



南開大學
Nankai University

计算机学院
计算机网络实验报告

实验 3-3: 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 14 日

目录

1 实验要求	1
2 协议设计	1
2.1 数据包格式	1
2.2 SR 协议设计	2
2.2.1 发送方	3
2.2.2 接收方	5
2.3 日志交互	5
2.4 总体实现	7
3 核心代码分析	7
3.1 数据包/协议相关	7
3.1.1 发送方	7
3.1.2 接收方	8
3.2 发送文件端	9
3.2.1 接受 ack 线程函数	9
3.2.2 超时重发线程函数	10
3.2.3 主线程完成分组发送	11
3.3 接收文件端	13
4 实验问题及分析	15
5 实验结果分析	16
5.1 无延时无丢失情况	17
5.2 有延时以及丢失的情况	17

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: SR(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 协议设计

我们在 UDP 的基础上设计选择确认 SR 的具体过程。

2.1 数据包格式

16位 2字节		16位 2字节	
<i>source_port</i>		<i>dest_port</i>	
<i>seq_number</i>		<i>acknowledge_number</i>	
<i>flag</i>	<i>ack_id</i>	<i>length</i>	
<i>checksum</i>			
<i>data</i>		4096字节	

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 3-3 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 32768 要小。

对于接收方而言，获取到的包，需要提取其中的 `seq_number`，进行相关分析。由于我们要对落入接受窗口的进行存储，而落入上一个窗口的重发 `ack`，其他情况忽略此分组。所以在这里设计了 2 个函数，用来分析接受包的 `seq_number`。

函数 `hasnowseqnum` 主要用来判断当前分组的 `seq_number` 是否落入接受窗口。如果当前分组落入接受窗口，那么我们应该进行存储接受，若是有序分组进行上交；若为失序分组，采取缓存即可，然后移动接收窗口。

函数 `hasprevseqnum` 主要是用来判断当前分组的 `seq_number` 是否落入上一个接收窗口。如果落入上一个接收窗口，那么应该采取重发对应 `ACK` 的措施。

- `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移动。
- `length` 字段代表数据包的长度。数据包存储的数组的固定长度由 `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
- `flag` 是我们定义的各种标志位，其中重要的标志位如 `LAST`，代表这个分组是最后一个分组，接收者可以完成接收文件的任务了。

2.2 SR 协议设计

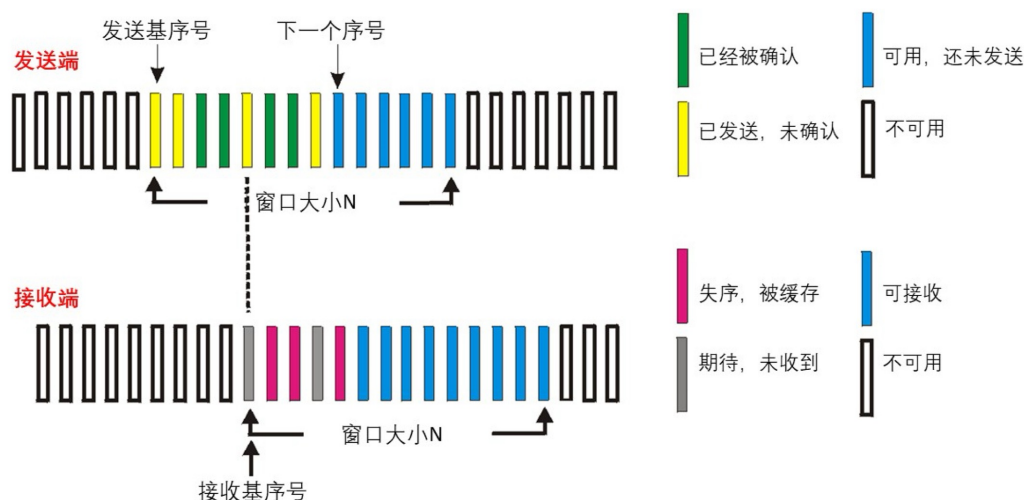


图 2.2: SR 的发送与接收方窗口图

结合上图2.2, 在本次设计的 SR 中, 发送窗口与接收窗口大小均为 N 。而在发送窗口中需要维护两个指针: 发送基序号、下一个序号, 表示我们已发送未确认的部分。

