南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验 3-3

基于 UDP 服务设计可靠传输协议并编程实现

杨浩甫　2113824

年级：2021 级

专业：计算机科学与技术

指导教师：张建忠、徐敬东

2023 年 12 月 15 日

目录

一、 实验基本描述

在实验 3-1 中，利用了数据报套接字在用户空间实现了面向连接的可靠数据传输。这包括建立连接、差错检测以及确认重传等功能并使用停等机制进行流量控制。本次实验则在此基础上做了改进，引入了基于 SR 的滑动窗口流量控制机制，并采用了选择确认的方式。

二、 实验具体工作

1. 在 3-1 的基础上，增加滑动窗口功能
2. 将 3-2 的 GBN 协议修改为 SR 协议
3. 采用选择确认机制
4. 对不同情况下的传输时间和吞吐率进行探究。

(一) 协议设计

1. 报文格式

本次实验采用和之前一样的报文设计：

字段	位置 (比特位)	大小 (比特位)
flag	0-31	32
seq	32-47	16
ack	48-63	16
len	64-95	32
num	96-127	32
checksum	128-143	16
data	144-1175	8192

表 1: 报文结构

32 - 8	7	6	5	4	3	2	1
	RE	EXT	ACK	END	START	FIN	SYN

表 2: flag 标志位设计

2. 流水线设计

考虑到 rdt3.0 停等协议的性能问题，未能够充分的提高链路利用率。

故本实验 3-3 发送端和接收端都将采用流水线协议进行性能优化：在确认未返回之前允许发送或接收多个分组。

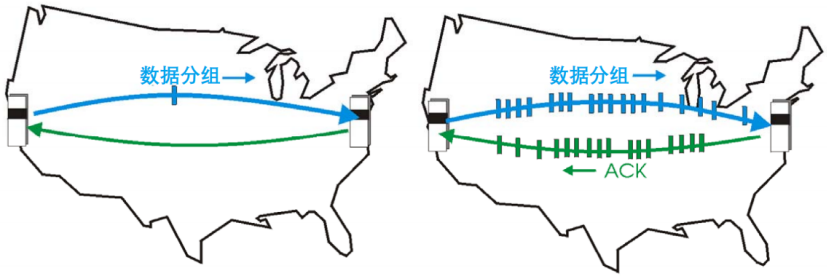

图 1: 流水线协议

3. 滑动窗口机制

在实验 3-3 中，采用了基于滑动窗口的流量控制机制。这种方法涉及发送方和接收方各自拥有缓存数组，其中发送方缓存包含已发送且得到确认的包序号、已发送但未确认的包序号以及未发送的包序号。接收方的缓存则包括已接收的包序号、正在接收的包序号和未接收的包序号。这两个数组各自有两个扫描指针，形成一个窗口。窗口的大小为 N，允许一次性发送 N 个报文段，实现了流水线机制，以提升传输性能。窗口随着协议运行的进程在序列号空间内向前滑动，允许新的数据进入发送窗口并发送到接收方。与 3-2 的 GBN 协议有所不同，本次实验接收端的窗口与发送端的窗口大小相同。

具体来说，窗口指代允许使用的序列号范围，其大小为 N，随着协议的运行在序列号空间内向前滑动。窗口被划分为左边界、发送边界和右边界，大小固定。左边界左侧是已经发送并得到确认的数据，左边界到发送边界是已发送但未得到确认的数据，发送边界到右边界是等待发送的数据，右边界右侧是不可发送的数据。

