

南开大学

计算机网络实验报告

实验 3-3



专 业_____信息安全_____
学 号_____2113662_____
姓 名_____张丛_____
班 级 _____信安一班_____

一、实验目的

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。

二、实验要求

- 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
- 选择确认：SR
- 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：1.jpg、2.jpg、3.jpg、helloworld.txt

三、实验原理

累积确认和选择确认的区别：

累积确认：对序号为 n 的分组的确认为采取累积确认的方式，表明接收方已正确接收到序号为 n 的以前且包括 n 在内的所有分组。

选择确认：允许接收方确认成功接收的多个、不连续的字节范围。

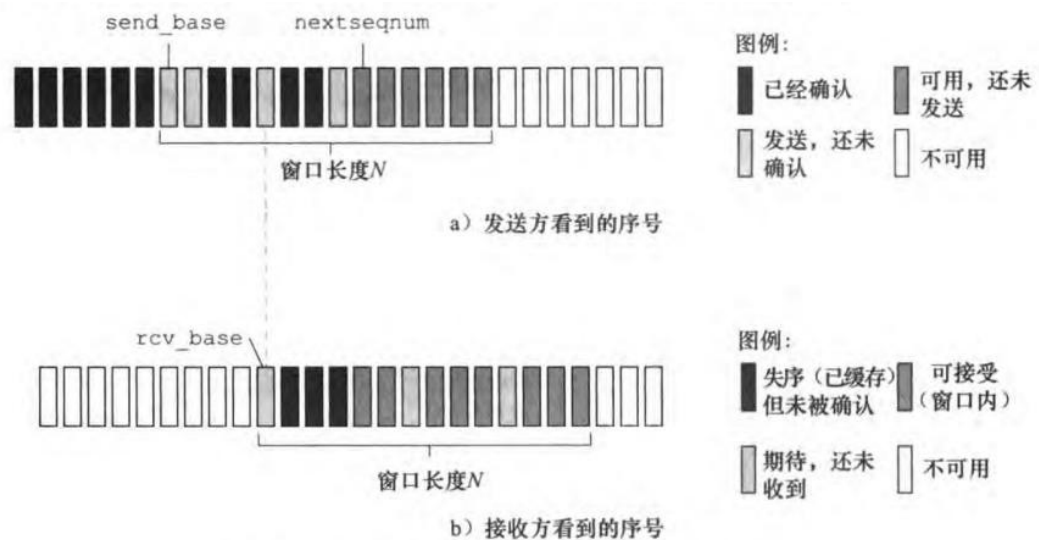


图 3-23 选择重传 (SR) 发送方与接收方的序号空间

选择重传:

