实验三：基于 UDP 服务设计可靠传输协议并编程实现

- 姓名 : 马永田
- 年级： 2020 级
- 专业 : 计算机科学与技术
- 指导教师 : 张建忠 & 徐敬东

实验要求

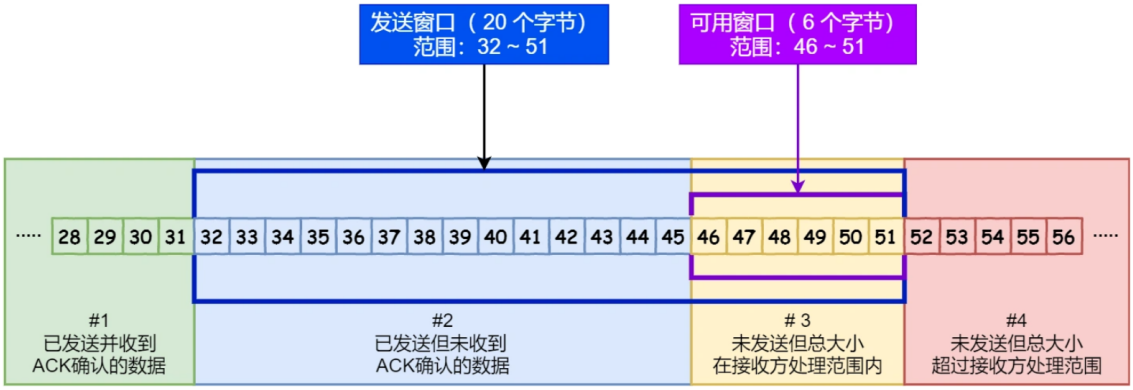
在实验3-1的基础上，将停等机制改成**基于滑动窗口的流量控制机制**，采用固定窗口大小，支持**累积确认**，完成给定测试文件的传输

1. 多个序列号
2. 发送缓冲区、接受缓冲区
3. 滑动窗口：Go Back N
4. 有必要日志输出（须显示传输过程中发送端、接收端的窗口具体情况）

GBN协议设计

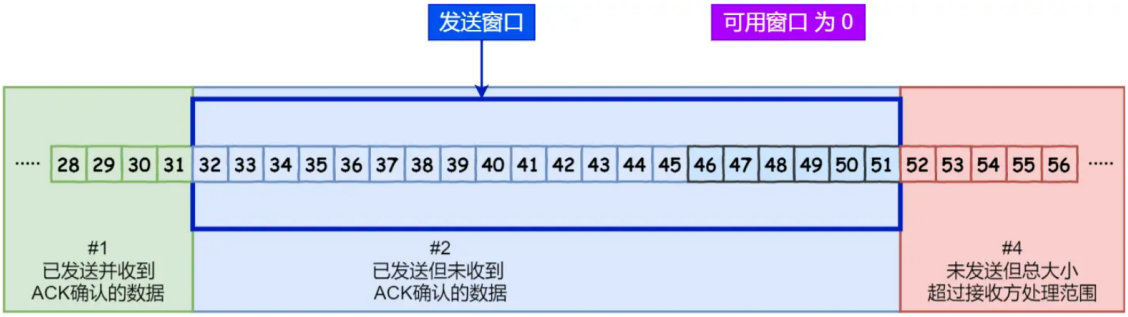
流程分析

对于**发送方的窗口**，如下图为发送方缓存的数据，根据处理的情况分成四个部分，其中深蓝色方框是发送窗口，紫色方框是可用窗口：



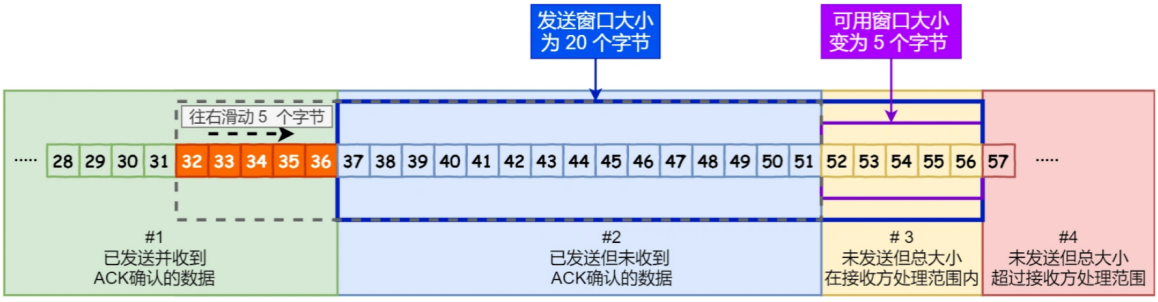
- #1 是已发送并收到 ACK确认的数据：1~31
- #2 是已发送但未收到 ACK确认的数据：32~45
- #3 是未发送但总大小在接收方处理范围内（接收方还有空间）：46~51
- #4 是未发送但总大小超过接收方处理范围（接收方没有空间）：52之后

