位, 使用后三位表示 FIN ACK SYN
    unsigned char seq;//八位, 传输的序列号
    Head()
    {
        checksum = 0;
        datasize = 0;
        flag = 0;
        seq = 0;
    }
};
```

(2) 三次握手 (以发送端为例)

首先进行第一次握手, 将数据包组装好后发送, 标志位设定为 SYN

```
Head head = Head(); //数据首部
head.flag = SYN; //标志设为 SYN
head.checksum = check((u_short*)&head, sizeof(head)); //计算校验和
char* buff = new char[sizeof(head)]; //缓冲数组
memcpy(buff, &head, sizeof(head)); //将首部放入缓冲数组
if (sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, length)
==SOCKET_ERROR)
{ //发送失败
    cout << "【第一次握手失败】" << endl;
```

```

        return ;
    }
    cout << "第一次握手成功【SYN】" << endl;

```

发送成功后等待客户端的回复，用 while 循环持续接收数据包，如果超时未接收到确认则重新发送第一次握手的数据包，接收到回复则为第二次握手成功

```

clock_t handstime = clock(); //记录发送第一次握手时间
u_long mode = 1;
ioctlsocket(socket, FIONBIO, &mode); //设置非阻塞模式
int handscount1 = 0; //记录超时重传次数
//第二次握手
while (recvfrom(socket, buff, sizeof(head), 0, (sockaddr*)&addr, &length) <= 0)
{
    //等待接收

    if (clock() - handstime > retime) //超时重传
    {
        memcpy(buff, &head, sizeof(head)); //将首部放入缓冲区
        sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, length); //再次发送
        handstime = clock(); //计时
        cout << "【连接超时！等待重传……】" << endl;
        handscount1++;
        if (handscount1 == handscount) {
            cout << "【等待超时】" << endl;
            return;
        }
    }
}
memcpy(&head, buff, sizeof(head)); //ACK 正确且检查校验和无误
if (head.flag == ACK && check((u_short*)&head, sizeof(head) == 0))
{
    cout << "第二次握手成功【SYN ACK】" << endl;
    handscount1 = 0;
}
else
{
    cout << "【第二次握手失败】" << endl;
    return;
}

```

继续发送第三次握手的数据包并等待回复，如果超时未接收到确认则重新发送第三次握手的数据包，接收到回复则为第三次握手成功

```

//第三次握手
head.flag = ACK_SYN; //ACK=1 SYN=1
head.checksum = check((u_short*)&head, sizeof(head)); //计算校验和

```

```

        sendto(socket, (char*)&head, sizeof(head), 0, (sockaddr*)&addr, length); //发送握手
请求
        bool win = 0; //检验是否连接成功的标志
        while (clock() - handstime <= retime)
        { //等待回应
            if (recvfrom(socket, buff, sizeof(head), 0, (sockaddr*)&addr, &length))
            { //收到报文
                win = 1;
                break;
            }
            //选择重发
            memcpy(buff, &head, sizeof(head));
            sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, length);
            handstime = clock();
            handscountl++;
            if (handscountl == handscount) {
                cout << "【等待超时】" << endl;
                return;
            }
        }
        if (!win)
        {
            cout << "【第三次握手失败】" << endl;
            return;
        }
        cout << "第三次握手成功【ACK】" << endl;

```

(3) 校验和计算

校验数据以 16 位为单位进行累加求和，如果累加和超过 16 位产生了进位，需将高 16 位置为 0，低 16 位加一。循环步骤，直至计算完成为止，最后将所取得的结果取反。

```

u_short check(u_short* head, int size)
{
    int count = (size + 1) / 2; //计算循环次数，每次循环计算两个 16 位的数据
    u_short* buf = (u_short*)malloc(size + 1); //动态分配字符串变量
    memset(buf, 0, size + 1); //数组清空
    memcpy(buf, head, size); //数组赋值
    u_long checkSum = 0;
    while (count--) {
        checkSum += *buf++; //将 2 个 16 进制数相加
        if (checkSum & 0xffff0000) { //如果相加结果的高十六位大于一，将十六位置零，并将
最低位加一
            checkSum &= 0xffff;
            checkSum++;
        }
    }
}

```