发送方仅重传那些它怀疑在接收方出错 (即丢失或受损) 的分组而避免了不必要的重传。

接收方将确认一个正确接收的分组而不管其是否按序, 失序的分组将被缓存直到所有丢失分组 (即序号更小的分组) 皆被收到为止, 这时才可以将一批分组按序交付给上层。

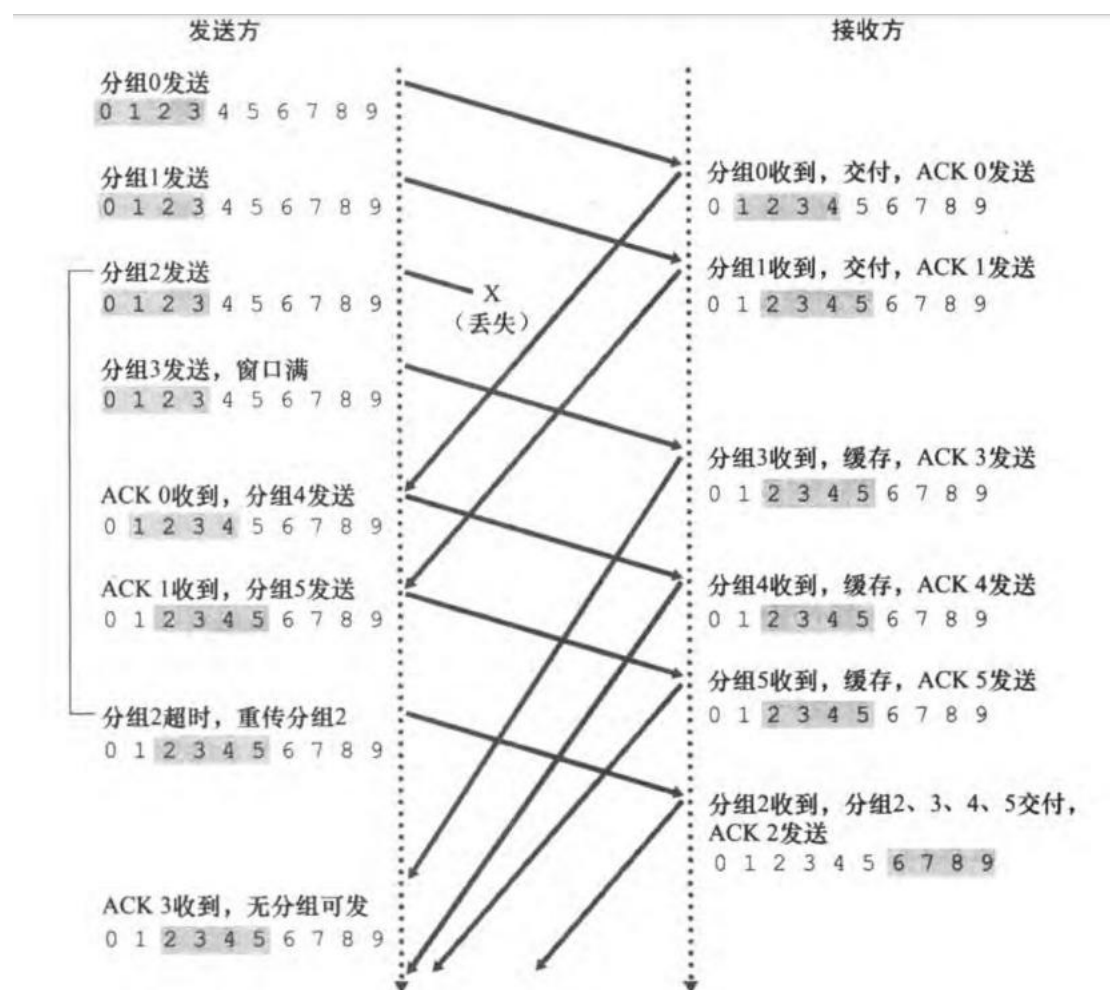
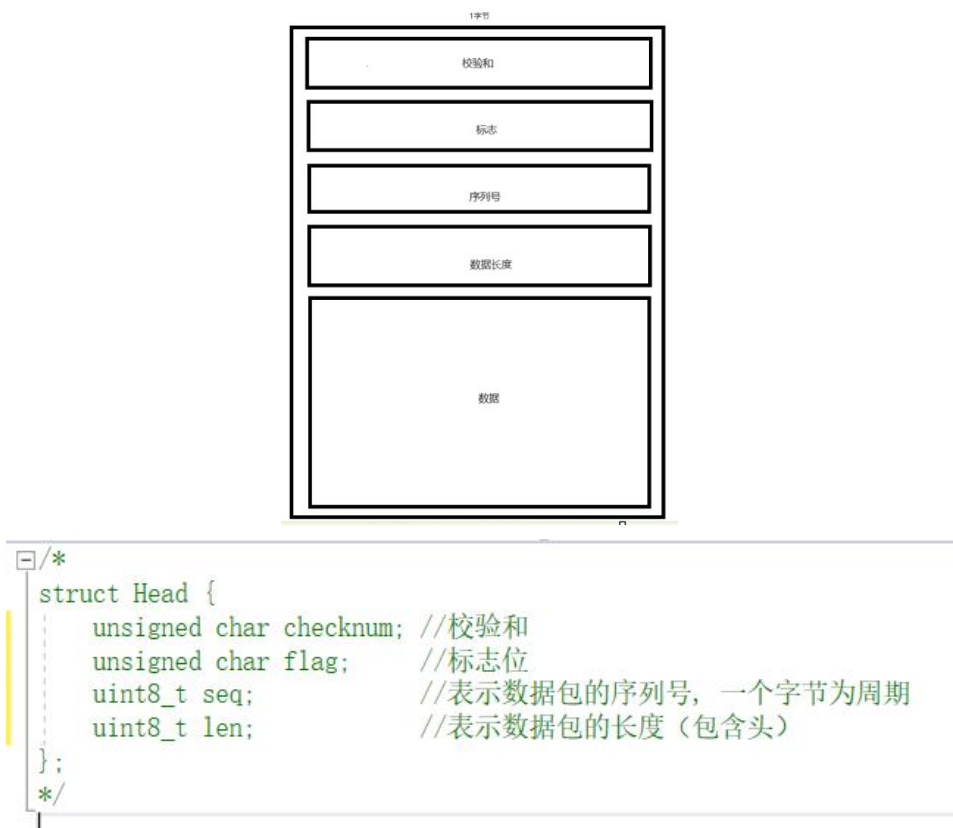


图 3-26 SR 操作

四、实验内容

在实验 3-1 中的重复内容（如三次握手、两次挥手、差错检测等等）不再赘述。

协议设计



在实验 3-2 中，我们已经实现了停等机制改成基于滑动窗口的流量控制机制，在本次实验 3-3 中，滑动窗口的基本思路没有太大变化，但需要根据选择重传来调整。

发送端(Client):

1. 从上层收到数据。当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。如果序号位于发送方的窗口内，则将数据打包并发送；否则就像在 GBN 中一样，要么将数据缓存，要么将其返回给上层以便以后传输。
2. 超时。定时器再次被用来防止丢失分组。然而，现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。可以使用单个硬件定时器模拟多个逻辑定时器的操作 [Varghese 1997]。
3. 收到 ACK。如果收到 ACK，倘若该分组序号在窗口内，则 SR 发送方将那个被确认的分组标记为已接收。如果该分组的序号等于 send_base，则窗口基序号向前移动到具有最小序号的未确认分组处。如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