如下图，当发送方把数据全部都一下发送出去后，可用窗口的大变小为 0 ，表明可用窗口耗尽，在未收到 ACK 确认之前无法继续发送数据。



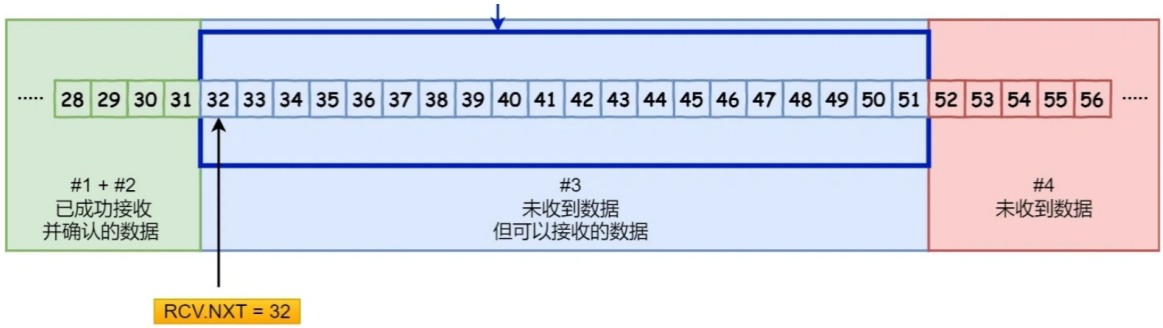
如下图，当收到之前发送的 `32~36` 分组的 ACK 确认应答后，**滑动窗口往右边移动 5 ，因为有 5 个分组被应答确认**，接下来 `52~56` 又变成了可用窗口，继续发送 `52~56` 这 5 个分组。

其中：由于接收端的累积确认能力是固定的，因此当被确认的间隔变小时，我们会认定接受方出现了问题，需要重传数据包，因此若此时被确认的 `32~36` 分组的大小相比上一次更小，发送方并不会像上述流程中一样继续发送，而是直接从 `36` 处重传当前窗口。



而对于接收方，实际接收窗口为1，由于设置了固定的累积确认大小， 因此也可看做是一个累积窗口，根据处理的情况划分成三个部分：

- #1 + #2 是已成功接收并确认的数据（等待应用进程读取）；
- #3 是未收到数据但可以继续接收并累积的数据；
- #4 未收到数据并处于下一次累积的数据；



通过一个确认指针(图中为32)，接收方能够只按序接收数据，当确认指针到达累积窗口末端时(图中为51)，发送ACK确认累积窗口中的所有数据，而当收到乱序的数据时：

1. 若该数据处于#1+#2部分，即小于确认指针，则表示已正确接收，直接丢弃
2. 若该数据大于当前的确认指针，提前接收到了后续的包，即发生了丢包事件(也可能只是来慢了)，立即停止累积直接发送ACK确认

发送方客户端

缓冲区写入

使用如下的readFile函数读取文件并写入到缓冲区中，其中设置了sendSeq作为发送序列号，每读入一个数据包该变量就会加一，其中最后一个数据包设置**EF位**表示文件传输结束，

```
bool readFile(char* fileName) {
    string filename(fileName);
    ifstream in(filename, ifstream::binary);
    bufIndex = sendSeq + 1;
    PacketNum = 0;
    data_p = 0;
    Packet sendBuf;
    char Char = in.get();   //读入字符char
    while (in) {
        sendBuf.Data[data_p] = Char;    //文件内容存入sendBuf中
        data_p++;
        if (data_p % BUF_SIZE == 0) {   //到达最大容量 下一个数据包
            sendBuf.length = data_p;
            sendBuf.Seq = bufIndex;
            setFile(&sendBuf);
            memcpy(buf[bufIndex], &sendBuf, sizeof(Packet));
            bufIndex++;
            PacketNum++;
            data_p = 0;
        }
        Char = in.get();
    }
    sendBuf.length = data_p;
    sendBuf.Seq = bufIndex;
    setFile(&sendBuf);
    setEnd(&sendBuf);
    memcpy(buf[bufIndex++], &sendBuf, sizeof(Packet));
    in.close();
    return 1;
}
```

文件发送

文件发送使用如下的sendFile函数实现：

```
int sendFile(char* fileName) {
    readFile(fileName)) {
//打包文件名
    int nameLength = strlen(fileName);
    Packet sendFileName, recvACK;
    setStart(&sendFileName);
    sendFileName.Seq = sendSeq++;
    sendFileName.index = PacketNum;
    sendFileName.length = nameLength;
    for (int i = 0; i < nameLength; i++) {
        sendFileName.Data[i] = fileName[i];
```

```
    }
//发送SF包并等待ACK
    sendPacket(&sendFileName);
    clock_t start = clock();
    clock_t end;
    int times = 0
    while (1) {
        if (recvPacket(&recvACK)) {
            if (checkACK(&recvACK) && recvACK.Ack == sendSeq) {
                break;
            }
        }
        end = clock();
        if(times >= 10){
            return 0
        }
        if (end - start > 500) {
            times ++;
            sendPacket(&sendFileName);
        }
    }
//开始传输文件
    clock_t RPS_s = clock();
    wndSlide();
    clock_t RPS_d = clock();
    //吞吐率计算
    double totalTime = (double)(RPS_d - RPS_s) / CLOCKS_PER_SEC;
    double RPS = (double)(PacketNum) * sizeof(Packet) * 8 / totalTime / 1024
/ 1024;
}
```

