南開大學
Nankai University

计算机学院

计算机网络实验报告

实验 3-2: 基于 UDP 服务设计可靠传输协议并编程实现

姓名：谢畅

学号：2113665

专业：计算机科学与技术

2023 年 12 月 1 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 >1，接受窗口 =1，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
- 累积确认: 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)

2 协议设计

这里协议类型采取 UDP，在 udp 基础上设计自己的协议。

2.1 数据包格式

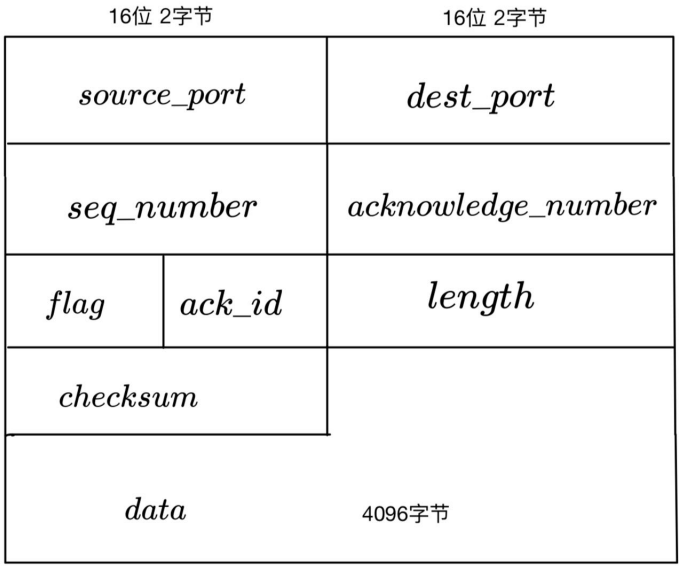

图 2.1: 自定义数据包格式 Mymsg 类型

上图2.1是我们设计的数据包格式，其中延续实验 3-1 中设计的数据包，以及数据包的相关处理函数，比如 *send_generate* 函数等等。这里仅针对 3-2 的改动进行数据包的说明。

- 这里的 seq_number 代表序列号，其中用 16 位进行表示。那么如果发送分组超过 65535，就会回到序列号 0。

  因此，可以看出实际上 $Window\_size \leq 2^n - 1$ 对于我们的测试文件而言，肯定会成立, 因为实验中设定的窗口大小肯定比 65535 要小。

  对于接收方而言，获取到的包，需要提取其中的 seq_number，与自己的 expectedseqnum 进行比较，相等才会接受包，否则拒绝这个数据包。因此，在 $Mymsg\_explain$ 类中新增 $hasseqnum$ 的成员函数，用于在接收端解析一条数据包。

- acknowledge_number 是确认号，发送者接受 ack 包，提取其中的 acknowledge_number，然后进行发送窗口的后沿移动。

- 最后值得一提的是，数据包的固定长度由 MSS 决定。可能数据包不足一个 MSS，但是数组是固定大小 MSS 的，这里数据包的长度由 length 字段决定

2.2 GBN 实现

图 2.2: GBN 发送窗口图

- **滑动窗口**：GBN 使用滑动窗口来控制发送方和接收方之间的数据传输。发送方和接收方都有一个窗口，用于跟踪传输的数据包。

  发送方窗口控制了可以连续发送但尚未被确认的数据包数量，以及对应的序号 seq。允许发送 N 个未被确认的分组。同时我们用 base 标识最小的已发送未被确认的分组，用 nextseqnum 标识最小的还未被发送的分组。只要窗口未满，我们便可以一直传输未被发送的分组.