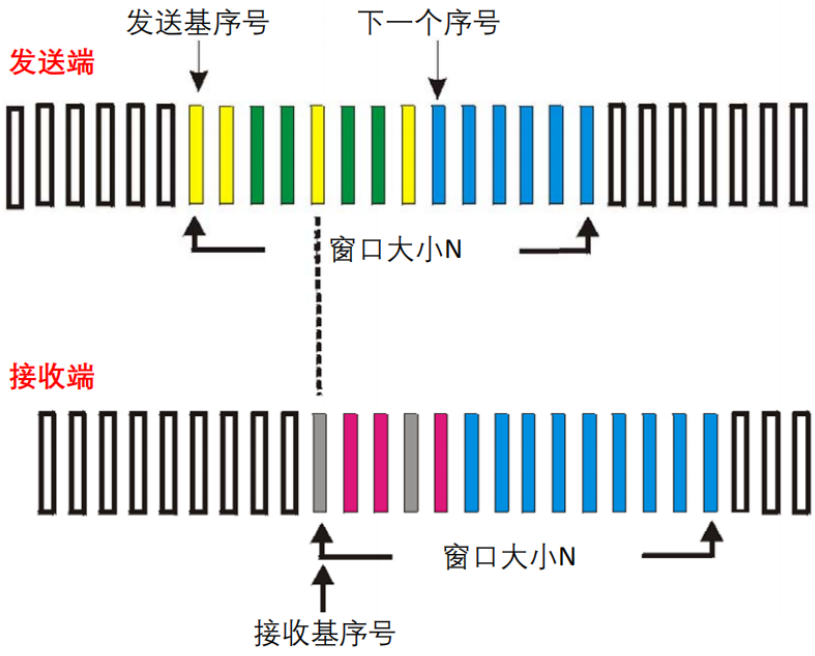

图 2: SR 滑动窗口

4. SR 协议

SR（Selective Repeat）是另一种可靠数据传输协议，与 GBN 类似，但在处理丢失的分组时有所不同。SR 协议允许发送方在未收到确认的情况下发送多个分组，接收方可以选择性地确认丢失的分组并请求其重传，而不是像 GBN 那样一旦发现丢失就重传所有未确认的分组。

关键特点包括:

- 窗口大小：类似于 GBN，发送方和接收方都有窗口大小限制，表示可发送和可接收的分组数量。
- 选择性重传：接收方可以选择性地确认收到的分组，并对丢失的分组请求重传，而不是简单地确认最大连续序列号之前的所有分组。
- 接收缓存：接收方会缓存已收到的分组，以便在需要时重排序或重传丢失的分组。

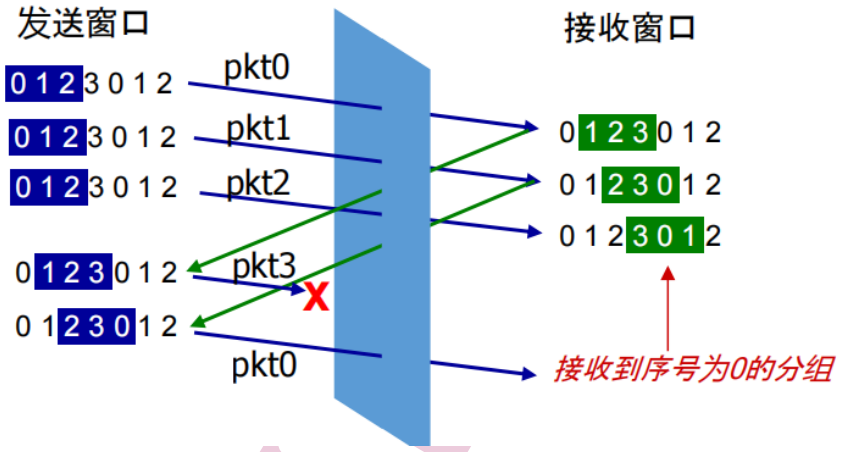

图 3: SR 协议交互示例

5. 选择确认

选择确认是一种用于可靠数据传输的确认机制，与累计确认有所不同。在选择确认中，**发送端不仅会确认已经按序收到达的最大序列号的报文，还会单独确认接收到的每个报文，即使它们不是按序到达的**。

当接收方收到报文时，它会发送一个 ACK 确认该报文的到达，并将确认号设置为已接收的最大连续序列号。同时，如果有任何丢失的报文到达，接收方会在 ACK 中包含这些丢失报文的序列号，告知发送方需要重传这些报文。