图 3-24 SR 发送方的事件与动作

在 3-3 中，设置窗口如下：

```
l send_buffer(char* buffer, int len) {  
  
    vector<int> win;                                //窗口队列  
    int timer[SEQ_SPACE + 1] = { 0 };              //计时器，记录包发送出去的时间  
    bool is_ack[SEQ_SPACE+1] = { 0 };              //标记接收到的ACK  
    int fail_num[SEQ_SPACE + 1] = { 0 };           //记录重传失败次数  
  
    int package_num = len / MAX_UDP_LEN + (len % MAX_UDP_LEN != 0); //数据包总数  
    cout << "数据包总数: " << package_num << endl;  
    int base = 0;                                    //基序号(0到package_num-1)  
    int has_send = 0;                                //已发送未确认的数量(0到package_num-1)  
    int next_seq = base;                             //标识未确认的数据包  
    int has_ack = 0;                                //已确认的数量
```

相较于实验 3-2，主要多了 is_ack[] 用于标记数据包。

然后创建发送数据包的线程：

```
thread sendThread  
([&]() {  
    //窗口还有空余并且还要数据包能进入窗口  
    while (win.size() < WIN_SIZE && has_send != package_num) {  
        lock_guard<std::mutex> lock(sendMutex);  
        send_package(buffer + has_send * MAX_UDP_LEN,  
            has_send == package_num - 1 ? len - (package_num - 1) * MAX_UDP_LEN : MAX_UDP_LEN,  
            next_seq,  
            has_send == package_num - 1);  
  
        timer[next_seq] = clock();  
        cout << "已发送数据包: " << (next_seq) << endl << endl;  
        win.push(next_seq);  
  
        next_seq++;  
        next_seq %= SEQ_SPACE;  
        has_send++;  
    }  
}
```

win.size()来控制窗口长度，has_ack 来判断是否还有未发送的包，send_package 来发送数据包，timer[]记录时间。

在发送端（Client）的发送线程（sendThread）中，还设置了**超时重传**：

```

if (!win.empty())
{
    //超时，且没有ACK的数据包，重传
    vector<int>::iterator it;
    int i = 0;
    for (it = win.begin(); it != win.end(); ++it)
    {
        if ((clock() - timer[*it] > TIMEOUT) && (is_ack[*it] == 0))
        {
            cout << "超时，重传数据包" << *it << endl;
            int temp = has_send - (win.size()-i);
            send_package(buffer + temp * MAX_UDP_LEN,
                temp == package_num - 1 ? len - (package_num - 1) * MAX_UDP_LEN : MAX_UDP_LEN,
                *it,
                temp == package_num - 1);

            //刷新计时器，记录重传次数
            timer[*it] = clock();
            fail_num[*it]++;

            if (fail_num[*it] >= 10)
            {
                exit(1);
            }
        }
    }
}

```

迭代窗口中的数据包，检查未标记的数据包是否超时，对所有的未标记的超时数据包进行重传。

使用 `vector<int>::iterator it` 进行迭代。

重传会刷新计时器，但不会改变数据包在窗口的顺序。

在发送线程（`sendThread`）之外，进行接收 ACK 并处理。即多线程同时收发。

```

1.  char recv[3];
2.  if (recvfrom(m_ClientSocket, recv, 3, 0, (sockaddr*)&m_ServerAddress, &len_addr) !=
    SOCKET_ERROR && check_sum(recv, 3) == 0 && recv[1] == ACK)
3.  {
4.      SUCCESS_RECV++;
5.      //大序 ACK
6.      if ((win.front() > SEQ_SPACE - WIN_SIZE) && ((int)recv[2] < WIN_SIZE)) {
7.          is_ack[(int)recv[2]] = 1;
8.      }
9.      if ((win.front() < (int)recv[2])) {
10.         if ((win.front() < WIN_SIZE) && ((int)recv[2] > SEQ_SPACE - WIN_SIZE))
11.         {
12.             break;

```

```

13.     }
14.     is_ack[(int)recv[2]] = 1;
15.     }
16.     cout << "接收到 ACK: " << (int)recv[2] << endl;
17.     cout << "窗口队列头的序号: " << win.front() << endl;
18.     cout << "窗口队列的大小: " << win.size() << endl << endl;
19.
20.     //接收到期望 ACK
21.     if (win.front() == (int)recv[2]) {
22.         is_ack[(int)recv[2]] = 1;
23.         //更新窗口
24.         int index = (int)recv[2];
25.         while (is_ack[index])
26.         {
27.             is_ack[index] = 0;
28.             base++;
29.             has_ack++;
30.             win.pop();
31.             index++;
32.             index %= SEQ_SPACE;
33.         }
34.     }
35.     //对于 ACK<win.front()的情况不用理会
36. }

```

分三种处理，接收到大序、小序和期望的 ACK。

（期望 ACK 就是窗口头的 ACK，大序和小序基于期望 ACK 来判断。）

对于**大序 ACK**:

```

    is_ack[(int)recv[2]] = 1;

```

就是将此数据包**标记**为已经正确接收。

被标记后，数据包将不会进入超时重传的检测。

对于**期望 ACK**:


```

//接收到期望ACK
if (win.front() == (int)recv[2]) {
    is_ack[(int)recv[2]] = 1;

    //更新窗口
    int index = (int)recv[2];
    while (is_ack[index])
    {
        is_ack[index] = 0;
        base++;
        has_ack++;
        win.pop();

        index++;
        index %= SEQ_SPACE;
    }
}