- **滑动窗口**: 在选择重发 SR 中, 发送方窗口大小大于 1, 接收方窗口大小大于 1, 在本次实验中将发送方与接收方窗口设置为同样大小。

发送窗口与接收窗口都是滑动的, 根据我们文件传输中的特定条件进行滑动。

发送方窗口大小定义了可以发送但尚未收到确认的数据包数量。它限制了发送方能够在不等待确认的情况下发送的数据数量。一旦接收到确认, 并且该确认分组序号为发送窗口后沿, 发送方窗口向前滑动, 允许发送新的数据包。

接收方窗口大小表示接收方可以乱序接收和处理的数据包数量。接收方窗口内的数据包按序接收并缓存, 以便在乱序到达时重新排序, 而随着按序到达的数据包的到来, 接收方窗口向前滑动, 允许接收更多的数据包。

- **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。
- **SR 选择重传策略**: 只会选择性针对特定丢失分组重传。

接收方会单独确认每个收到的数据包, 以告知发送方哪些数据包已经安全到达。这种独立确认让发送方可以知道哪些数据包需要被重传。

发送方维护一个窗口来追踪已发送但尚未收到确认的数据包。接收方收到损坏或丢失的数据包后, 只会请求重传这些特定的包, 而不是整个窗口内的数据。

由于发送方只会针对接收方未正确接收的数据包进行重传, 而不是整个窗口内的数据。这提高了网络的利用率和效率。

2.2.1 发送方

➤ 发送端

- **接收上层数据**: 如果发送窗口中有可用的序号, 则发送分组
- **超时(n)**: 重传分组n, 重启定时器
- **接收ACK(n)**: n 在 $[\text{send_base}, \text{send_base}+N-1]$ 区间, 将分组n标记为已接收, 如果是窗口中最小的未确认的分组, 则窗口向前滑动, 基序号为下一个未确认分组的序号

图 2.3: SR 中发送方的运行机制

结合图2.3, 我们可以设计了如下具体的发送方运行机制:

- 首先，我们用主线程进行发送分组的过程，如果当前发送窗口仍有可用的序号，那么就发送分组。

由于选择重传 SR 中，我们是对每个发送的分组安装计时器。这里经过思考，采用如下方式：建立一个全局数组 *time_flag* 用于对应每个分组是否开始计时的标志，建立一个全局数组 *start_stamp* 用于对应每个分组的起始计时时间，这里数组索引就是我们分组的序号。

选择全局数组来控制计时，且索引对应分组的序号的方式，对我们多线程运行是有较大的便利的。这样我们重发线程与接收 ack 线程，不会访问同一个数据变量，也不会产生多线程冲突。

在每次发送一个分组时，我们要开始对此分组进行计时，对 *time_flag* 及 *start_stamp* 数组对应项进行赋值。

- 创建一个子线程用于接收 ACK。在这个子线程中会检查接收到的分组。

在这里会首先检查接受分组的字节数，接受分组的校验和，接受分组的标志位是否为 ACK，同时如果分组序号 *seq* 在我们的发送窗口内，才可以接收。

如果经过检验，允许被接收，那么开始进行处理。首先取消当前分组的计时，同时判断该分组序号是否为最小未确认的分组序号，如果是，那么将发送方的窗口后沿移动。

- 创建一个子线程用于检测超时重发。这里我们只建立了一个子线程用于超时重发。主要思路是遍历我们发送窗口中维护的两个指针之间，已发送未被确认的分组。如果该分组对应的计时器超时，那么重发此分组，并重新计时。

由于为了避免线程之间产生冲突，这里重发是重新产生分组，从我们的 *buffer* 全局大数组中产生对应的分组，然后遍历的其实是上述定义的全局数组 *time_flag* 以及 *start_stamp* 数组。