  但是我们在发送方设置了一个定时器，用于定时 base 指向的数据包。如果此数据包超过 $TIMEOUT$ 仍然没有被确认，就需要超时重发。如果此时当前窗口内所有已发送分组都得到了确认，我们便关闭定时器；如果仍然存在已发送但是没有得到确认的分组，我们便需要重新开启计时器。

  接收方窗口大小为 1，用于确定下次应该接受的数据包 seq 序号。

- **序号**：数据包被赋予序号，以便发送方和接收方可以对数据包进行识别和排序。序号用于确定数据包的顺序，并帮助接收方检测并处理重复数据包。

- **Go-Back-N 策略**：如果发生丢失或损坏，发送方将重传从未收到确认的第一个数据包开始的所有数据包。这意味着即使只有一个数据包丢失，后续数据包也会被全部重新发送。

2.2.1 发送方

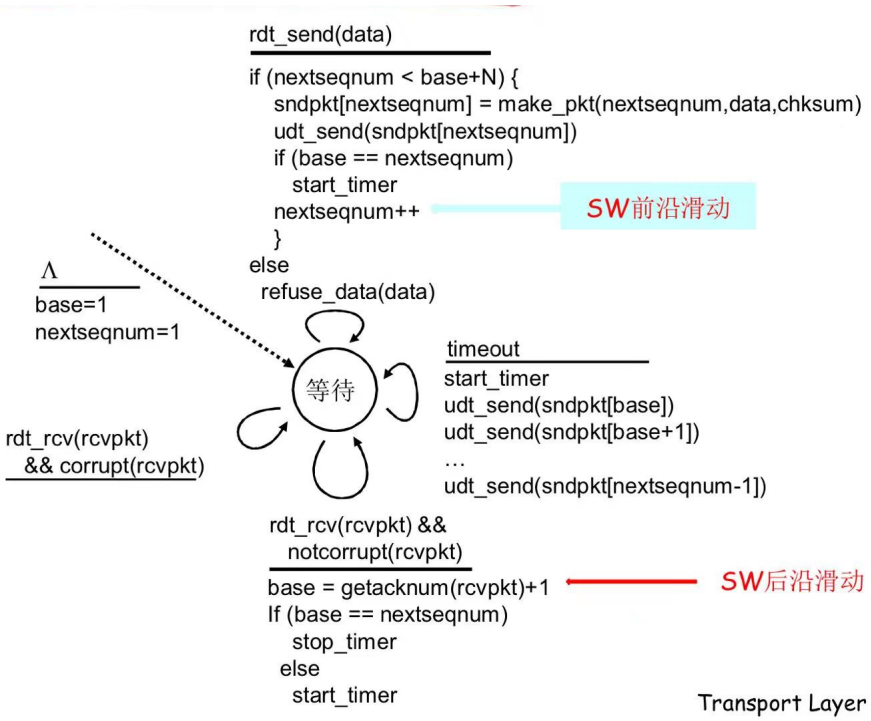

图 2.3: GBN: 发送方扩展的 FSM

根据图2.3，我们设计了本次实验的发送方的 GBN 流水线机制。

- 可以看到这里设计了两个变量 base，nextseqnum。这里是发送窗口的前沿以及后沿。

  base 与 nextseqnum 相当于发送窗口的指针。在发送多个包时，前沿不断移动，代表发送但未确认的数据包增加。在接受 ack 时，后延移动，代表之前的包已被确认，不在发送窗口/发送缓冲区范围之内。

- 其中还需要设计一个队列 udt_send，保存发送窗口的数据包。

  在我们发送数据包时，将产生的数据包压入队列。在接受 ack 时，将最老的数据包给删除。这样重发的时候，可以保证队列里面是发送窗口范畴内的数据包。

- 由于我们的序列号实际上是有限制的，16 位。所以这里需要一个额外的变量 seq_len，它不会有周期，而是记录分组发送的个数，用一个较多位数的变量表示即可，比如 32 位。

  不过在本次实验中，测试文件的分组个数远远小于 16 位能表示的最大序列号，其实这个问题不必考虑，不过为了完善性，这里还是纳入考虑范畴。

- 关于**多线程考虑**

  在上面的状态机中，我们可以发现一共有三个主要动作: 发送多个数据包、超时重发、接受 ack。

  这里可以用主线程发送多个数据包，产生 2 个子线程分别进行超时重传、接受 ack。然后我们接受 ack 的子线程可以设置为阻塞模式的接受。在超时重传后，自然会接收到 ack。