选择确认允许接收方灵活地确认单独的报文，而不仅限于确认最大连续序列号之前的所有报文。这种灵活性意味着即使中间某些报文丢失，接收方仍然可以及时地确认其他已收到的报文，从而促使发送方只重传丢失的报文，而不是整个窗口内的所有未确认报文。

这个机制对于减少不必要的重传以及提高网络利用率非常有用。然而，选择确认也可能导致发送方收到不按序到达的 ACK，因此发送方需要维护一个记录，以便根据接收方的选择性确认进行相应的重传。这种方式可以更好地适应网络中的丢包情况，提高了数据传输的效率和可靠性。

6. 超时重传

在选择重复（Selective Repeat）协议中，超时重传是确保数据可靠传输的重要机制之一，用于处理网络中的丢包情况，确保数据的可靠传输。当发送方发送数据并启动定时器以等待接收端

的确认，但在超时时间内未收到确认时，超时重传机制就会被触发。

超时重传的过程大致如下：

1. **数据发送：**发送方将数据分组发送到接收方，并启动定时器。在发送数据后，发送方会等待接收端的确认。

2. **超时计时器：**发送方设置一个定时器，在发送数据后开始计时。如果在设定的超时时间内没有收到对应的确认，定时器将会超时。

3. **超时处理：**当定时器超时时，发送方会认为之前发送的数据丢失了。因此，发送方会重新发送这些未确认的数据包，而不是等待接收端的确认。

4. **重传数据：**发送方根据超时重传机制重新发送丢失的数据包。这样可以确保接收端能够收到丢失的数据，并且能够在接收到数据后发送确认，从而维持协议的正常运行。

(二) 具体实现

1. 窗口设置

首先，我在接收端和发送端定义了一些与滑动窗口协议相关的参数和数据结构。

- WINDOW_SIZE：表示发送（接收）窗口的大小，即允许同时发送但未收到确认的数据包数量。
- message window[WINDOW_SIZE]：这是一个数组，用于存储发送窗口内的数据包。在滑动窗口协议中，数据包按顺序发送，但并非所有数据包都已收到确认。这个数组用于存储已发送但未收到确认的数据包。
- base：是滑动窗口的基序号，代表发送（接收）窗口的起始位置。
- nextSeqNum：表示下一个待发送的序号，即当前窗口中最后一个未发送的数据包序号加一。
- acked[WINDOW_SIZE]：这是一个布尔数组，用于标记发送窗口内的消息是否被确认；在接收端代表当前数据包已经被接收。
- timeout：定义了超时时间，表示在等待确认时允许的最大时间。
- packetTimes[WINDOW_SIZE]：这个数组记录了每个数据包的发送时间，用于计算超时。
- buffer: 这个 map 结构缓存接收端收到的数据包。

滑动窗口设置

```
const int WINDOW_SIZE = 10; // 窗口大小
message window[WINDOW_SIZE]; // 发送窗口
int base = 0; // 发送窗口的基序号
int nextSeqNum = 0; // 下一个待发送的序号
bool acked[WINDOW_SIZE] = { false }; // 标记发送窗口内的消息是否被确认
std::chrono::milliseconds timeout(800); // 800毫秒超时时间
int packetTimes[WINDOW_SIZE];// 记录每个数据包的发送时间

 std::map<u_short, message> buffer; // 存储接收到的数据包
```

2. 发送端

接收线程：

我在发送端利用一个线程函数处理接收 ACK，并完成选择确认。主要功能有：

1. **接收 ACK 消息：**使用非阻塞方式接收来自接收端的消息，包括 ACK 和拓展的 ACK（在此示例中是 isACK() 和 isEXT() 的检查）。非阻塞接收确保即使没有消息到达，线程也不会因等待消息而被阻塞。