- 延续上一次的发送窗口设计，这里采用一样的窗口数据结构。

```
1 deque<Mymsg>window;
```

这里的发送窗口在发送文件会进行相关存储操作。而在接收 ack 时，若此接收确认是当前最小未确认的分组，那么会进行弹出操作，同时更新 base 保证了滑动窗口的移动。而在超时重发没有用到此数据结构，是为了避免线程出现冲突。

2.2.2 接收方

➤ 接收端：接收分组n:

- n在[rcv_base, rcv_base+N-1]区间，发送ACK(n)，缓存失序分组，按序到达的分组交付给上层，窗口向前滑动
- n在[rcv_base-N, rcv_base-1]区间，发送ACK(n)

图 2.4: SR 中接收方的运行机制

结合图2.4，我们可以设计如下接收方运行的机制。

- 首先为了表示窗口，我们设计了如下的数据结构

```

1      struct SR_receive_ele {
2          Mymsg m;
3          bool receive = 0;
4      };
5      deque<SR_receive_ele>window(WINDOW_SIZE);

```

可以看到为了判断某个接收窗口内分组被接收，使用了一个 *receive* 的变量作为标志。这里的 *window* 实际上大小是固定的，在交付的时候，我们会弹出，同时为了保持 *window* 大小的固定，会弹进一些空的元素，之后可以改写。

- 只用一个线程完成接收方的工作即可。

具体接收分组 n，判断分组 n 的序号，如果在当前接收窗口内，那么可以缓存并判断是否可以交付，如果是有序分组进行交付，接收窗口前移；如果在上一个接收窗口内，重发 ack；其他情况忽视此分组即可。

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

```

1      printf("发送seq:%d, 校验和:%d\n", temp.seq_number, temp.checksum);
2
3      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，这里发送窗口理解为已发送未被确认部分，而发送缓冲区理解为一个固定的大小，而发送窗口不大于发送缓冲区。

```
1   printf("  接收到确认ACK, ack:%d\n", r_temp.msg.acknowledge_number);
2   printf("
      接受窗口最小的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。

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

最后传输时间、吞吐率

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

• 接收方

接收方在收到当前接收窗口的分组时，进行缓存，在接收窗口时打印信息，针对接受的分组以及发送的 ack，进行相关打印即可，由于接收窗口是固定大小，所以只用打印接收窗口后沿 rcvbase 即可。

```
1   printf("接收到分组, 确认并发送ACK,ack:%d,校验和:%d\n",
      s_temp.acknowledge_number, s_temp.checksum);
2   printf("进行交付后, 接收窗口size:%d,rcvbase:%d\n", WINDOW_SIZE,
      rcvbase);
3   printf("  重发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 SR 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

这里仅体现新加的部分

3.1.1 发送方

```
1 int time_flag[8000] = { 0 };
2 //针对分组进行计时的时间
3 //为了避免出现多线程问题，所以采取此措施
4 unsigned long long start_stamp[8000] = { 0 };
5 //发送窗口 采取 SR 协议
6 deque<Mymsg>window;
7 //计时器相关标志
8 #define STARTtime 2
9 #define Endtime 1
10 int finish_flag = 0; //完成传输的标志
11 unsigned long seq_len = 0; //发送的分组个数
12 unsigned short base = 0;
13 unsigned short nextseqnum = 0; //与 seq_num 一致，最大 65535，超过会自动转为 0
```

- 全局数组 *time_flag* 作为分组的计时标志，下标与分组的序号相对应。
- 全局数组 *start_stamp* 作为每个分组对应的计时器，下标与分组的序号相对应。
- 发送窗口采用一个双端队列的形式，其中储存我们的数据包 *Mymsg*。
- 设计了 *time_flag* 的一些标志，比如开始计时标志 *STARTtime*，结束计时标志 *Endtime*。
- 维护两个指针，*base*、*nextseqnum*，作为发送窗口内部已发送未确认的后沿、前沿。

3.1.2 接收方