1. 该部分首先调用readFile函数将文件读取到缓冲区中
2. 之后将文件名打包并设置**SF位**后发送，表示开始传输文件
3. SF包发送后设置**计时器**等待接收方的**ACK确认**
4. 若超时则重发SF包，**重发10次**后放弃发送文件
5. 若设置时间内收到ACK确认，调用**wndSlide**函数开始发送文件

缓冲区发送

缓冲区的发送使用如下的sendBuf函数实现，其中**wndStart**表示**窗口起始**位置，**i**表示最后一个**已发送但未确认**序号，**wndEnd**表示发送**窗口结束**位置，该部分的逻辑较为简单，即通过**bufStop**和**retrans**标志来控制缓冲区的发送与重传：

1. bufStop为真时跳出循环，停止缓冲区的发送，否则停留在while循环中继续发送缓冲区
2. retrans为真时重传窗口内容，从最后一个**已发送且被确认**序号开始发送到**发送窗口末端**
3. 而retrans未假时则继续发送缓冲区内容，从最后一个**已发送但未确认**序号开始发送到**发送窗口末端**

```
void sendBuf() {
    int i = wndEnd;
    int end = wndEnd;
    while (!bufStop) {
        if (i >= bufIndex) {
```

```
            continue;
        }
        if (!retrans) {
            if(i < wndEnd) {
                sendPacket((Packet*)buf[i]);
                i++;
            }
        }
        else {
            retrans = false;
            end = wndEnd;
            for (i = wndStart; i < end; i++) {
                sendPacket((Packet*)buf[i]);
            }
        }
    }
    return;
}
```

窗口滑动

窗口的控制部分使用如下的wndSlide函数实现：

```
void wndSlide() {
    bufStop = false;
    wndStart = sendSeq; //设置窗口从序列号开始
    wndEnd = wndStart + WND_SIZE;
    thread sendThread(sendBuf); //启动Buf发送线程
    Packet recvP;
    int AckSize = 0;
    wndPointer = wndStart + 1;  //指向已确认序列号
    clockStart = clock();
    retrans = true;
    while (true) {
        if (recvPacket(&recvP)) {
            if (checkACK(&recvP) && recvP.Ack > wndStart) { //ACK确认号大于当前窗口起始
                if (recvP.Ack >= bufIndex) {    //若缓冲区发送完毕，停止发送跳出循环
                    bufStop = true;
                    sendThread.join();
                    return;
                }
                int nowSize = recvP.Ack - wndStart;
                sendSeq = recvP.Ack;    //发送成功 sendSeq序列号更新
                wndStart = recvP.Ack;   //窗口滑动
                wndEnd = (wndStart + WND_SIZE) >= bufIndex ? bufIndex : (wndStart + WND_SIZE);
                if (nowSize >= AckSize) { //累积确认大小变化
                    AckSize = nowSize;
                }
                else {      //若此次确认的长度比上次更小 说明对方没有正确接收到本组全部帧
                    retrans = true;//直接重传全部分组
```

```
                }
                clockStart = clock(); //成功收到ACK，重置计时器
                continue;
            }
        }
        clockEnd = clock(); //若超时 重传
        if (clockEnd - clockStart > 500) {
            retrans = true;
            clockStart = clock();
        }
    }
}
```

由于本次实验的窗口采用的是固定大小，因此窗口的控制逻辑也较为简单：

1. 首先该函数会对窗口起始wndStart、缓冲区发送标志bufStop等**变量进行设置**，并启动**sendBuf线程**持续发送缓冲区。
2. 之后进入while循环不断接收对方发来的ACK确认，
3. 若收到的确认**序列号Ack小于等于窗口起始**，说明接收方可能乱序接收到了，或是回复的多个ACK乱序到达发送方，是之前**已经被确认**过的序列号，因此无需考虑，直接**舍弃**。
4. 若确认**序列号Ack大于窗口起始**，表示有新的**已发送但未被确认**的帧被确认了，更新wndStart，窗口进行滑动。

利用接收端确认间隔大小的变化实现快速重传：

其中对于收到的Ack序列号，由于接收方进行累积确认时该累积大小也是固定的，因此若收到的确认序列号Ack间隔变小，说明接收方的接收出现了问题，例如可能是乱序接收或者是未接收到，导致提前发送ACK确认。若是对这种情况不进行区分，按照之前SendBuf函数的逻辑，文件传输的性能就会收到一定的影响，举一个简单的例子：