```

就是在一个 while 里面进行迭代，移动窗口，移动到具有最小序号的未确认分组处。

index 用于迭代分组序号（数据包序号）。

窗口移动时，还需要恢复 is_ack[] 状态（即置为 0）以便下一个序号空间使用。

此时窗口长度减小，在 sendThread 线程内会进行判断并发送新数据包。

对于**小序 ACK**，就是什么都不用做（因为已经确认了，还不es窗口内）。

接收端 (Server) :

1. 序号在 $[rcv_base, rcv_base + N - 1]$ 内的分组被正确接收。在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。如果该分组以前没收到过，则缓存该分组。如果该分组的序号等于接收窗口的基序号（图 3-23 中的 rcv_base ），则该分组以及以前缓存的序号连续的（起始于 rcv_base 的）分组交付给上层。然后，接收窗口按向前移动分组的编号向上交付这些分组。举例子来说，考虑一下图 3-26。当收到一个序号为 $rcv_base = 2$ 的分组时，该分组及分组 3、4、5 可被交付给上层。
2. 序号在 $[rcv_base - N, rcv_base - 1]$ 内的分组被正确收到。在此情况下，必须产生一个 ACK，即使该分组是接收方以前已确认过的分组。
3. 其他情况。忽略该分组。

图 3-25 SR 接收方的事件与动作

此次实验，接收端的难点在于对缓存区的处理。

设立缓存区：

```
id recv_packet(char* message, int& len_recv) {  
    char recv[MAX_UDP_LEN + 4]; //接收的数据包  
    char package_buffer[RECV_BUFFER_SIZE * (MAX_UDP_LEN + 4)]; //缓存区  
    int package_req[RECV_BUFFER_SIZE]; //缓存区序号标识  
    for (int i = 0; i < RECV_BUFFER_SIZE; i++)  
    {  
        package_req[i] = -2; //-2表示未占用  
    }  
    int buffer_len = 0; //现有缓存数据包数量  
  
    uint8_t recv_req = 0; //数据包序号  
    len_recv = 0; //缓存区字节序号
```

package_buffer 用于缓存数据包, package_req 用于标识缓存的数据包序号和位置。

首先还是不断接收数据包：

```
while (true) {  
    memset(recv, 0, sizeof(recv));  
    while (recvfrom(m_ServerSocket, recv, MAX_UDP_LEN + 4, 0, (sockaddr*)&m_ClientAddress, &len_addr) == SOCKET_ERROR)
```

处理数据包：

首先，对应接收到的数据包，不管序号大小，都**回应 ACK**。

即便是，对于已经确认过的，甚至不在窗口内的，也需要回应（即 base-N 到 base-1 的数据包），因为要考虑之前的回应 ACK 丢失的问题：

```
char send[3];
int is_write = 0; //是否交付
if (check_sum(recv, MAX_UDP_LEN + 4) == 0)
{
    send[1] = ACK; //标志
    send[2] = (int)recv[2] % 100;
    send[0] = check_sum(send + 1, 2); //校验和

    //因为在这里就回应ACK了，得保证大序数据包进缓存区，否则没进缓存区却回应了ACK会有问题
    //通过发送端和接收端的窗口长度控制

    //另一种就是缓存成功后才回应ACK，但这时需要考虑不需要缓存的数据包的ACK

    sendto(m_ServerSocket, send, 3, 0, (sockaddr*)&m_ClientAdress, sizeof(m_ClientAdress));
    cout << endl << "期望的包: " << (int)recv_req << endl;
    cout << "接收端确认序列号: " << (int)recv[2] << endl;
```

这里有一个问题，就是，你既然回应了 ACK，对于你接收端需要缓存的数据包，你接收端就必须缓存。这需要控制窗口长度和缓存区大小，分别在发送端和接收端实现。

还有一种方法，就是，对于需要缓存的数据包，接收端缓存后再发送 ACK,这样不必过分担心缓存区溢出的问题，但需要对“不需要缓存的数据包”进行额外处理。

若接收到的是，**没收到过的数据包**，需要缓存：

```

//是否缓存过此数据包
if (find_value(package_req, RECV_BUFFER_SIZE, (int)recv[2]) != -1)
{
    cout << "数据包" << (int)recv[2] << "已缓存" << endl << endl;
}
else
{
    if (buffer_len >= RECV_BUFFER_SIZE)
    {
        cout << "缓存区溢出啦！！呜呜呜……发送端搁这乱发啥呢" << endl << endl;
    }
    else    //缓存区未满
    {
        //temp即可缓存位置
        int temp = find_value(package_req, RECV_BUFFER_SIZE, -2);
        if ((int)recv[2] == 5) { ; }
        package_req[temp] = (int)recv[2];
        for (int i = 0; i < MAX_UDP_LEN + 4; i++)
        {
            package_buffer[temp * (MAX_UDP_LEN + 4) + i] = recv[i];
        }
        cout << "缓存数据包" << package_req[temp] << endl;
        buffer_len++;
        cout << "缓存包的个数：" << buffer_len << endl << endl;
    }
}