```
1 unsigned short rcvbase = 0;
2 //接收窗口
3 struct SR_receive_ele {
4     Mymsg m;
5     bool receive = 0;
6 };
7 deque<SR_receive_ele>window(WINDOW_SIZE);
8
9 class Mymsg_explain {
10 public:
11     ....
12     bool hasnowseqnum(unsigned short rcvbase) {
13         if (rcvbase <= msg.seq_number && msg.seq_number
14             <= rcvbase + WINDOW_SIZE - 1) {
15             return 1;
16         }
17         return 0;
18     }
19     bool hasprevseqnum(unsigned short rcvbase) {
20         if (rcvbase - WINDOW_SIZE <= msg.seq_number && msg.seq_number
21             <= rcvbase - 1) {
22             return 1;
23         }
24         return 0;
25     }
26 };
```

- 使用双端队列定义我们的接收窗口，其中接收窗口认为设置成固定大小。在弹出的时候，会弹进一个空的元素，方便之后修改其值。
- 双端队列其中存储的元素类型是一个结构体。因为我们需要标识已经收到的分组，所以增加一个 *receive* 来判断，这样交付分组的时候就可以根据此标识符，来确定是否可以交付。
- 在解析数据包的类 *Mymsg_explain* 中增加两个成员函数，其中 *hasnowseqnum* 判断当前数据包的序号，是否处于当前接收窗口范围之内；而 *hasprevseqnum* 判断当前数据包的序号，是否处于前一个接收窗口。

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、窗口未满足发送数据包的责任。

3.2.1 接受 ack 线程函数

具体思路如下：

- *finish_flag* 表示文件是否传输完成的标志。在接受完所有分组的 ack 后，会将此标志位置位，从而退出。
- 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，属于发送窗口内的分组确认序号），才会进入相关语句，否则继续循环。

首先我们会要停止对应分组计时器的计时，以此标识我们此特定分组的成功发送。同时需要检测此分组确认序号是否为最小的未确认的序号，如果是，需要进行窗口的移动。这里窗口的移动采用了一个 *while* 循环，具体是利用我们发送窗口 *window*，而且利用每一个成功发送的分组，对应计时器的标志是 *Endtime*，从而可以知晓发送窗口的哪些分组已经成功被确认，进行移动。

- 如果当前窗口的后沿移动到后，那么将 *finish_flag* 置位，完成文件的发送。

```

1
2 // 模拟接收 ACK 的函数
3 void receiveACK(int groupN) {
4     while (!finish_flag) {
5         char curr_buf[Msg_size];
6         int ret = recvfrom(clientSocket, curr_buf, Msg_size, 0,
7             (struct sockaddr*)&serverAddr, &server_addrlen);
8         Mymsg_explain r_temp = Mymsg_explain(receive_char2msg(curr_buf));
9         unsigned short r_temp_ack = r_temp.msg.acknowledge_number;
10        if (ret > 0 && r_temp.isACK() && check_sum(curr_buf, Msg_size) == 0
11            && r_temp_ack >= base && r_temp_ack < base + WINDOW_SIZE - 1) {
12            resend_flag = 0;
13            //停止该分组的计时
14            time_flag[r_temp_ack] = Endtime;
15            //如果是最小分组，窗口前移
16            if (base == r_temp.msg.acknowledge_number) {
17                //这里可能会弹出多个 从最老的开始 发送都是加在 back 尾部 弹出在 front
18                while (window.size() && time_flag[window.front().seq_number] == Endtime) {

```