```

    }
    return ~(checksum & 0xffff); //对最后的结果取反
}

```

(4) 发送文件（以发送端为例）

将文件拆分成多个固定大小为 maxlen=4096 的数据包，并输出总数据包个数，在循环中将所有数据包发送。

```

num = 0;
Head head;
int addrlen = sizeof(addr); //地址的长度
int bagsum = data_len / maxlen; //数据包总数，等于数据长度/一次发送的字节数
if (data_len % maxlen) {
    bagsum++; //向上取整
}
totalnum = bagsum;
int base = -1; //发送窗口开始的位置
int nextseq = 0; //发送窗口结束的位置
char* buff = new char[sizeof(head)]; //数据缓冲区
clock_t starttime; //计时

while(base < bagsum - 1)
{
    int len;
    if (nextseq == bagsum - 1)
    { //最后一个数据包是向上取整的结果，因此数据长度是剩余所有
        len = data_len - (bagsum - 1) * maxlen;
    }
    else
    { //非最后一个数据长度均为 maxlen
        len = maxlen;
    }
    //sendbag 部分、数据段包发送
    //头部初始化及校验和计算
    int addrlen = sizeof(addr);
    Head head;
    char* buf = new char[maxlen + sizeof(head)];
    head.datasize = len; //使用传入的 data 的长度定义头部 datasize
    head.seq = unsigned char(nextseq % 256); //序列号
    memcpy(buf, &head, sizeof(head)); //拷贝首部的数据
    memcpy(buf + sizeof(head), data + nextseq * maxlen, sizeof(head) + len); //
数据 data 拷贝到缓冲数组
    head.checksum = check((u_short*)buf, sizeof(head) + len); //计算数据部分的校验和
    memcpy(buf, &head, sizeof(head)); //更新后的头部再次拷贝到缓冲数组
    //发送

```

```

        sendto(socket, buf, len + sizeof(head), 0, (sockaddr*)&addr, addrlen); //发送
        cout << "-----发送第" << num << "/" << totalnum << "个数据包-----" << endl;
        cout << "【发送】标志位 = " << head.flag << " 序列号 = " << int(head.seq) << " 校验和 = " << int(head.checksum) << endl;
        //没有超过窗口大小且未发送完，并且不是窗口的结尾
        if (nextseq < base + slidewindows && nextseq != bagsum) //没有处理超过窗口大小的情况
        {
            starttime = clock(); //记录发送时间
            nextseq++; //发送窗口后沿向后滑动
        }
        u_long mode = 1;
        ioctlsocket(socket, FIONBIO, &mode); //设置非阻塞模式
        cout << "滑动窗口剩余大小: " << slidewindows - nextseq + base << endl;

```

在接收回复中进行累计确认，对接收到的回复数据包进行检测，如果出现校验和不正确或 head 的标志位不为 ACK 的情况，那么就是出现了丢包的问题，需要将滑动窗口的前沿移动到出现丢包的数据包位置，然后再重新发送。如果持续没有接收到回复则代表发送的数据包丢失，需要进行超时重传。如果接收到数据包回复，则认为这个序号之前的数据包都已成功到达，继续发送数据包。

```

Sleep(50); //休眠 50ms 等待接收，防止频繁超时重传使得频繁反馈丢包信息
if (recvfrom(socket, buf, maxlength, 0, (sockaddr*)&addr, &addrlen)) { //接收消息
    memcpy(&head, buf, sizeof(head)); //缓冲区接收到信息，读取
    u_short checksum1 = check((u_short*)&head, sizeof(head)); //计算校验和
    //如果校验和不正确或没有收到 ACK
    if (int(checksum1) != 0 || head.flag != ACK)
    {
        cout << "-----接收到第" << num++ << "个数据包的回复-----" << endl;
        cout << "【接收】标志位 = " << head.flag << " 序列号 = " << int(head.seq) << " 校验和 = " << int(head.checksum) << endl;
        cout << "【传输错误丢包重传】" << endl;
        nextseq = base + 1; //回到 base+1 号即未被确认的最低序号数据包
        cout << "窗口前沿 = " << base << " 窗口后沿 = " << nextseq << endl;
        num--;
        continue; //丢包处理，进入下一轮循环重新发送
    }
    //接收到正确的数据包，输出接收信息
    else
    {
        if (int(head.seq) >= base % 256)
        { //序号号没用完，可以继续使用
            base += int(head.seq) - base % 256;
            cout << "-----接收第" << num++ << "个数据包的回复-----" <<

```