假设接收方每累积20个正确的帧就回复ACK，我们现在发送方的窗口序列号为10 ~ 50，这样的话接sendBuf线程会将其中内容全部发送直到窗口末端后停下，此时：

1. 若是接受方正常接收且回复Ack=30，则发送方将窗口滑动到30 ~ 70，sendBuf继续发送50 ~ 70直到窗口末端后停下，正常运行。
2. 但若是接收方错误接收，例如28号帧丢失，则接受方回复Ack=28，发送方收到该确认ACK后窗口滑动到28 ~ 68，之后sendBuf就会继续发送50 ~ 68的数据帧。而接收方的视角则是由于28号帧缺失，从29号帧开始收到的均为乱序的，因此从29开始到68都会舍弃，这样一来后续又额外发送的50 ~ 68号数据帧都是无意义的。且直到超时重传之前，**接收方都无法收到正确的帧**，发送方也就无法收到新的ACK，也就**无法滑动窗口。**

也就是说，实际上发送方从29号帧起发送的帧均是无效的，这样看来最高效的选择应当是：当发送方意识到这件事时就应该立即停止发送后面的 **在窗口内但尚未发送** 的帧，并立即从28号帧开始重传。

因此在此处我通过添加了一个变量来记录发送方的**累积确认间隔**，通过其间隔的变化来简单的判断**是否需要立即重传**，而此处所添加的AckSize在一定程度上也体现了**接收方接收能力的变化。**

接收方服务端

接收缓冲区与累积确认

该部分内容通过如下的recvFile函数实现：

```
int recvFile(Packet recvFileName) {
    Packet sendACK;
```

```
    sendACK.Seq = sendSeq;
    setACK(&sendACK, &recvFileName);
    sendPacket(&sendACK);
    memset(&sendACK, 0, sizeof(Packet));
    //获取文件名
    int PacketNum = recvFileName.index;
    int nameLength = recvFileName.length;
    char* fileName = new char[nameLength + 1];
    memset(fileName, 0, nameLength + 1);
    for (int i = 0; i < nameLength; i++) {
        fileName[i] = recvFileName.Data[i];
    }
    fileName[nameLength] = '\0';
    Packet recvP;

    wndStart = recvFileName.Seq + 1;
    wndEnd = wndStart + WND_SIZE;
    wndPointer = wndStart;//指向第一个等待到来并进行确认的
    bool flag = true;
    clockStart = clock();
    while (1) {
        if (recvPacket(&recvP) && checkFile(&recvP)) {
            if (recvP.Seq == wndPointer) {
                orderFlag = true;
                memcpy(buf[wndPointer++], &recvP, sizeof(Packet));
                if (checkEnd(&recvP)) {
                    wndStart = wndPointer;
                    wndEnd = wndPointer;
                    setACK(&sendACK, &recvP);
                    sendPacket(&sendACK);
                    break;
                }
                if (wndPointer == wndEnd) { //累积满
                    wndStart = wndEnd;
                    wndEnd = wndStart + WND_SIZE;   //滑动新窗口
                    setACK(&sendACK, &recvP);
                    /* clock_t a = clock();
                    clock_t b = clock();
                    while ((b - a) < 5) {
                    b = clock();
                    } */
                    sendPacket(&sendACK);   //确认旧窗口
                }
            }
            else if (recvP.Seq > wndPointer) { //乱序 超前接收  应当接收的来晚了
                if (orderFlag) {
                    orderFlag = false;
                    wndStart = wndPointer;
                    wndEnd = wndStart + WND_SIZE;   //滑动新窗口
                    setACK(&sendACK, wndPointer);
                    sendPacket(&sendACK);   //重新发送 已确认缓冲区中最后一个
                }
            }
            else {
                //do nothing 已确认过 丢弃
```

```
            }
        }
    }
    outFile(fileName, PacketNum, recvFileName.Seq + 1);
    return 1;
}
```

该部分的流程如下：

1. 接收方接收到**SF包**后解析出其中的文件名，并**同步**发送方的发送序列号
2. 进入while循环，使用一个变量**wndPointer**来记录成功接收到的帧号，以此实现**按序接收数据帧。**
3. 若接收到的数据帧的序列号**等于wndPointer**，则代表是**按序接收**，将其写**入接收缓冲区**并让wndPinter加一
    1. 当收到**EF包**时表示文件传输结束，回复ACK确认并跳出循环。
    2. 当wndPointer累积增加了一定大小后，发送一个ACK确认之前按序正确收到的分组，并请求新的分组，实现**累积确认。**
4. 若接收到的数据帧的序列号**小于wndPointer**，则代表该帧**已被确认**，直接**丢弃**
5. 若接收到的数据帧的序列号**大于wndPointer**，则代表产生了**分组乱序或是丢失**的情况，按照GBN协议，我们应该在此处回复一个ACK，即重新开始累积确认。