```

19         window.pop_front();//最老的删除 最新的 back 不变
20         base++;
21     }
22 }
23 if (base == groupN) {
24     finish_flag = 1;
25     return;//接受完毕
26 }
27 }
28 }
29 return;
30 }

```

3.2.2 超时重发线程函数

具体思路如下:

- 由于我们需要检测发送从窗口内的每个分组的计时器情况,为了不重发建立线程,销毁线程,这里采用一个线程遍历发送窗口内的分组,判断每个分组有无超时。
- *finish_flag* 是文件发送完完毕的标志,由接受 *ack* 线程进行置位。
- 由于我们维护了发送窗口内的两个指针 *base,nextseqnum*,分别代表已发送未确认的后沿、前沿。所以这里只用遍历这些分组即可。
- 首先判断此分组是否开始计时,且计时器是否超时。只有计时开始而且超时,才会进入处理,计时开始是在发送分组时处理的。

首先重新开启计时器。然后进行重发,主要是重新产生对应的分组,然后重发即可。

- 这里设置了一个 *ERRORCOUNAT*,如果一直重发而不接收 *ack*,那么超过这个次数,我们会定义为传输出错,直接停止。

而如果仅超过 *MISSCOUNT*,这里会在前两次进行调整超时时间的策略。

```

1
2 void timeoutResend(int groupN, int len) {
3     while (!finish_flag) {
4         //判断发送窗口的元素是否存在超时
5         for (int i = base; i < nextseqnum; i++) {
6             if (time_flag[i] == STARTtime && GetTickCount64() - start_stamp[i] > timeout) {
7                 //重置时间

```

```

8         start_stamp[i] = GetTickCount64();
9         //重发
10        Mymsg temp = send_generateMsg(source_port, dest_port, i, 0, 0, '0');
11        if (i == groupN - 1) {
12            temp.length = len - (groupN - 1) * MSS;
13            temp.flag |= LAST;
14            memcpy(temp.data, buffer + i * MSS, len - (groupN - 1) * MSS);
15        }
16        else
17            memcpy(temp.data, buffer + i * MSS, MSS);
18        temp.checksum = 0;
19        char* p = msg2char(temp);
20        temp.checksum = check_sum(p, Msg_size);
21        sendto(clientSocket, msg2char(temp), Msg_size, 0,
22        (struct sockaddr*)&serverAddr, server_addrlen);
23        resend_flag++;
24        //如果连续重发, 没有收到 ack, 超过 ERRORCOUNT, 鉴定为出错
25        if (resend_flag > ERRORCOUNT) {
26            printf("server has exited or something crashed!\n");
27            finish_flag = 1;
28        }
29        else if (changetime_flag < 2 && resend_flag > MISSCOUNT) {
30            printf(" 调整 TIMEOUT!\n");
31            changetime_flag++;
32            resend_flag = 0;
33            timeout *= 1.5;
34        }
35    }
36    }
37    }
38    return;
39 }

```

3.2.3 主线程完成分组发送

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。

- 在发送数据包的同时，需要开启此数据包的计时，对 *time_flag* 及 *start_stamp* 数组进行相关赋值，同时发送窗口储存此分组。
- 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。

```

1 //主线程完成发送分组的任务
2 void send_Msgs(char* buffer, int len) {
3     //分组
4     .....
5     std::thread ackThread(receiveACK, groupN);
6     std::thread resendThread(timeoutResend, groupN, len);
7     while (!finish_flag) {
8         while (nextseqnum < base + WINDOW_SIZE && seq_len <= groupN - 1) {
9             Mymsg temp = send_generateMsg(source_port, dest_port, nextseqnum, 0, 0, '0');
10            ...//这里进行数据端的赋值，省略
11            sendto(clientSocket, msg2char(temp), Msg_size, 0,
12                (struct sockaddr*)&serverAddr, server_addrlen);
13            //储存进发送缓冲区
14            //发送新分组，开启计时
15            time_flag[nextseqnum] = STARTtime;//开始计时
16            start_stamp[nextseqnum] = GetTickCount64();
17            window.push_back(temp);
18
19            printf(" 发送 seq:%d, 校验和:%d\n", temp.seq_number, temp.checksum);
20            //前沿移动，以及总位数移动
21            nextseqnum++;//这里最大状态数是 65535 超过回到 0
22            seq_len++;
23            //不让发送端发送过快
24            Sleep(20);
25        }
26    }
27    ackThread.join();
28    resendThread.join();
29    return;
30
31 }

```

3.3 接收文件端

具体思路如下:

- **接收的分组在接收窗口内:** 如果经过校验和检验等, 开始缓存。首先发送特定 ack 表示此分组已被接收, 同时缓存此分组至接收窗口内部。

这里我们采用 *SR_receive_ele* 类型作为接收窗口的元素类型, 结构体第一个属性就是我们的数据包, 第二个属性是一个标识符, 标识此分组接收成功, 可以交付。然后写入接收窗口 *window* 内部。

如果此分组的序号恰好是我们接收窗口的后沿, 那么可以进行交付。交付主要利用了一个循环。如果此窗口头部 (恰好是最先该接收的分组) 被接收到, 那么进行交付, 将 *window* 内的数据写入我们的数组 *buffer* 中。在交付时我们要弹出头部元素, 为了保证接收窗口大小不变, 这里在尾部增加一个空元素, 那么可以用 *window[]* 去修改。

- **接收到分组在上一个接收窗口内:** 此时我们需要重发 ack 即可, 不用处理别的事情。