  而且为了实验发送的效率考虑，以及 GBN 实现的合理性，这里使用 3 个线程同时运行是必要的。

  最后，我们要考虑线程的一些冲突。在超时重发时，可能同时会收到 ack，这样我们 base 就会改变。所以实际上这两个线程会冲突。但是，只要我们设置锁机制就可以解决，这确保了在重发阶段，对窗口数据的访问不会与接收 ACK 的线程发生冲突。

2.2.2 接收方

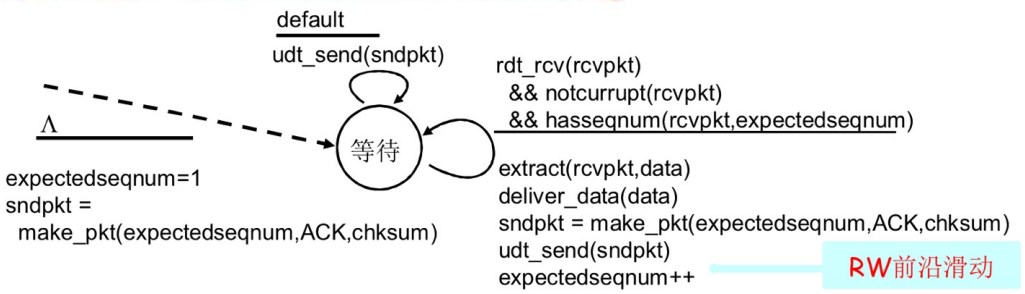

图 2.4: GBN: 接收方扩展的 FSM

由于 GBN 的接收窗口/接收缓冲区为 1，因此这里实际上用一个变量就可以解决问题。这里我们定义了一个 expectedseqnum 表示接受窗口，只有接受的数据包的 seq_num，是我们期待的序列号，我们才会接受。

在接收后，进行相关动作。主要是发送 ack 数据包，这里确认号为 expectedseqnum，然后接收窗口的前沿需要移动。

其中我们的接受 $recvfrom$ 设置为阻塞模式，因为发送方会检测到超时然后重发。

2.2.3 累计确认的概念分析

1. 接收方在接收到符合要求的序号的分组后，发送 ACK，且 ack 的序号等于当前接受分组的序号

2. 发送方接收到 ACK, 对 ACK 包中的序号之前的所有分组进行**累计确认**，即确认该分组之前的所有分组都被接收到。

3. 累积确认的逻辑在于: 接收方只有接收到了按序到来的分组，才会发送 ACK，代表之前的所有分组都接收成功。所以发送方能够进行累计确认，认证之前的所有分组都成功发送。

2.3 日志交互

- 发送方

  在发送文件线程内，首先发送文件会产生如下日志，输出发送数据包的序列号，以及校验和，然后输出对应发送窗口的情况

```
printf("发送seq:%d, 校验和:%d\n", temp.seq_number, temp.checksum);

printf("发送窗口size:%d,base:%d,nextseqnum:%d\n", nextseqnum -
    base, base, nextseqnum);
```

  在接收线程内，会打印接受的 ack 的相关情况，以及 base 移动后的发送窗口情况。

```
printf("接收到累计确认ACK, ack:%d\n",
    r_temp.msg.acknowledge_number);

 printf("接受ack后, 发送窗口size:%d,base:%d,nextseqnum:%d\n",
    nextseqnum - base, base, nextseqnum);
```

  在重发线程内，会打印重发的分组相关情况。

```
printf("重发seq:%d,校验和:%d\n", window[i].seq_number,
    window[i].checksum);
```

  最后传输时间、吞吐率

```
 unsigned long long start_stamp1 = GetTickCount64();//开始计时
send_Msgs(buffer, file_len);
unsigned long long end_stamp1 = GetTickCount64();//开始计时
cout << "--------------------" << endl;
cout << "传输文件大小:" << file_len << "bytes" << endl;
cout << "文件传输时间:" << end_stamp1 - start_stamp1 << "ms\n";
double kbps = (file_len * 8.0) / (end_stamp1 - start_stamp1);
cout << "吞吐率为:" << kbps << "kbps" << endl;
cout << "--------------------" << endl;
```