```

流程是：

- 判断是否是大序（考虑序号空间）
- 是否缓存过
- 缓存区满了没
- 没满就要将数据包存进 package_buffer, 将序号存进 package_req。

若收到过的小序数据包，已经回应了 ACK，无需额外处理。

若接收端期望的数据包：

```

//等序数据包
if ((uint8_t)recv[2] == recv_req)
{
    is_write = 1;
}

```

```

    if (is_write)
    {
        recv_req++;
        recv_req %= 100;
        break;
    }

```

不需要缓存，直接进行交付。

交付数据如下：

```

if (recv[1] == END) {
    cout << "已接收最后一个数据包" << endl;
    for (int i = 4; i < recv[3] + 4; i++)
        message[len_recv++] = recv[i];
    break;
}
// 将数据交付
for (int i = 4; i < MAX_UDP_LEN + 4; i++) {
    message[len_recv++] = recv[i];
}

```

并且需要交付，连续的缓存过的分组（如果有）：

```

//尝试继续交付缓存区
int index = find_value(package_req, RECV_BUFFER_SIZE, recv_req);
int is_over = 0;
while (index != -1)
{
    //若最后一个包在缓存区
    if (package_buffer[index * (MAX_UDP_LEN + 4) + 1] == END)
    {
        cout << "缓存区已交付最后一个数据包" << endl;
        for (int i = 4; i < package_buffer[index * (MAX_UDP_LEN + 4) + 3] + 4; i++)
            message[len_recv++] = package_buffer[index * (MAX_UDP_LEN + 4) + i];
        recv_req++;
        recv_req %= 100;

        package_req[index] = -2;
        buffer_len--;

        is_over = 1;
    }
    else //正常的

```

```

else //正常的
{
    for (int i = 4; i < MAX_UDP_LEN + 4; i++)
        message[len_rcv++] = package_buffer[index * (MAX_UDP_LEN + 4) + i];
    rcv_req++;
    rcv_req %= 100;

    //清理
    package_req[index] = -2;
    buffer_len--;
}
index = find_value(package_req, RECV_BUFFER_SIZE, rcv_req);
}
if(is_over)
{
    break;
}

```

因为最后一个包也可能在缓存区，所以进行了判断。

缓存区交付后，package_req[]置为初始状态-2 表示没有占用。

其中 find_value 的作用是在数组中找到 value 的位置，找不到则返回-1.

```

//在list中找到值为value的位置，没有则返回-1
int find_value(int* list, int len, int value)
{
    for (int i = 0; i < len; i++)
    {
        if (list[i] == value)
        {
            return i;
        }
    }
    return -1;
}

```

以上就是发送端和接收端相对于实验 3-2 的主要更改。

实验效果

在路由器设置丢包、延时后

Router

路由器IP:
127 . 0 . 0 . 1

服务器IP:
127 . 0 . 0 . 1

端口:
4003

服务器端口:
4002

丢包率:
5
%

延时:
1
ms

确定

修改

日志

对于四个测试文件：

1. jpg:

C:\Users\zc\Desktop\Server\UDP可靠传输

服务端正在运行-----ip: 127.0.0.1

接收到客户端第一次SYN

已发送SYN_ACK

接收到客户端ACK

已完成三次握手

开始接收文件

期望的包: 0

接收端确认序列号: 0

已接收最后一个数据包

文件名: 1. jpg

期望的包: 0

接收端确认序列号: 0

期望的包: 1

接收端确认序列号: 1

期望的包: 2

接收端确认序列号: 2

期望的包: 3

接收端确认序列号: 3

期望的包: 4

接收端确认序列号: 4

期望的包: 5

接收端确认序列号: 5

C:\Users\zc\Desktop\Client\UDP可靠传输

已发送第一次握手请求SYN

接收到服务端二次握手SYN_ACK

已发送第三次握手ACK

连接到服务端

请输入窗口大小（默认10）: 17

请输入发送的文件名

1. jpg

数据包总数: 1

已发送数据包: 0

接收到ACK: 0

窗口队列头的序号: 0

窗口队列的大小: 1

接收到所有ACK，发送完毕

文件名传输完毕

数据包总数: 186

已发送数据包: 0

已发送数据包: 1

已发送数据包: 2

已发送数据包: 3

已发送数据包: 4

C:\Users\zc\Desktop\Server\UDP可靠传输3-3-Server.exe

期望的包: 77
接收端确认序列号: 83
缓存数据包83
缓存包的个数: 6

期望的包: 77
接收端确认序列号: 84
缓存数据包84
缓存包的个数: 7

期望的包: 77
接收端确认序列号: 85
缓存数据包85
缓存包的个数: 8

期望的包: 77
接收端确认序列号: 77
缓存区已交付最后一个数据包
信息接收成功
文件接收成功

接收到的包总数: 0
开始断开连接
成功接收第一次挥手消息
成功发送第二次挥手
挥手成功, 断开连接
请按任意键继续

C:\Users\zc\Desktop\Client\UDP可靠传输3-3-Client.exe

接收到ACK: 83
窗口队列头的序号: 77
窗口队列的大小: 9

接收到ACK: 84
窗口队列头的序号: 77
窗口队列的大小: 9

接收到ACK: 85
窗口队列头的序号: 77
窗口队列的大小: 9

超时, 重传数据包77
接收到ACK: 77
窗口队列头的序号: 77
窗口队列的大小: 9

接收到所有ACK, 发送完毕
1.jpg文件传输完毕
传输时间: 15
吞吐量: 990.588 kbps
丢包率: 21%

发包总次数: 239
进行二次挥手ing
已发送第一次挥手
接收第二次挥手
挥手成功, 断开连接

Router

路由器IP: 127.0.0.1 服务器IP: 127.0.0.1
端口: 4003 服务器端口: 4003
丢包率: 5% 延时: 1
确定 修改

日志


count:7.
Delay 1 ms.
count:8.
Delay 1 ms.
count:9.
Delay 1 ms.
count:10.
Delay 1 ms.
count:11.

Server

1.jpg
UDP可靠传输3-3-Server.exe

1.jpg

JPG 文件

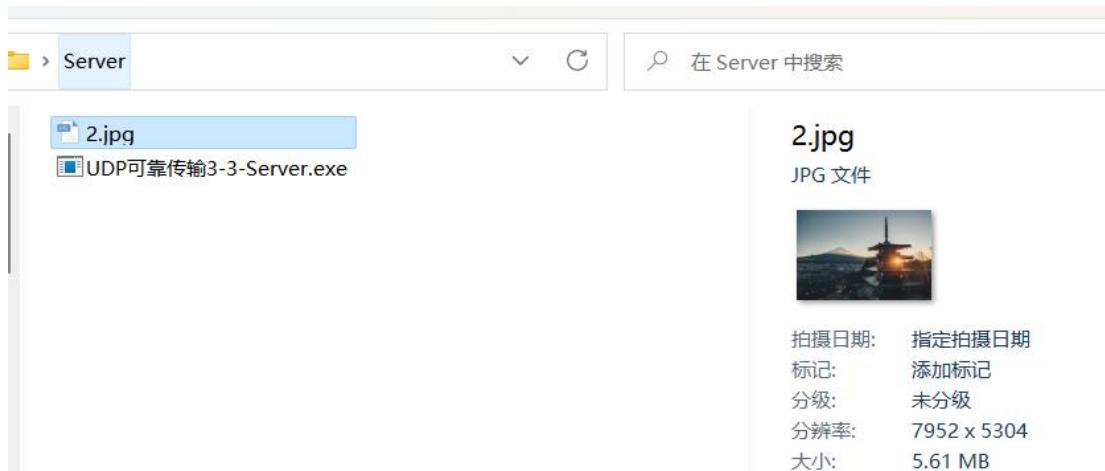


拍摄日期: 指定拍摄日期
标记: 添加标记
分级: 未分级
分辨率: 3840 x 2560
大小: 1.76 MB

2. jpg:

C:\Users\zc\Desktop\Server\	C:\Users\zc\Desktop\Client\UDP可靠传输3-3
服务端正在运行-----ip: 1	已发送第一次握手请求SYN
接收到客户端第一次SYN	接收到服务端二次握手SYN_ACK
已发送SYN_ACK	已发送第三次握手ACK
接收到客户端ACK	连接到服务端
已完成三次握手	
开始接收文件	请输入窗口大小（默认10）： 13
期望的包：0	请输入发送的文件名
接收端确认序列号：0	2.jpg
已接收最后一个数据包	数据包总数：1
文件名：2.jpg	已发送数据包：0
期望的包：0	接收到ACK：0
接收端确认序列号：0	窗口队列头的序号：0
	窗口队列的大小：1
期望的包：1	接收到所有ACK，发送完毕
接收端确认序列号：1	文件名传输完毕
期望的包：2	数据包总数：590
接收端确认序列号：2	已发送数据包：0
期望的包：3	已发送数据包：1
接收端确认序列号：3	已发送数据包：2
期望的包：4	已发送数据包：3
接收端确认序列号：4	
期望的包：5	
接收端确认序列号：5	

C:\Users\zc\Desktop\Server\	C:\Users\zc\Desktop\Client\UDP可靠传输3-3-Client.exe	Router
期望的包：80	超时，重传数据包80	路由器IP: 127.0.0.1 服务器IP: 127.0.0.1
接收端确认序列号：86	接收到ACK：80	端口: 4003 服务器端口: 4002
缓存数据包86	窗口队列头的序号：80	丢包率: 5% 延时: 1 ms
缓存包的个数：6	窗口队列的大小：10	<input type="button" value="确定"/> <input type="button" value="修改"/>
期望的包：80	超时，重传数据包88	日志
接收端确认序列号：87	接收到ACK：88	count:8.
缓存数据包87	窗口队列头的序号：88	Delay 1 ms.
缓存包的个数：7	窗口队列的大小：2	count:9.
期望的包：80	超时，重传数据包89	Delay 1 ms.
接收端确认序列号：80	接收到ACK：89	count:10.
	窗口队列头的序号：89	Delay 1 ms.
期望的包：88	窗口队列的大小：1	count:11.
接收端确认序列号：88	接收到所有ACK，发送完毕	Delay 1 ms.
	2.jpg文件传输完毕	count:12.
期望的包：89	传输时间：53	
接收端确认序列号：89	吞吐量：890.34 kbps	
已接收最后一个数据包	丢包率：21%	
信息接收成功		
文件接收成功		
接收到的包总数：0	发包总次数：755	
开始断开连接	进行二次挥手ing	
成功接收第一次挥手消息	已发送第一次挥手	
成功发送第二次挥手	已发送第一次挥手	
挥手成功，断开连接	接收第二次挥手	
请按任意键继续...	挥手成功，断开连接	
	请按任意键继续...	