```

1
2 //buffer 储存收到的数据
3 void recv_Msgs(char* buffer, int& len) {
4     ...
5     len = 0; //字节数
6     rcvbase = 0;
7     int seq_len = 0; //分组数
8     while (1) {
9         char curr_buf[Msg_size];
10        int ret = recvfrom(...);
11        Mymsg msg_temp = receive_char2msg(curr_buf);
12        Mymsg_explain r_temp = Mymsg_explain(msg_temp);
13        if (ret > 0 && r_temp.hasnowseqnum(rcvbase) && check_sum(curr_buf, Msg_size) == 0) {
14            ...
15            //发送 ACK 对接受到的分组确认
16            Mymsg s_temp = send_generateMsg(source_port, dest_port, 0,
17            r_temp.msg.seq_number, ACK, '0');
18            sendto(serverSocket, msg2char(s_temp), Msg_size, 0,
19            (struct sockaddr*)&clientAddr, client_addrlen);
20            //首先缓存
21            SR_receive_ele ele;
22            ele.m = msg_temp;

```

```

23     ele.receive = 1; //表示此分组被接收, 可以交付
24     //因为初始化及后续都保证了 window 的 size, 所以可以这样使用
25     window[r_temp.msg.seq_number - rcvbase] = ele;
26     //如果有序, 交付多个
27     if (rcvbase == r_temp.msg.seq_number) {
28         while (window.size() && window.front().receive) {
29             //交付分组
30             if (window.front().m.length < MSS) {
31                 memcpy(buffer + seq_len * MSS, window.front().m.data,
32                     window.front().m.length);
33                 len += window.front().m.length;
34             }
35             else {
36                 memcpy(buffer + seq_len * MSS, window.front().m.data, MSS);
37                 len += MSS;
38             }
39             //判断是否为最后一个包
40             if (window.front().m.flag & LAST) {
41                 printf(" 接受完毕\n");
42                 return;
43             }
44             rcvbase++; //窗口后沿前移 下一个期待接受 seq
45             seq_len++;
46             window.pop_front();
47             //我们要保证 window 的尺寸, 这里 push 空消息
48             SR_receive_ele nullmsg;
49             window.push_back(nullmsg);
50         }
51     }
52 }
53 else if (ret > 0 && r_temp.hasprevseqnum(rcvbase) &&
54 check_sum(curr_buf, Msg_size) == 0) {
55     //只用发送 ACK
56     Mymsg s_temp = send_generateMsg(source_port, dest_port, 0,
57     r_temp.msg.seq_number, ACK, '0'); //seq 代表分组编号 ack=seq+1
58     sendto(serverSocket, msg2char(s_temp), Msg_size, 0,
59     (struct sockaddr*)&clientAddr, client_addrlen);
60 }
61 else

```