但实际上当发生丢包时，从**丢包处开始**发送方已经发送的所有分组(直到重传之前)，**均会被接收方判定为乱序**，这样的话接收方就会重复发送很**多个ACK**，这样是没有意义的，也会同时占用发送方和接收方的资源，因此在此处我添加了一个**orderFlag变量**来标志是否为乱序：

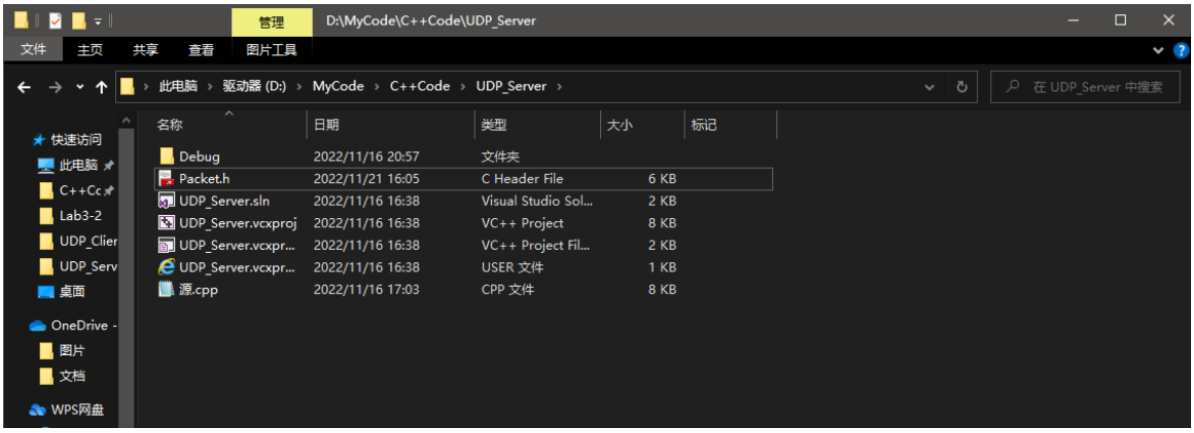
1. 当按序接收时会将该变量**置为true**，表示**有序接收**
2. 当且仅当**发生乱序且该变量为真**时，接收方才会回复一个ACK确认，重新开始累积，并将其**置为false**
3. 之后由于该变量为false，因此接收方会**忽略后面的乱序分组**
4. 直到下一次**正确接收到有序分组**时，该变量**重新为true**