```
endl;
        cout << "【接收】标志位 = " << head.flag << " 序列号 = " << int(head.seq)
<< " 校验和 = " << int(head.checksum) << endl;
        cout << "滑动窗口数 = " << slidewindows << " 窗口前沿 = " << base <<
" 窗口后沿 = " << nextseq << endl;
        }else{//序列号用完了
            if (base % 256 > 256 - slidewindows - 1)
            {
                base += 256 + int(head.seq) - base % 256;
                cout << "-----接收第" << num++ << "个数据包的回复-----"
<< endl;
                cout << "【接收】标志位 = " << head.flag << " 序列号 = " <<
int(head.seq) << " 校验和 = " << int(head.checksum) << endl;
                cout << "滑动窗口数 = " << slidewindows << " 窗口前沿 = " << base
<< " 窗口后沿 = " << nextseq << endl;
            }
        }
    }
}
else
{//未接收到消息
    if (clock() - starttime > retime)
    {
        nextseq = base + 1; //回退到first+1 处
        cout << "【超时重新发送】" << endl;
    }
}
mode = 0;
ioctlsocket(socket, FIONBIO, &mode); //阻塞
}
```

如果所有数据包发送完毕，则最后组装一个 END 数据包发送给接收端，表示文件发送结束，可以断开连接，同时进行超时重传的检测。

```
//传输完毕
buff = new char[sizeof(head)]; //缓冲数组
newbag(head, END, buff); //调用函数生成 ACK=SYN=FIN=1 的数据包，表示结束
sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, addrlen);
clock_t starttime2 = clock(); //计时

//处理超时重传
while (1)
{
    u_long mode = 1;
    ioctlsocket(socket, FIONBIO, &mode); //设置非阻塞模式
```

```

//等待接收消息
while (recvfrom(socket, buff, maxlength, 0, (sockaddr*)&addr, &addrlen) <=
0)
{
    if (clock() - starttime2 > retime) //超时重传
    {
        char* buf = new char[sizeof(head)]; //缓冲数组
        newbag(head, END, buf); //调用函数生成 ACK=SYN=FIN=1 的数据包，表示
结束
        cout << "【超时等待重传】" << endl;
        sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlen);
//继续发送相同的数据包
        starttime2 = clock(); //新一轮计时
    }
}
memcpy(&head, buff, sizeof(head)); //缓冲区接收到信息，读取到首部
if (head.flag == END)
{ //接收到 END 口令
    cout << "传输成功!" << endl;
    //sendsuccess = 1;
    break;
}
}
u_long mode = 0;
ioctlsocket(socket, FIONBIO, &mode); //改回阻塞模式
}

```

(5) 接收文件（以接收端为例）

在接收文件时，对数据包的序列号进行确认，如果出现了丢包，则会一直等待该包的序列号，对于后面数据包的序列号检测就会发生错误，则对发送端构造数据包发送消息表明自己需要的数据包序号，发送端就会知道自己哪个包传输错误，需要重传。如果数据包检测无误则写入，并回复 ACK。

```

//接收文件
int recvfile(SOCKET& socket, SOCKADDR_IN& addr, char* data)
{
    int addrlen = sizeof(addr);
    long int sum = 0; //文件长度，要返回的数据
    Head head;
    char* buf = new char[maxsize + sizeof(head)]; //缓冲数组长度是数据+头部的最大大小
    int seq = 0; //期待序列号清 0
    int num = 0; //数据包编号
    while (1)
    { //接收数据包
        int recvlenght = recvfrom(socket, buf, sizeof(head) + maxsize, 0, (sockaddr*)&addr,

```