3. jpg:

```
C:\Users\zc\Desktop\Client\UDP可靠传输3-3-Client.exe 服务端正在运行-----ip: 127.0.0.1, 端口: 4002
已发送第一次握手请求SYN 接收到客户端第一次SYN
接收到服务端二次握手SYN_ACK 已发送SYN_ACK
已发送第三次握手ACK 接收到客户端ACK
连接到服务端 已完成三次握手
开始接收文件
请输入窗口大小（默认10）： 16 期望的包： 0
请输入发送的文件名 接收端确认序列号： 0
3. jpg 已接收最后一个数据包
数据包总数： 1 文件名： 3. jpg
已发送数据包： 0
期望的包： 0
接收到ACK： 0 接收端确认序列号： 0
窗口队列头的序号： 0
窗口队列的大小： 1 期望的包： 1
接收端确认序列号： 1
接收所有16个数据包并发送ACK
```

C:\Users\zc\Desktop\Client\UDP可靠传输3-3-Server.exe

已发送数据包: 96

接收到ACK: 96

窗口队列头的序号: 93

窗口队列的大小: 4

超时, 重传数据包93

接收到ACK: 93

窗口队列头的序号: 93

窗口队列的大小: 4

接收到所有ACK, 发送完毕

3.jpg文件传输完毕

传输时间: 94

吞吐量: 1018.64 kbps

丢包率: 21%

发包总次数: 1533

进行二次挥手ing

已发送第一次挥手

已发送第一次挥手

已发送第一次挥手

接收第二次挥手

挥手成功, 断开连接

请按任意键继续. . .

期望的包: 80

接收端确认序列号: 95

缓存数据包95

缓存包的个数: 14

期望的包: 80

接收端确认序列号: 80

期望的包: 93

接收端确认序列号: 96

缓存数据包96

缓存包的个数: 3

期望的包: 93

接收端确认序列号: 93

缓存区已交付最后一个数据包

信息接收成功

文件接收成功

接收到的包总数: 0

开始断开连接

成功接收第一次挥手消息

成功发送第二次挥手

挥手成功, 断开连接

请按任意键继续. . .

Router

路由器IP: 127 . 0 . 0 . 1

端口: 4003

丢包率: 5 %

确定

count:4.
Delay 1 ms.
count:5.
Delay 1 ms.
count:6.
Delay 1 ms.
count:7.
Delay 1 ms.
count:8.