这样一来，遇到丢包所造成的后续多个分组均为乱序的情况时，接收方只会在收到第一个乱序分组时才会回复ACK确认，后面的乱序分组均会被忽略掉。

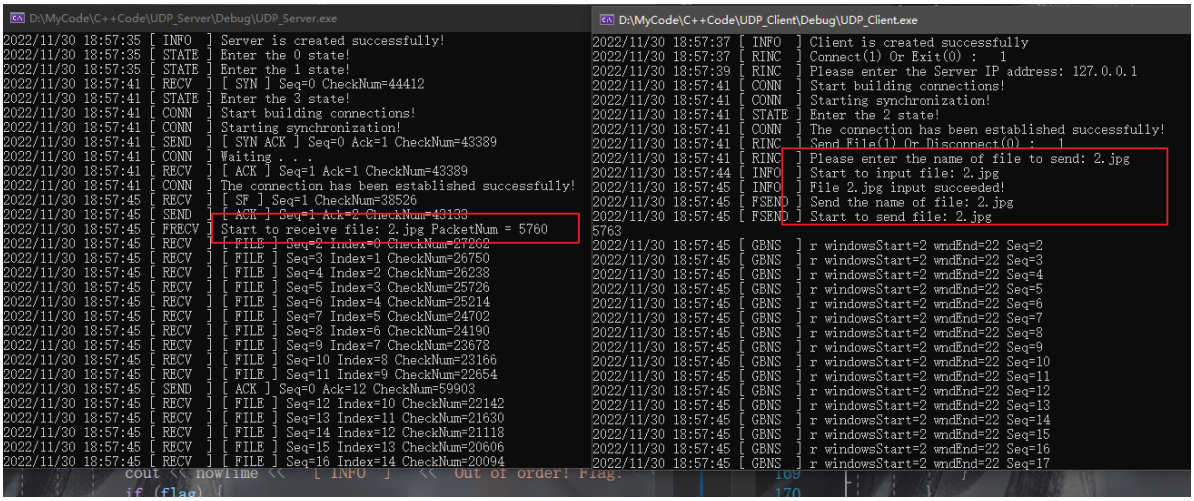
程序执行演示

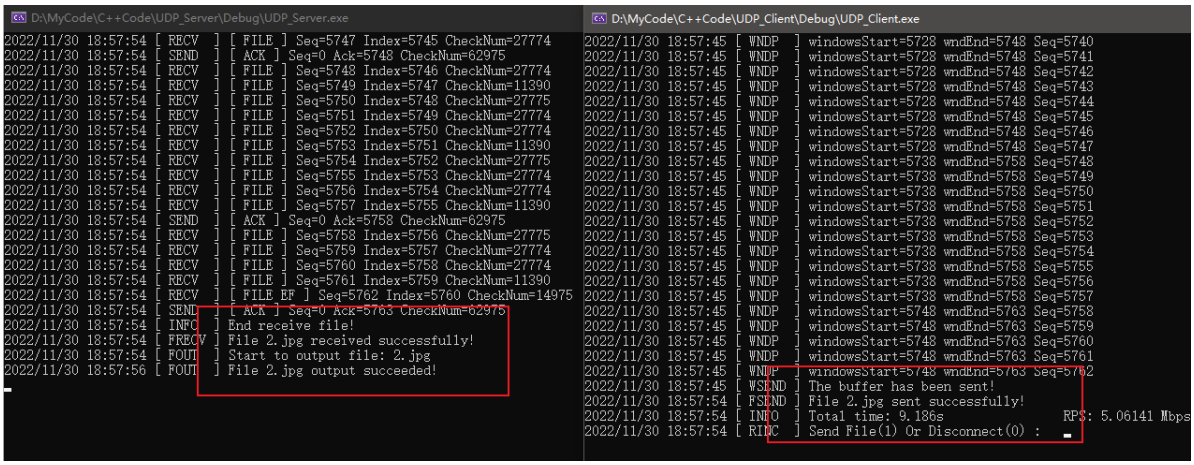
测试文件传输

首先将测试文件移到客户端目录下，并确保服务端目录下无其他文件：

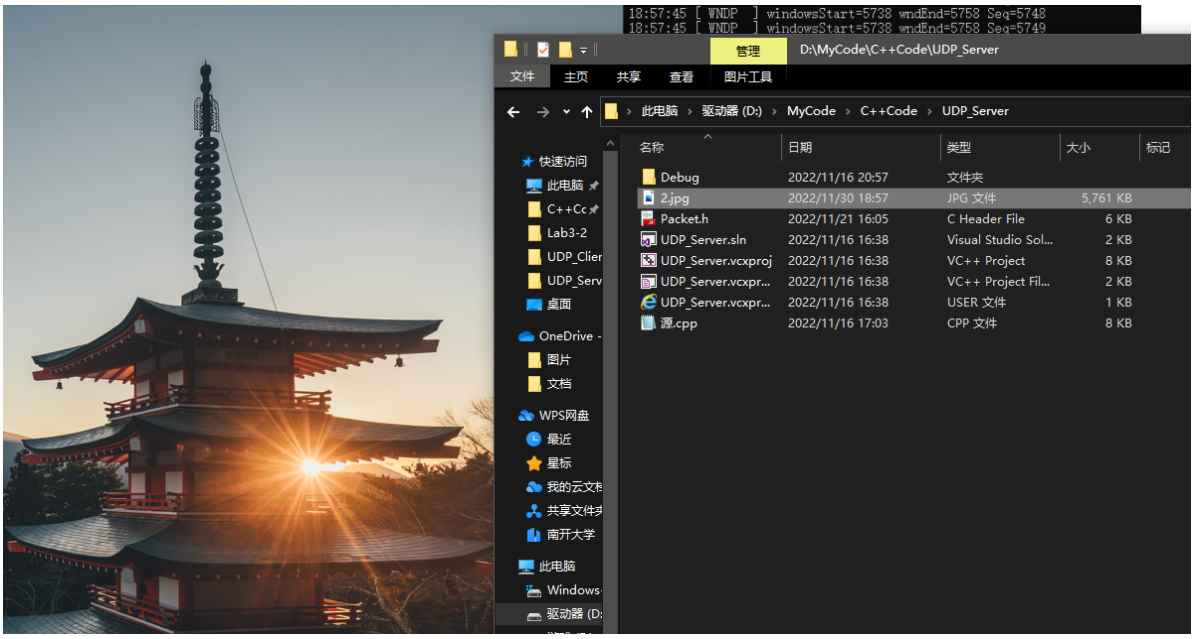


之后打开客户端与服务端开始传输测试文件 2.jpg：





如下图可见对应目录下出现了刚刚传输的图片2.jpg，且能够正常的打开，说明文件成功传输，其中由于I/O占用较长时间，因此吞吐率RPS较低。



分析窗口滑动过程

测试中设置发送方固定窗口大小为20，接收方累积10帧回复一个ACK确认。通过选择性丢包与接收端延迟发送ACK来检验分析窗口的滑动过程：

接收方延时设置

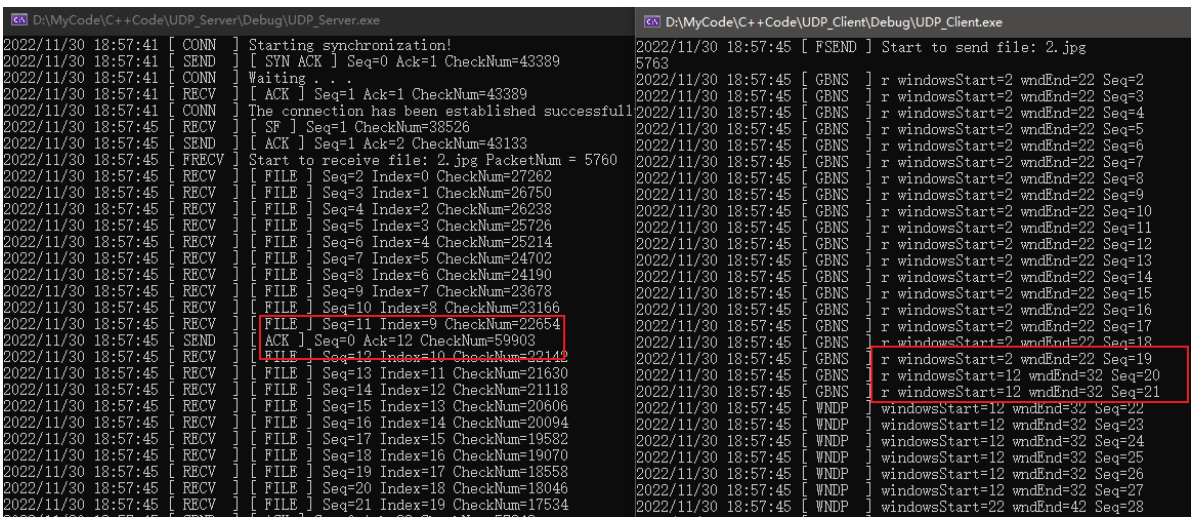
```
if (wndPointer == wndEnd) { //累积满