```
62         continue;
63     }
64 }
```

4 实验问题及分析

在实验过程中，主要是服务器发送的 ACK，可能被丢失。虽然路由器说明中显示，其不会丢失服务器发送的包，但是从日志分析来看，服务器发送了 ACK，但是客户端没有收到的情况，是明显存在的。

这样会导致一个问题，就是最后服务器确实收到文件了，但是客户端不知道服务器是否收到文件，导致客户端无法结束。

在这种问题下，我进行了以下相关设计并解决了问题。

- 首先，服务器如果已经成功接收到文件，此时应该再传一次 ACK 确认，同时标志位加上 *LAST* 标志，如果客户端接收到此确认就可以直接结束，因为服务器成功收到文件。

```
1     //判断是否为最后一个包
2     if (window.front().m.flag & LAST) {
3         if (name_flag) {
4             //文件传输完毕，要发一条消息告知
5             Sleep(20);
6             Mymsg s_temp = send_generateMsg(source_port, dest_port, 0,
7             r_temp.msg.seq_number, ACK | LAST, '0');
8             sendto(serverSocket, msg2char(s_temp), Msg_size, 0,
9             (struct sockaddr*)&clientAddr, client_addrlen);
10            printf(" 结束标志\n");
11        }
12        printf(" 接受完毕\n");
13        return;
14    }
```

- 加入服务器最后退出前，传送的 ACK 也被路由器丢失，此时，我们针对客户端做改进。

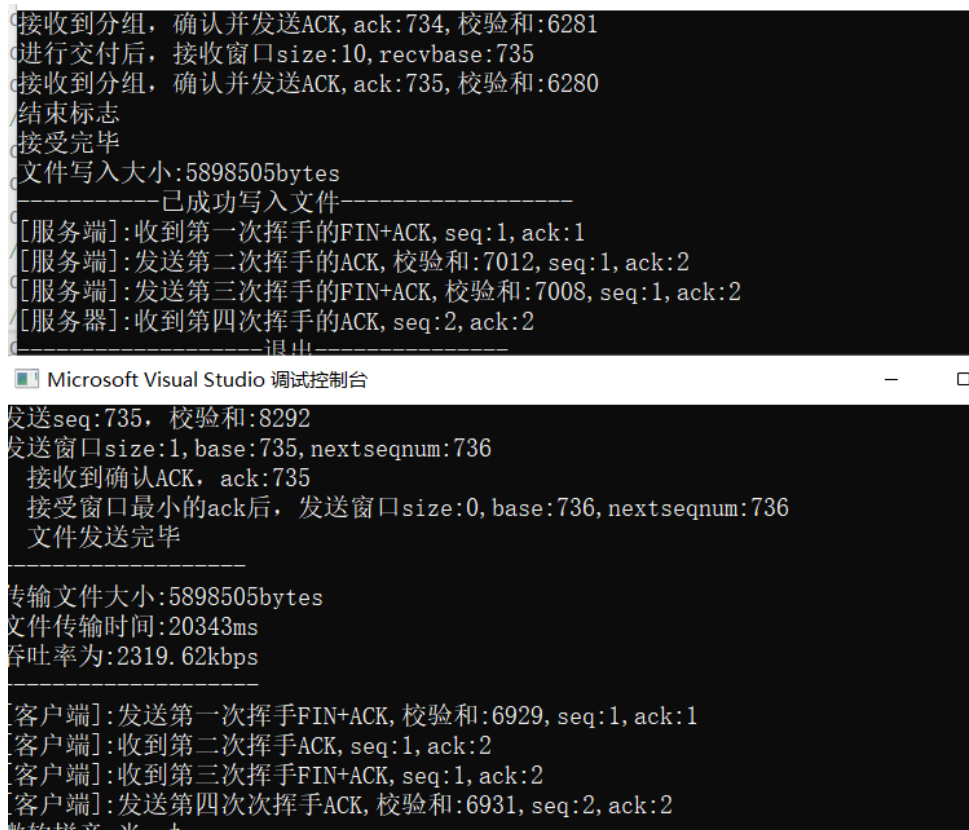
客户端如果连续重发，而中途没有接收到 ack 确认，我们其实可以认为是服务器退出。此时认为服务器收到文件，那么客户端直接结束文件传输即可。