```

&addrlen); //接收报文长度
    memcpy(&head, buf, sizeof(head));
    if (head.flag == END && check((u_short*)&head, sizeof(head)) == 0) //END 标志位,
    校验和为0, 结束
    {
        cout << "【传输成功】" << endl;
        break; //结束跳出 while 循环
    }

    if (head.flag == unsigned char(0) && check((u_short*)buf, recvlength -
sizeof(head))) //校验和不为0 且 flag 是无符号字符
    {
        //判断收到的数据包是否正确
        if (seq != int(head.seq))
        { //seq 不相等, 数据包接收有误
            cout << "【接收错误, 丢包发生】" << endl;
            //重新发送
            head.flag = SYN; //标志位随便设一个非 ACK
            head.datasize = 0; //数据部分为0
            head.checksum = 0; //校验和设为0
            head.checksum = check((u_short*)&head, sizeof(head)); //重新计算校验和
            memcpy(buf, &head, sizeof(head)); //拷贝到数组
            sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlen);
            //重新发送 ACK
            cout << "-----重新发送第" << num - 1 << "个数据包的回复-----" << endl;
            cout << "【重新发送】标志位 = " << head.flag << " 序列号 = " << (int)head.seq
<< " 校验和 = " << int(head.checksum) << endl; //ACK 等于 seq
            continue; //丢包
        }
        //接收到数据包正确
        cout << "-----接收第" << num << "个数据包-----" << endl;
        cout << "【接收】标志位 = " << head.flag << " 序列号 = " << int(head.seq) <<
" 校验和 = " << int(head.checksum) << endl;
        char* bufdata = new char[recvlength - sizeof(head)]; //数组的大小是接收到的
        报文长度减去头部大小
        memcpy(bufdata, buf + sizeof(head), recvlength - sizeof(head)); //从头部后面
        开始拷贝, 把数据拷贝到缓冲数组
        memcpy(data + sum, bufdata, recvlength - sizeof(head));
        sum = sum + int(head.datasize);

        //初始化首部
        newbag2(head, ACK, buf, seq);
        //发送 ACK
        sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlen);
    }

```

```

        cout << "-----发送第" << num++ << "个数据包的回复-----" << endl;
        cout << "【发送】标志位 = " << head.flag << " 序列号 = " << (int)head.seq <<
" 校验和 = " << int(head.checksum) << endl;
        seq++; //序列号加
        seq %= 256; //超过 255 要取模
    }
}
//发送 END 信息，结束
newbag(head, END, buf);
if (sendto(socket, buf, sizeof(head), 0, (sockaddr*)&addr, addrlen) == SOCKET_ERROR)
{
    cout << "【发送错误】" << endl;
    return -1; //发送错误
}
return sum; //返回接收到的数据包字节总数
}

```

(6) 四次挥手

由于第二次挥手和第三次挥手可以重合在一次，因此代码只写了三次挥手。第一次挥手由发送端发起，组装数据包后进行发送。

```

//关闭连接 三次挥手
void fourbye(SOCKET& socket, SOCKADDR_IN& addr)
{
    int addrlen = sizeof(addr);
    Head head;
    char* buff = new char[sizeof(head)];
    //第一次挥手
    head.flag = FIN;
    //head.checksum = 0; //校验和置 0
    head.checksum = check((u_short*)&head, sizeof(head));
    memcpy(buff, &head, sizeof(head));
    if (sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, addrlen) ==
SOCKET_ERROR)
    {
        cout << "【第一次挥手失败】" << endl;
        return;
    }
    cout << "第一次挥手【FIN ACK】" << endl;
    clock_t byetime = clock(); //记录发送第一次挥手时间

    u_long mode = 1;
    ioctlsocket(socket, FIONBIO, &mode);
}

```

发送完数据包后需要利用 while 循环持续等待接收端发送的确认，从而判定是否需要超

时重传，以及进行校验和的检验判断数据包是否正确，如果正确则第二次挥手成功。

```
//第二次挥手
while (recvfrom(socket, buff, sizeof(head), 0, (sockaddr*)&addr, &addrlen) <= 0)
{
    //等待接收
    if (clock() - byetime > retime)//超时重传
    {
        memcpy(buff, &head, sizeof(head)); //将首部放入缓冲区
        sendto(socket, buff, sizeof(head), 0, (sockaddr*)&addr, addrlen);
        byetime = clock();
    }
}

//进行校验和检验
memcpy(&head, buff, sizeof(head));
if (head.flag == ACK && check((u_short*)&head, sizeof(head) == 0))
{
    cout << "第二次挥手【FIN ACK】" << endl;
}
else
{
    cout << "【第二次挥手失败】" << endl;
    return;
}
```

组装第三次挥手的数据包，如果发送成功则判定第三次挥手成功，此时不需要等待客户端的响应，直接断开连接即可。