- 接收方

  实际上接收方的比较简单，针对接受的分组以及发送的 ack，进行相关打印即可。

```
printf("接受窗口size:1,接受seq:%d,check_sum=0\n",
    r_temp.msg.seq_number);

printf("发送ACK,ack:%d,校验和:%d\n", s_temp.acknowledge_number,
    s_temp.checksum);
```

2.4 总体实现

此次实验在 3-1 的基础进行，我们文件传输时，发送端与接收端建立连接，断开连接的过程不变，按照上次的三次握手、四次挥手进行。

而可靠数据传输，在上次采用 rdt3.0+ 停等机制。在这里我们改为流水线机制，采用 GBN 策略，可以完成基于 udp 的可靠的数据传输。

3 核心代码分析

3.1 数据包/协议相关

在实验 3-1 的基础上，改动的不多，具体如下：

```
#define WINDOW_SIZE 7
//数据段最大的大小
#define MSS 8016
struct Mymsg {
    unsigned short source_port;//源端口
    unsigned short dest_port;//目的端口
    unsigned short seq_number;//发送数据包的序号 16 位
    unsigned short acknowledge_number;//确认号字段 16 位
    char flag;//标志位
    char ack_id;//rdt3.0 ack 有 ack0 与 ack1
    unsigned short length;//数据包长度 16 位
    unsigned short checksum;//校验和 16 位
    char data[MSS];//数据大小，最长 MSS
};
...
```

```
class Mymsg_explain {
public:
    Mymsg msg;
    Mymsg_explain(Mymsg s) :msg(s) {}
    void Mymsg_copy(Mymsg s) { memcpy(&msg, &s, Msg_size); }
    ......
    bool isLAST() {
        if (msg.flag & LAST)
            return 1;
        return 0;
    }
    bool hasseqnum(unsigned short expected) {
        if (msg.seq_number == expected)
            return 1;
        return 0;
    }
};
//发送方
unsigned short base = 0;
unsigned short nextseqnum = 0;//与 seq_num 一致，最大 65535，超过会自动转为 0
deque<Mymsg>window;
//接收方
unsigned short expectedseqnum = 0;
```

数据包格式没有变化，这里不再重复 3-1 内容。重点在于上面的函数 *hasseqnum*，在接收方接受数据包时，会检测数据包对应的 seq，与接收窗口的期待值是否一样。

而 *isLAST* 实际上告诉接收方，这个数据包是最后一个，可以完成相关传输了。

其中，发送方关键的有 3 个变量。*base* 代表发送窗口后沿、*nextseqnum* 代表发送窗口前沿。*window* 代表发送窗口对应的分组内容，超时重发会用到 window 内的数据包，进行重发操作。

接收方实际上用一个 *expectedseqnum*，就可以表示接收窗口。

3.2 发送文件端

这里采用多线程的方式，3 个线程分别承担: 超时重传、接受 ack、窗口未满发送数据包的责任。

3.2.1 接受 ack 线程函数

具体思路如下:

- 这里的 $finish\_flag$ 代表传输完成的标志，只要传输没有完成就在函数里面进行循环。
- **等待接受 ack，采用阻塞模式**。只有接受的分组经过检验后合格 (校验和检验，ACK 检验)，才会进入相关语句，否则继续循环。
- **锁机制**。首先我们的线程和发送数据包线程不会冲突，此线程处理 base，而发送数据包处理 nextseqnum，发送窗口也可以被这两个线程一起调整。

  但是接受 ack 线程与超时重发会冲突，这里必须要采用一个锁，保证超时重发做完了，才能处理收到的 ack。或者，只有处理完了当前收到的 ack，才能超时重发。因为超时重发会用到 base 以及 window，而我们接受 ack 的动作显然会对这两个变量进行修改。