```

1 //如果连续重发，没有收到 ack，超过 MISSCOUNT，鉴定为 server 退出了接收文件过程
2 if (resend_flag >= ERRORCOUNT) {
3     printf("server has exited\n");
4     finish_flag = 1;
5     break;
6 }

```

- 其实我们可以确保服务器一定能接收到文件，只是在这里为了确保客户端可以正常退出发送文件的过程，所以进行了如上的改进。如下图，成功退出了，解决了问题。自此实验成功完成。



```

接收到分组，确认并发送ACK, ack:734, 校验和:6281
进行交付后，接收窗口size:10, recvbase:735
接收到分组，确认并发送ACK, ack:735, 校验和:6280
结束标志
接受完毕
文件写入大小:5898505bytes
-----已成功写入文件-----
[服务端]:收到第一次挥手的FIN+ACK, seq:1, ack:1
[服务端]:发送第二次挥手的ACK, 校验和:7012, seq:1, ack:2
[服务端]:发送第三次挥手的FIN+ACK, 校验和:7008, seq:1, ack:2
[服务器]:收到第四次挥手的ACK, seq:2, ack:2
-----退出-----

Microsoft Visual Studio 调试控制台

发送seq:735, 校验和:8292
发送窗口size:1, base:735, nextseqnum:736
  接收到确认ACK, ack:735
  接受窗口最小的ack后，发送窗口size:0, base:736, nextseqnum:736
  文件发送完毕
-----
传输文件大小:5898505bytes
文件传输时间:20343ms
吞吐率为:2319.62kbps
-----
[客户端]:发送第一次挥手FIN+ACK, 校验和:6929, seq:1, ack:1
[客户端]:收到第二次挥手ACK, seq:1, ack:2
[客户端]:收到第三次挥手FIN+ACK, seq:1, ack:2
[客户端]:发送第四次挥手ACK, 校验和:6931, seq:2, ack:2

```

图 4.5: 传输 1.jpg 的日志输出

5 实验结果分析

注意这里客户端每发送一个分组是会睡眠 20ms，再继续发送的，所以下面的延时会比较大，这样做的考虑是：路由器会丢失服务器传来的 ACK，只有降低客户端发送分组的速度，才能降低服务器发送 ACK 的速度，从而避免过多的丢失服务器发送的 ACK。

5.1 无延时无丢失情况

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	7328	23578	47031	6485
吞吐率/kbps	2027.68	2001.36	2035.93	2042.63

表 1: 在 $MSS : 8016$ 、 $WINDOW_SIZE : 4$ 下进行测试

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	7343	23219	47031	6531
吞吐率/kbps	2023.54	2032.3	2035.93	2028.24

表 2: 在 $MSS : 8016$ 、 $WINDOW_SIZE : 7$ 下进行测试

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	7282	23266	47187	6485
吞吐率/kbps	2040.49	2028.2	2029.2	2042.63

表 3: 在 $MSS : 8016$ 、 $WINDOW_SIZE : 15$ 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 8016, 而在 $WINDOW_SIZE$ 不同的条件下测试传输结果。

可以看到在路由器无延时、无丢包时, 我们的窗口大小对传输时延、吞吐率没有较大影响, 上面 3 张表的吞吐率都差不多一致。

5.2 有延时以及丢失的情况

MSS 设置为 8016, 窗口设置为 10, 延时设置 5ms, 程序的 $TIMEOUT$ 设置为 500ms, 丢包率设置 5%。这里测试选择重传的效率。

结果如下表所示

	1.jpg	2.jpg	3.jpg	helloworld.txt
传输时延/ms	14281	32265	65391	10172
吞吐率/kbps	1040.46	1462.51	1464.3	1302.25

表 4: 延时 5ms, 丢包率 5% 下, 窗口大小 10 进行测试

可以发现吞吐率与传输时延下降的并不多, 对比之前 GBN, 在丢包与延时的情况下, 效率是比较高的。