```
//第三次挥手
head.flag = ACK_FIN;
head.checksum = check((u_short*)&head, sizeof(head)); //计算校验和
memcpy(buff, &head, sizeof(head));
if (sendto(socket, (char*)&head, sizeof(head), 0, (sockaddr*)&addr, addrlen) == -1)
{
    cout << "【第三次挥手失败】" << endl;
    return;
}

cout << "第三次挥手【ACK】" << endl;
cout << "【结束连接】" << endl;
cout << "-----" << endl;
}
```

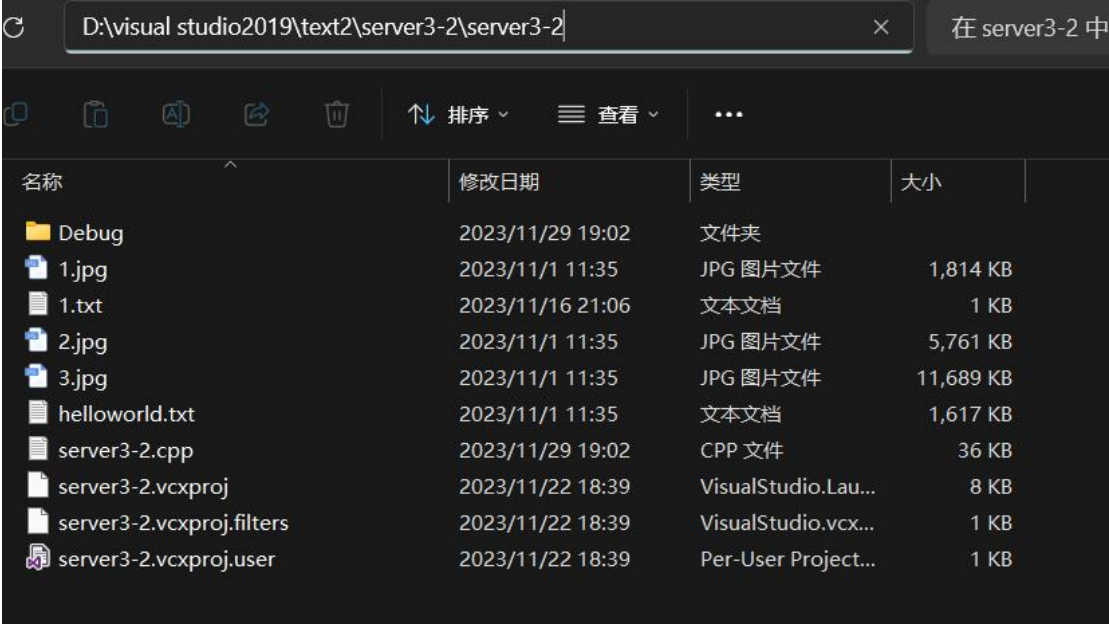
四、实验结果

1.运行截图

(1) 将路由器设置端口号与 IP 地址，并设置丢包率为 5%，延迟 10ms



(2) 将测试文件放置于发送端的程序目录下



(3) 发送 1.jpg,2.jpg,3.jpg,helloworld.txt 文件测试

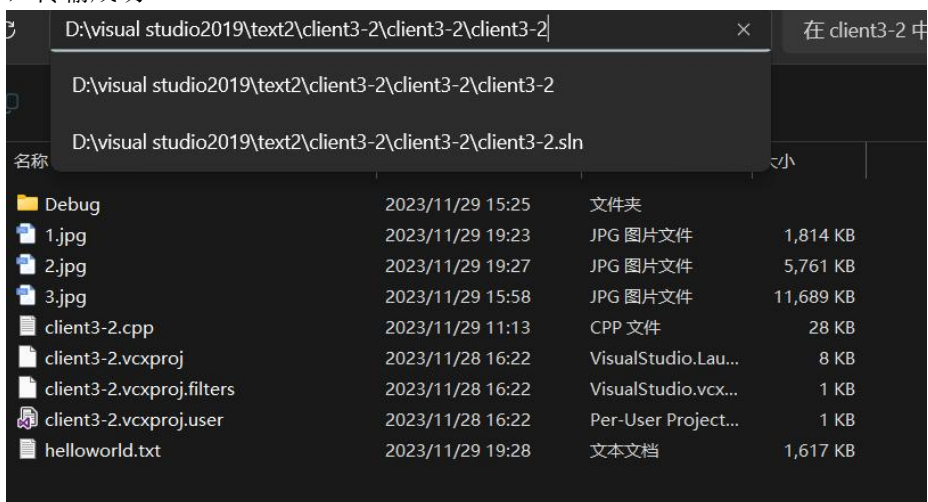
```
传输成功!
【文件名称】=1. jpg
【文件大小】=1813. 82KB
【传输总时间】=39618ms
【吞吐率】=46. 8815byte/ms
第一次挥手【FIN ACK】
第二次挥手【FIN ACK】
第三次挥手【ACK】
【结束连接】
```