  最后，可以发现超时重发与发送数据包不会冲突。超时重发的过程中，不会接受 ack，显然发送窗口不会改变。而在发送数据包的过程中，设置的超时时间必须小于发送时间。因此，综上所述，**我们只需要对接受 ack 线程，超时重发线程设置锁机制即可**。
- **收到 ack 后采取动作**。这里我们首先进行 $base$、$window$ 的修改。需要将发送窗口的后沿进行移动，$window$ 队列储存的后沿数据包进行删除。然后是对计时进行修改。如果发送窗口为空，那么需要停止计时，因为现在没有发送数据包，这里采取全局变量 $time\_flag$ 作为计时的标志。在发送窗口不为 0，重新开始计时。
- **检测是否为最后一个分组**，如果是最后一个分组，对 $finish\_flag$ 进行修改，由于是全局变量，其他线程也会结束自己的工作，最后由主线程统一的销毁两个子线程。

```
// 接收 ACK 的函数
void receiveACK(int groupN) {
    while (!finish_flag) {
        char curr_buf[Msg_size];
        int ret = recvfrom(clientSocket, curr_buf, Msg_size, 0,
        (struct sockaddr*)&serverAddr, &server_addrlen);
        Mymsg_explain r_temp = Mymsg_explain(receive_char2msg(curr_buf));
        if (ret > 0 && r_temp.isACK() && check_sum(curr_buf, Msg_size) == 0) {
            std::lock_guard<std::mutex> lock(gbnMutex);//离开作用域自动释放锁
            //base = r_temp.msg.seq_number + 1;
            //window 移动
```

```
            resend_flag = 0;
            printf(" 接收到累计确认 ACK, ack:%d\n", r_temp.msg.acknowledge_number);
            while (base <= r_temp.msg.acknowledge_number) {
                window.pop_front();//最老的删除 最新的 back 不变
                base++;
            }
            printf(" 接受 ack 后, 发送窗口 size:%d,base:%d,nextseqnum:%d\n",
            nextseqnum - base, base, nextseqnum);
            //接下来计时相关处理, 省略展示 ....
        }
    }
    return;
}
```

然后主线程还有一个 *send_file* 函数，与 3-1 实验一致，这里不再重述。具体功能是: 完成文件的传输，在其中调用 *send_Msgs* 进行文件名/文件内容的发送。

3.2.2 超时重发线程函数

具体思路如下:

- *finish_flag* 代表传输完成标志。在没有传输完成需要进入循环。
- *time_flag* 是计时标志，只有开始计时，并且超时了，我们才会进入重发。
- **锁机制**。这里采用与接受 ack 线程中同样的锁，那么两个线程不能同时持有锁，保证了线程之间不会同时对一个共享变量进行修改。
- **重发分组**。我们会重发 *window* 中储存的发送窗口的数据分组。这里，我们重发的数据包个数为 $nextseqnum - base$，与发送窗口的尺寸一致。

```
void timeoutResend() {
    while (!finish_flag) {
        if (time_flag && GetTickCount64() - start_stamp > timeout) {
            //离开作用域自动释放锁
            std::lock_guard<std::mutex> lock(gbnMutex);
            //从 window 最老的 front 开始重发 base  nextseqnum-1
            for (int i = 0; i <= nextseqnum - base - 1; i++) {
                sendto(clientSocket, msg2char(window[i]), Msg_size, 0,
                (struct sockaddr*)&serverAddr, server_addrlen);
```

```
                printf(" 重发 seq:%d, 校验和:%d\n", window[i].seq_number
                , window[i].checksum);
            }
            resend_flag++;
            time_flag = 1;
            start_stamp = GetTickCount64();
        }
    }
    return;
}
```

3.2.3 主线程

具体思路如下:

- 完成相关初始化，创建子线程。
- 主线程进行数据包的流水线发送，一次性发送多个数据包，直至发送窗口已满。
- 当接受 ack 的子线程确认发送完成后，*finish_flag* 被置位，完成传输，销毁子线程，结束此函数。

```
void send_Msgs(char* buffer, int len) {
    ......
    std::thread ackThread(receiveACK, groupN);
    std::thread resendThread(timeoutResend);
    while (!finish_flag) {
        //只能发送 0-groupN-1 的包 最后 seq_len 为 groupN
        while (nextseqnum < base + WINDOW_SIZE && seq_len <= groupN - 1) {
            Mymsg temp = send_generateMsg(source_port, dest_port,
            nextseqnum, 0, 0, '0');
            ......
            sendto(clientSocket, msg2char(temp), Msg_size, 0,
            (struct sockaddr*)&serverAddr, server_addrlen);
            //储存进发送缓冲区
            window.push_back(temp);
            if (base == nextseqnum) {
                start_stamp = GetTickCount64();
                time_flag = 1;//启动计时
            }
```

```
            printf(" 发送 seq:%d, 校验和:%d\n", temp.seq_number, temp.checksum);
            //前沿移动，以及总位数移动
            nextseqnum++;//这里最大状态数是 65535 超过回到 0
            seq_len++;
            printf(" 发送窗口 size:%d,base:%d,nextseqnum:%d\n",
            nextseqnum - base, base, nextseqnum);
        }
    }
    ackThread.join();
    resendThread.join();
    return;
}
```

3.3 接收文件端

具体思路如下:

- 首先**初始化**，完成一些标志位的赋值, 比如 $expectedseqnum$ 等，设置 $recvfrom$ 为阻塞模式。
- **接受分组**, 进入 while 循环
    1. 首先检查接受的分组是否正确。具体调用 $hasseqnum$ 函数检查接受分组的 seq 是否等于 $expectedseqnum$。然后调用 $check\_sum$，进行校验和的检验。
    2. 检测数据包的长度，对传入函数的参数 len 进行修改。
    3. 发送 ACK，其中 $Mymsg$ 中的 ack 设置为 $expectedseqnum$，代表的是对前 $expectedseqnum$ 进行**累积确认**。
    4. 完成接受窗口的移动，直接对 $expectednum$ 进行 $+1$ 即可。
- **接受完毕检测**。如果某一个数据包的 $isLAST()$ 为 true, 说明接收完毕，返回即可。

```
void recv_Msgs(char* buffer, int& len) {
    //设置为阻塞模式... 省略代码
    len = 0;//字节数
    expectedseqnum = 0;
    int seq_len = 0;//分组数
    //分组序号 expectedsenum 一个周期是 65535
    //其中 seq_len 是记录所有分组的个数
    while (1) {
```

```
        char curr_buf[Msg_size];
        int ret = recvfrom(serverSocket, curr_buf, Msg_size,
        0, (struct sockaddr*)&clientAddr, &client_addrlen);
        Mymsg_explain r_temp = Mymsg_explain(receive_char2msg(curr_buf));
        //检测接受的 seq 是否正确
        if (ret > 0 && r_temp.hasseqnum(expectedseqnum)
        && check_sum(curr_buf, Msg_size) == 0) {
            //打印接受 seq 的日志，这里省略展示
            //判断 seq 的长度
            if (r_temp.msg.length < MSS) {
                memcpy(buffer + seq_len * MSS, r_temp.msg.data, r_temp.msg.length);
                len += r_temp.msg.length;
            }
            else {
                memcpy(buffer + seq_len * MSS, r_temp.msg.data, MSS);
                len += MSS;
            }
            //发送 ACK 对前 expectedseqnum 进行累计确认
            Mymsg s_temp = send_generateMsg(source_port, dest_port,
            0, expectedseqnum, ACK, '0');//seq 代表分组编号 ack=seq+1
            sendto(serverSocket, msg2char(s_temp), Msg_size, 0,
            (struct sockaddr*)&clientAddr, client_addrlen);
            //发送 ack 的日志，这里省略展示
            expectedseqnum++;
            seq_len++;//与 expectedseqnum 区别是分组的总数而不是序号
            //解析是否是最后一个数据包
            if (r_temp.isLAST()) {
                printf(" 接受完毕\n");
                return;
            }
        }
        else
            continue;
    }
}
```