3.jpg

UDP可靠传输3-3-Server.exe

3.jpg

JPG 文件



拍摄日期: 指定拍摄日期

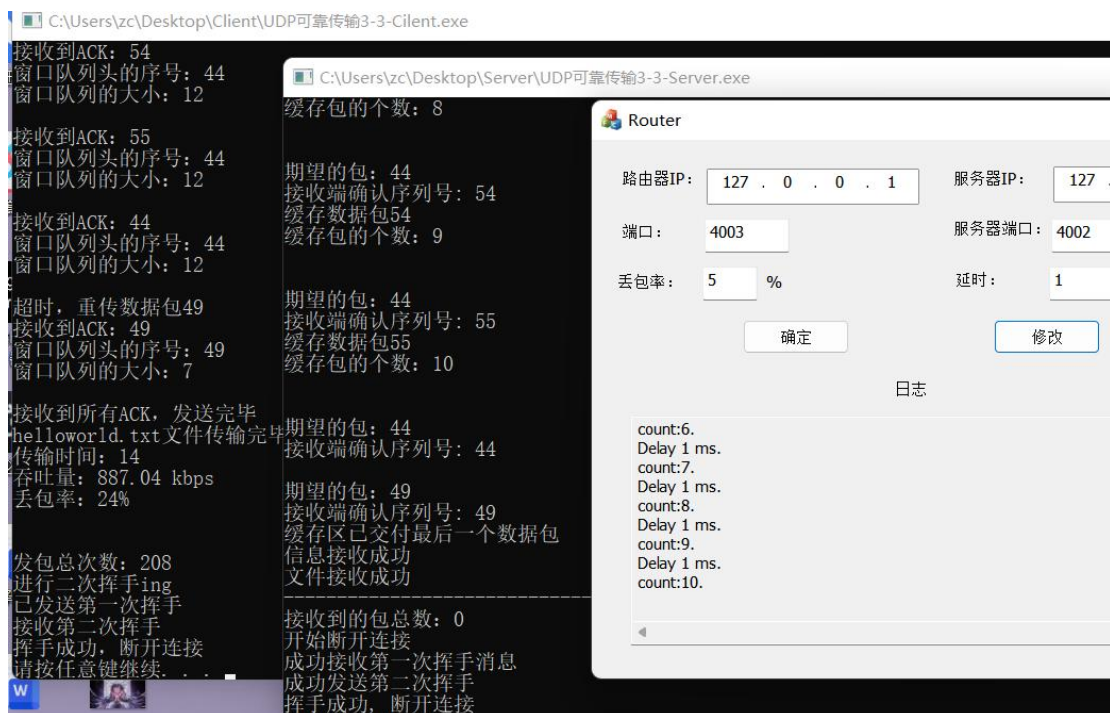
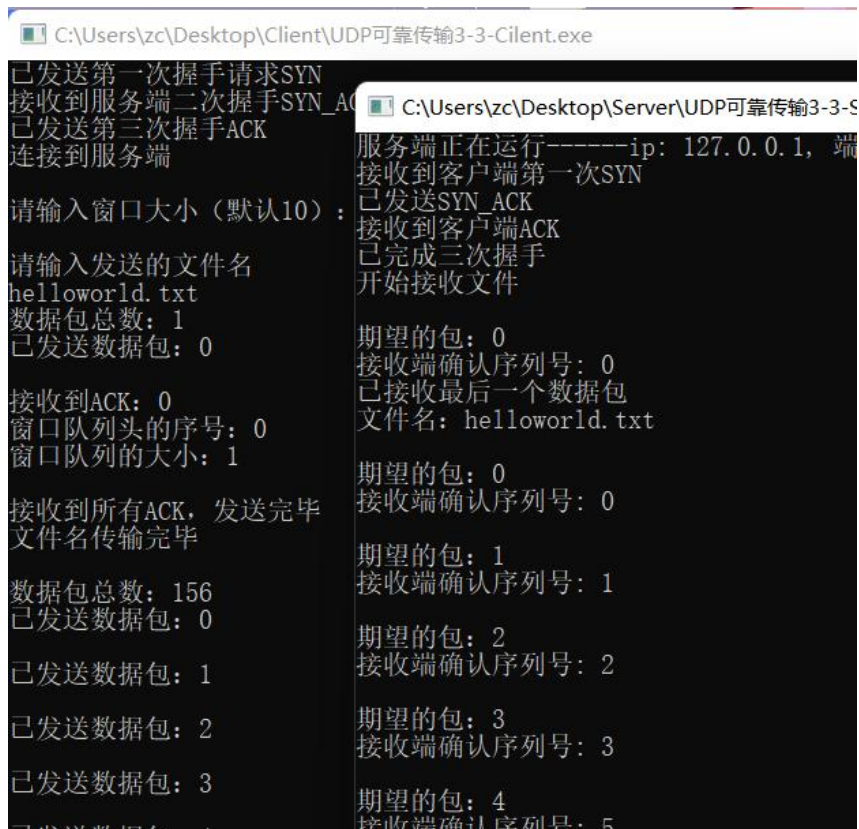
标记: 添加标记

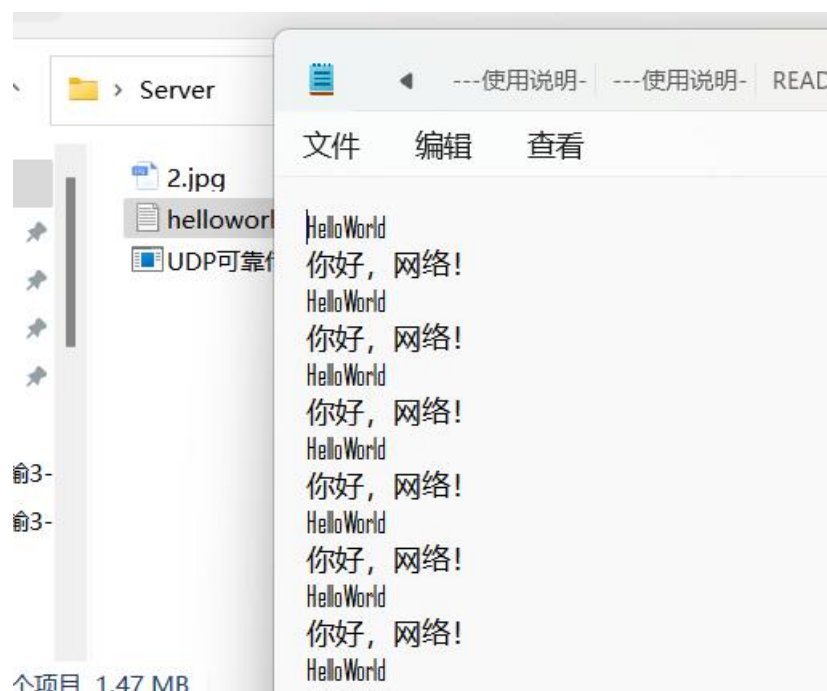
分级: 未分级

分辨率: 10044 x 7171

大小: 11.4 MB

4. helloworld.txt:





五、思考与总结

实现了 UDP 可靠传输的滑动窗口机制和选择确认机制，复习到课上的理论知识点。

中途有不懂的地方和概念模糊的地方，去翻书找到了。

使用了多线程。