```
传输成功!
【文件名称】=2. jpg
【文件大小】=5760. 26KB
【传输总时间】=168746ms
【吞吐率】=34. 9549byte/ms
第一次挥手【FIN ACK】
第二次挥手【FIN ACK】
第三次挥手【ACK】
【结束连接】
```

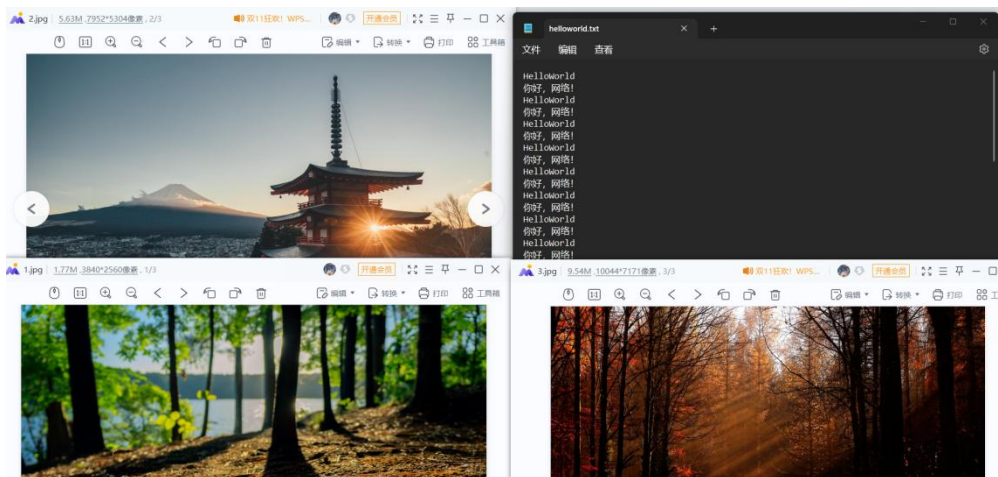
```
传输成功!
【文件名称】=3. jpg
【文件大小】=11688. 5KB
【传输总时间】=218335ms
【吞吐率】=54. 8194byte/ms
第一次挥手【FIN ACK】
第二次挥手【FIN ACK】
第三次挥手【ACK】
【结束连接】
```