4 实验问题及分析

4.1 窗口过大的问题及分析

当窗口设置为 32，可以发现如下图的问题。

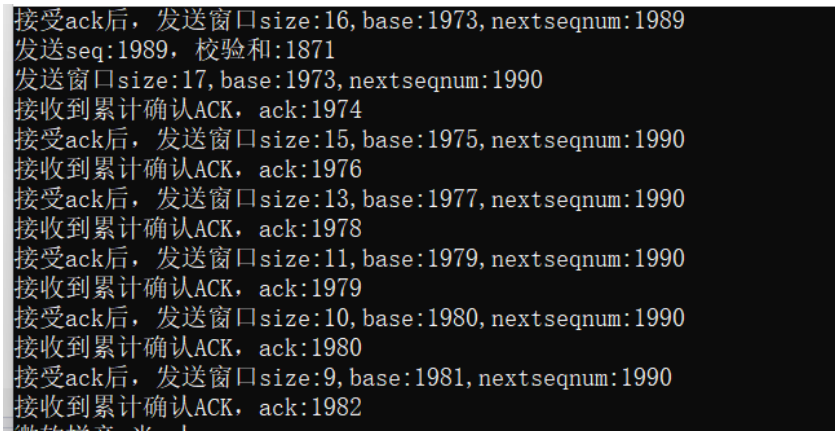

图 4.5: 发送方收到的 ack 不连续

在窗口 <16 时，发送方收到的累计确认一般都是连续的。因为接收方收到一个包，就会传累计确认。但是在图4.5中，很明显，发送方收到的累计确认 ack 从 1974 跳到了 1976，而累计确认 1975 不见了，但是接受文件方是传了累积确认 1975 的。

这里猜测，由于窗口太大，发送方传输的速度过快，导致接收方发送的 ack 也过快，导致路由器出现问题。

因为这里设置当窗口 >16 时，每次连续发送一次，sleep 1 毫秒。

此时，结果如下所示，可以看到图4.6 中 ack 明显是连续的。而在测试过程中，如果发送方最后没有收到最后一个分组的 ack，可能会一直卡死，但是接收方已经接收完毕退出。