    wndStart = wndEnd;
    wndEnd = wndStart + WND_SIZE;    //滑动新窗口
    setACK(&sendACK, &recvP);
    clock_t a = clock();
    clock_t b = clock();
    while ((b - a) < 5) {
        b = clock();
    }
    sendPacket(&sendACK);    //确认旧窗口
}
//设置ACK延时5ms发送
```

发送方选择性丢包

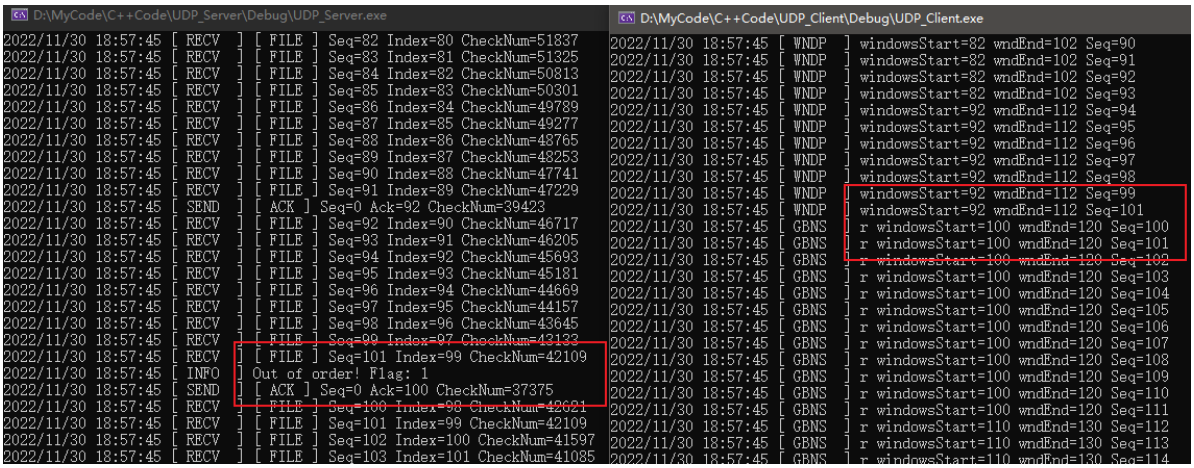
```
int discard = 100;

if(i < wndEnd) {
    if (i == discard) {
        discard = 498;
        i++;
        continue;
    }
    sendPacket((Packet*)buf[i]);
    i++;
}
// 设置最开始不发送100和498号帧
```