```
传输成功!
【文件名称】=helloworld. txt
【文件大小】=1617KB
【传输总时间】=63122ms
【吞吐率】=26. 2319byte/ms
第一次挥手【FIN ACK】
第二次挥手【FIN ACK】
第三次挥手【ACK】
【结束连接】
```

发现四个文件都传输完成，并且文件大小和系统显示的都相同，经检测也都能正常打开文件，传输成功。

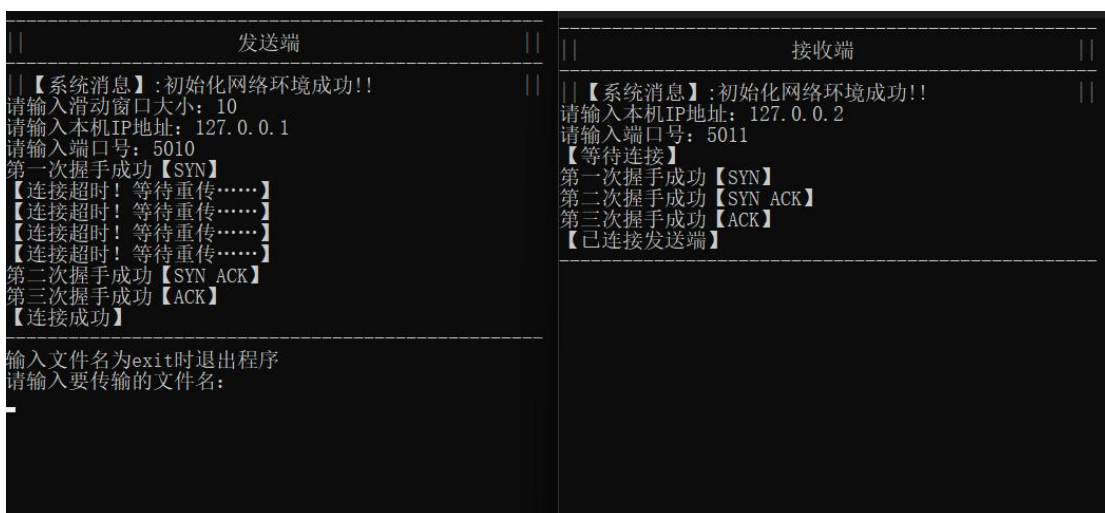


名称	日期/时间	类型	大小
Debug	2023/11/29 15:25	文件夹	
1.jpg	2023/11/29 19:23	JPG 图片文件	1,814 KB
2.jpg	2023/11/29 19:27	JPG 图片文件	5,761 KB
3.jpg	2023/11/29 15:58	JPG 图片文件	11,689 KB
client3-2.cpp	2023/11/29 11:13	CPP 文件	28 KB
client3-2.vcxproj	2023/11/28 16:22	VisualStudio.Lau...	8 KB
client3-2.vcxproj.filters	2023/11/28 16:22	VisualStudio.vcx...	1 KB
client3-2.vcxproj.user	2023/11/28 16:22	Per-User Project...	1 KB
helloworld.txt	2023/11/29 19:28	文本文档	1,617 KB



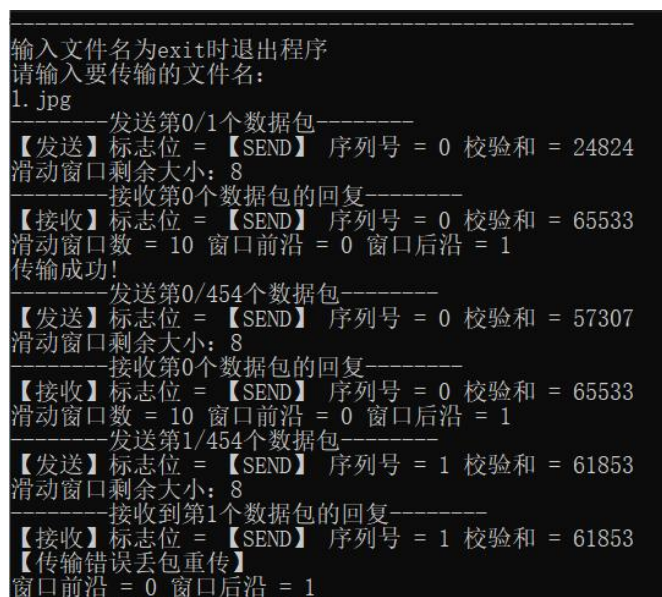
2.传输结果分析

- (1) 使用刚才设定的 router 程序来模拟丢包和延迟
- (2) 输入滑动窗口大小、对应的路由器 IP 和服务端 IP 等信息，实现连接



- (3) 传输测试文件 1.jpg

可以发现开始传输时会打印输出标志位、序列号、校验和、滑动窗口剩余大小等信息



如果产生丢包的情况，则会在发送端显示出传输错误丢包重传，接收端也会发现有接受错误丢包发生的情况出现，并且发送端就会重新发送上一个序号的数据包。