因此，这里最好加上 sleep，不能让发送方传输速度过快。

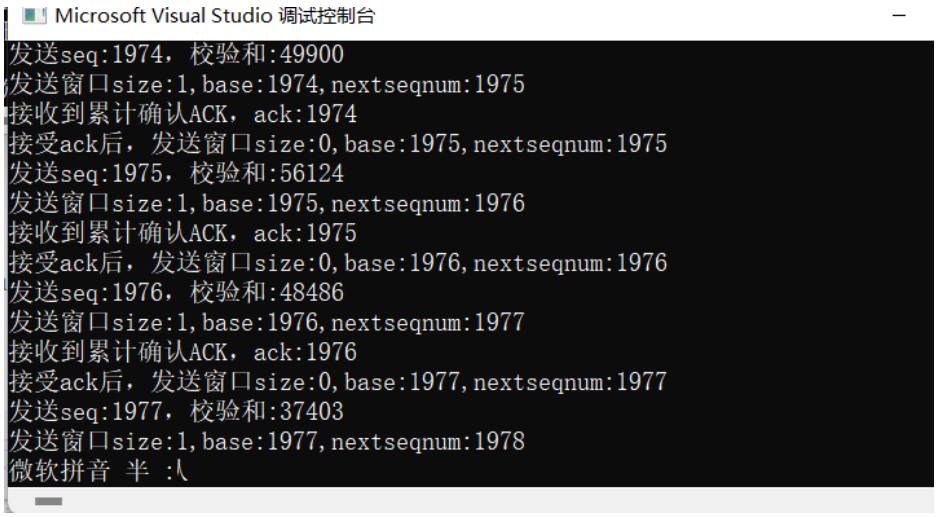

图 4.6: 发送方收到的 ack 连续

4.2 丢包延时相关问题及分析

由于路由器的相关问题，当丢包率设置为 50% 时，窗口设置为 8，会出现接收文件方收不到文件的问题。

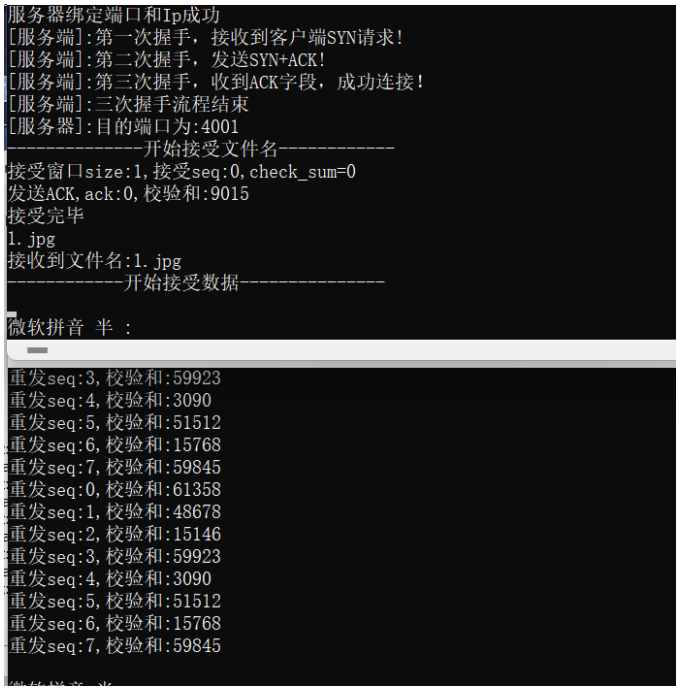

图 4.7: 接收方收不到文件的情况

研究后发现,当窗口设置为偶数,丢包为 2 个包丢 1 个时,假设发送方传的是 0,1,2,3,4,5,6,7。如果第 0 个包丢失，那么很明显偶数个包每次都会丢失。那么就会造成重传后，同样的包仍然被丢弃。

因此，这里窗口最好设置为奇数，在奇数情况下，实验时明显可以在丢包率 50% 的情况传输文件。

5 实验结果分析

5.1 无延时无丢失情况

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	2140	7281	14532	2000
吞吐率/kbps	6943.38	6480.98	6589.04	6623.23

表 1: 在 MSS : 6016、$WINDOW\_SIZE$: 4 下进行测试

上面的表1，表2，表3是对四个文件进行测试的结果。分别在 MSS 为 6016，而在 $WINDOW\_SIZE$ 不同的条件下测试传输结果。

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	2250	7234	16109	2015
吞吐率/kbps	6603.92	6523.09	5944	6573.93

表 2: 在 MSS : 6016、$WINDOW\_SIZE$: 7 下进行测试

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	2110	6797	13594	1891
吞吐率/kbps	7042.1	6942.48	7043.69	7005

表 3: 在 MSS : 6016、$WINDOW\_SIZE$: 15 下进行测试

很明显在发送窗口大小为 15 时，传输的时间是三张表最小的，而且吞吐率也是最大的。说明窗口大小可以适当调大一些，这样我们的传输性能会好一些，但是不能太大。比如在实验问题一节中分析的结论：当窗口大小为 32，会出现接收方的累计确认 ack 可能被路由器丢失的情况，这样发送方就会在最后一个数据包上出现问题。(因为接收方接收到文件就退出了，而无法判断最后一个 ack 是否发送成功)

5.2 有延时以及丢失的情况

MSS 设置为 8016, 窗口设置为 7, 延时设置 20ms, 程序的 $TIMEOUT$ 设置为 250ms, 丢包率设置 2%。这里测试丢失重传 GBN 的效率。

结果如下表所示

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	21703	58718	113688	18797
吞吐率/kbps	684.644	803.638	842.234	704.712

表 4: 延时 20ms，丢包率 2% 下进行测试

可以发现，与无延时，无丢包率的传输对比起来，明显这一组的传输时延更长，吞吐率更小。可能是因为 GBN 在丢包时会将整个分组全部重传导致的，这将造成诸多无用的重传，因此在有延时、丢包的情况下，GBN 的性能并不理想。