窗口滑动过程分析

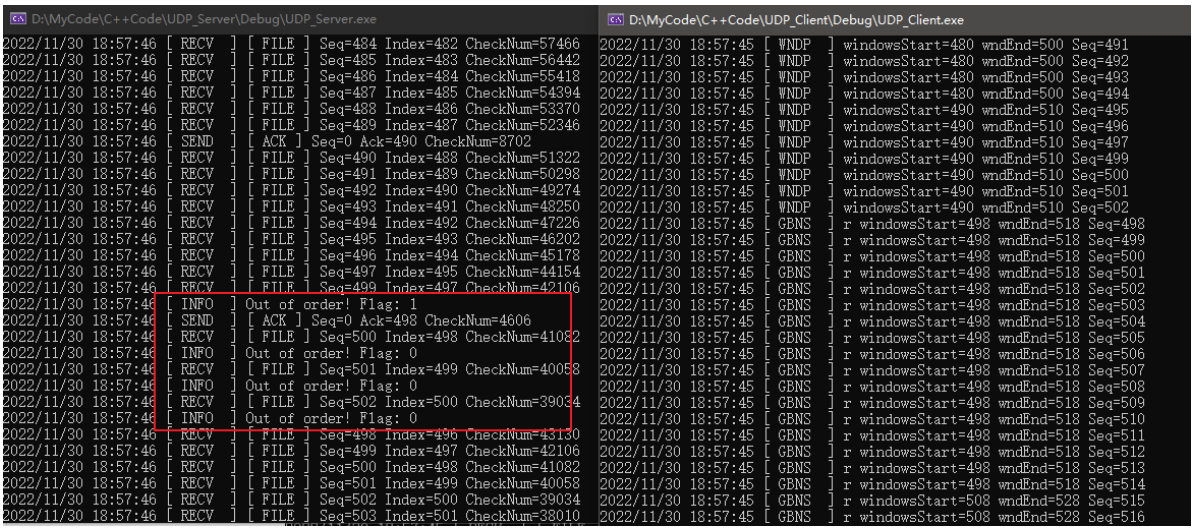
如下图可见，左侧接收方收到了Seq = 11的分组后回复了一个ACK确认，右侧的发送方最初的窗口为2 ~ 22当其发送完Seq = 11的帧后没有立刻收到ACK确认，而是在发送Seq = 19之后，收到了ACK确认，之后窗口滑动。注意此时实际上发送方最开始的窗口还未发送完毕，窗口就向后滑动了，通过设置这样一个接收端回复延时可以看出，发送方的缓冲区发送与分组确认并滑动窗口是流水线模式执行的：即便未收到ACK也会继续发送后面的数据帧，当收到ACK后立刻将窗口进行滑动而不是等待当前窗口发送完毕。



如下图可见，当发送方第一次发送到Seq = 100的帧时会直接跳过不发送，继续发送窗口内的未发送的数据，而接收方在收到乱序的分组后会提前结束累积并回复一个确认ACK，发送方收到该ACK后得知接收方接受能力变小，说明接收方出了问题，不会再继续发送当前窗口中未发送的内容，而是直接Go Back N，从丢失处开始立即重传，而发送方收到序号正确的分组后也会继续累计。



当遇到由于缺失造成后续分组均为乱序无法接受的情况时，为了避免浪费资源，前文中提到通过设置一个标志变量来实现只在第一次乱序时发送确认ACK，后续的连续乱序均会进行忽略，如下可见当收到Seq = 497后又收到了Seq = 499时，即Seq = 498的分组丢失时，接收方会发送一个ACK，而之后的Seq = 500的乱序分组到达时并不会再重复发送ACK。



程序测试小结

虽然通过设置了多个标志变量来区分各种情况，但实际上协议中重传的设计仍有很多的漏洞，例如如果是使用路由程序中固定每隔N个包就丢失一次的丢包模式，若N太小，可能就会导致接收方虽然乱序接收，但确认间隔可能会不变或持续变小，如当N为5而累积大小为10时，每五个包就会丢失一次，接收方的累积大小就会一直是5 ，这种情况下只能等待发送方的超时重传，因此接收方与发送方均会有计时器，发生超时会进行重传，牺牲性能来确保能够正确的发送文件。相反，若是丢包率较低且随机的情况下，该方案的性能还是比较好的，发送方只需接收方的累积能力就可以立即判断是否重传。此外实验中还遇到了一种更糟糕的情况：由于丢包率与窗口大小设置的过于“巧妙”，发送方从丢失数据包处开始，到接收到间隔变小的ACK并启动重传的这段时间内，由于接收方的回复有延迟，且发送方的速度较快，发送方会继续发送多个分组，而路由程序在收到后又会进行累计，这时就可能会出现恰好重传时丢包的情况，这样程序就会堵在这里，无法继续发送后续的分组。

实验中也考虑了**另一种方案：**当接收方收到乱序分组时会持续发送最后一个正确确认的ACK，而发送方收到多个相同ACK后立即停止发送可用窗口，并从出问题(即重复收到的ACK的Ack号)处进行重传，但在实际实现中发现仍旧会有一些漏洞，例如可能被丢弃的包为发送方发送的最后倒数第二个包， 这样接收方只会收到一个乱序分组，也就只会回复一个ACK，只能等待超时重传；相反，若乱序分组过多，发送

方就会收到很多个相同的ACK，就会重传多次，若要解决这种问题，就需要为连续接收两个相同ACK后重发的情况添加一些限制，例如在一定时间内不会重发第二次等。并且，在实际场景中固定间隔连续丢包的概率应该是极小的，方案一即可满足需求，因此综合考究之后并没有选择该方案。