```
发送第0/454个数据包-----
【发送】标志位 = 【SEND】 序列号 = 0 校验和 = 57307
滑动窗口剩余大小: 8
接收第0个数据包的回复-----
【接收】标志位 = 【SEND】 序列号 = 0 校验和 = 65533
滑动窗口数 = 10 窗口前沿 = 0 窗口后沿 = 1
发送第1/454个数据包-----
【发送】标志位 = 【SEND】 序列号 = 1 校验和 = 61853
滑动窗口剩余大小: 8
接收第1个数据包的回复-----
【接收】标志位 = 【SEND】 序列号 = 1 校验和 = 61853
【传输错误丢包重传】
窗口前沿 = 0 窗口后沿 = 1
发送第1/454个数据包-----
【发送】标志位 = 【SEND】 序列号 = 1 校验和 = 61853
滑动窗口剩余大小: 8
接收第1个数据包的回复-----
【接收】标志位 = 【SEND】 序列号 = 1 校验和 = 65277
滑动窗口数 = 10 窗口前沿 = 1 窗口后沿 = 2
发送第2/454个数据包-----
【发送】标志位 = 【SEND】 序列号 = 2 校验和 = 23719
滑动窗口剩余大小: 8
接收第2个数据包的回复-----
【接收】标志位 = 【SEND】 序列号 = 1 校验和 = 65278
【传输错误丢包重传】
窗口前沿 = 1 窗口后沿 = 2
发送第2/454个数据包-----
【发送】标志位 = 【SEND】 序列号 = 2 校验和 = 23719
滑动窗口剩余大小: 8
接收第2个数据包的回复-----
发送第0个数据包的回复-----
【发送】标志位 = 【END】 序列号 = 0 校验和 = 65533
【传输成功】
接收第0个数据包-----
【接收】标志位 = 【RECEIVE】 序列号 = 0 校验和 = 57307
发送第0个数据包的回复-----
【发送】标志位 = 【END】 序列号 = 0 校验和 = 65533
接收第1个数据包-----
【接收】标志位 = 【RECEIVE】 序列号 = 1 校验和 = 61853
发送第1个数据包的回复-----
【发送】标志位 = 【END】 序列号 = 1 校验和 = 65277
【接收错误, 丢包发生】
重新发送第1个数据包的回复-----
【重新发送】标志位 = 【RESEND】 序列号 = 1 校验和 = 65278
接收第2个数据包-----
【接收】标志位 = 【RECEIVE】 序列号 = 2 校验和 = 23719
发送第2个数据包的回复-----
【发送】标志位 = 【END】 序列号 = 2 校验和 = 65021
接收第3个数据包-----
【接收】标志位 = 【RECEIVE】 序列号 = 3 校验和 = 40525
发送第3个数据包的回复-----
【发送】标志位 = 【END】 序列号 = 3 校验和 = 64765
【接收错误, 丢包发生】
重新发送第3个数据包的回复-----
【重新发送】标志位 = 【RESEND】 序列号 = 3 校验和 = 64766
接收第4个数据包-----
【接收】标志位 = 【RECEIVE】 序列号 = 4 校验和 = 50251
发送第4个数据包的回复-----
【发送】标志位 = 【END】 序列号 = 4 校验和 = 64509
【接收错误, 丢包发生】
```

传输成功后会打印文件名称、文件大小、传输总时间、吞吐率等信息，由于发送端和接收端的计时节点不同，所以也会导致出现一些细微的差异。

```
【接收】标志位 = 【SEND】 序列号 = 454 校验和 = 64509
滑动窗口数 = 10 窗口前沿 = 454 窗口后沿 = 454
传输成功!
【文件名称】=1. jpg
【文件大小】=1813. 82KB
【传输总时间】=49453ms
【吞吐率】=37. 5579byte/ms
第一次挥手 【FIN ACK】
第二次挥手 【FIN ACK】
第三次挥手 【ACK】
【结束连接】
【重新发送】标志位 = 【RESEND】 序列号 = 3 校验和 = 64766
【传输成功】
第一次挥手成功 【FIN ACK】
第二次挥手成功 【FIN ACK】
第三次挥手成功 【ACK】
【结束连接】
【文件传输成功】
【文件名称】=1. jpg
【文件大小】=1813. 82KB
【传输总时间】=49452ms
【吞吐率】=37. 5587byte/ms
```

五、心得体会

通过此次实验，学习到了 UDP 连接等相关的知识，对于建立连接、差错检测、累计确认、超时重传、Go Back N 等知识有了深入的了解，巩固了所学的知识。

也发现了许多问题，例如网络延迟较高时或者丢包率较大时发送效率就会显著下降，卡顿明显，以及滑动窗口过大时也会出现问题，可能是代码还有待完善，还要继续学习。

同时发送窗口应该设置一个区域范围为 4-32，不能随意输入，过大会导致出现传送卡顿的现象，例如为 10000 时程序会卡死。

六、附录

完整代码参照 GitHub:

<https://github.com/Q-qiuqiu/Computer-Networks/tree/main/lab3-2>