2. **解析 ACK 信息：**从接收到的消息中提取确认号（ackNum），表示接收到的报文序号。

3. **选择确认：**这里根据收到的 ACK 确认了哪些分组，将对应的位置置为已确认（false）。

4. **窗口更新：**使用接收到的确认号更新窗口状态，并根据基准值（base）移动窗口。通过循环，根据接收到的 ACK 号，将对应的位置设置为已确认状态，并更新基准值，使窗口向前滑动。

4. **输出窗口状态：**在控制台输出当前窗口的状态，用不同颜色标识已确认、未确认和未发送的分组。这有助于调试和可视化了解窗口内的数据包状态。

5. **检查终止消息：**如果收到了结束消息（isEND()），则退出线程。

接收文件线程

```
void receiveACK() {
    int iMode = 0; //1: 非阻塞, 0: 阻塞
    ioctlsocket(client, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
    while (isRunning) {
        message ackMsg = recvmessage(client, serveraddr);
        if (ackMsg.isACK()&& ackMsg.isEXT()) {
            std::lock_guard<std::mutex> lock(mtx); // 在代码块中自动上锁, 出
                作用域时自动释放
            int ackNum = ackMsg.ack;
            acked[(ackNum-1) % WINDOW_SIZE] = false;
            while (base < ackNum && !acked[base % WINDOW_SIZE]) {
                (base)++;
            }
            cout << endl;
            // 输出当前窗口占用情况
            cout << "接收数据包, 当前窗口占用情况: " << endl;
            int i = 0;
            for (int j = base; j < base + WINDOW_SIZE; ++i, j++) {
                if (acked[j % WINDOW_SIZE] && j < nextSeqNum) {
                    SetColor(14, 0);
                    cout << "[␣" << i << "␣]" << "␣";
                }
                else {
                    if (j < nextSeqNum) {
                        SetColor(13, 0);
                        cout << "[␣" << i << "␣]" << "␣";
                    }
                    else {
                        SetColor(15, 0);
                        cout << "[␣" << i << "␣]" << "␣";
                    }
                }
```

```
            }
            cout << endl<<endl;
        }
        if (ackMsg.isEND()) {
            ExitThread(TRUE);
        }
    }
    return;
}
```

发送线程：

我通过一个滑动窗口发送文件。

首先创建一个消息（message），将数据分块读取到消息中，然后发送该消息到服务器端。发送完成后，记录已发送的数据信息和时间，并更新序号，以便发送下一个数据块。发送的数据被存储在一个发送窗口内，窗口大小为 WINDOW_SIZE。

传输文件

```
while (filelen || base != nextSeqNum) {
    std::lock_guard<std::mutex> lock(mtx);
    if (nextSeqNum < base + WINDOW_SIZE && filelen > 0) {
        message msg;
        msg.seq = nextSeqNum;
        msg.len = min(filelen, 1024);
        inFile.read(msg.data, msg.len);
        filelen -= msg.len;
        msg.setchecksum();
        SetColor(15, 0);
        sendData(client, serveraddr, msg);
        cout << "Client:发出" << msg.seq << "号数据包" << endl;
        cout << "Client:此时已发送数据右端在窗口的位置:␣" << nextSeqNum %
            WINDOW_SIZE << endl;

        msg.output();

        acked[nextSeqNum % WINDOW_SIZE] = true;
        packetTimes[nextSeqNum % WINDOW_SIZE] = clock(); // 记录发送时间
        nextSeqNum++;
    }
```

超时重传线程

当某个数据包超过了设定的超时时间仍未收到确认时，会触发超时重传机制。代码中使用了 packetTimes 数组来记录发送每个数据包的时间。先遍历窗口中的数据包，检查每个数据包的发送时间，如果发现某个数据包超时且未被确认，则会重新发送该数据包。这有助于保证数据的可靠传输，特别是在网络中存在丢包或延迟的情况下。

超时重传

```
void timeoutResend() {
    while (isRunning) {
```

```
        // 超时重传检查逻辑
        for (int i = base; i < nextSeqNum; ++i) {
            int index = i % WINDOW_SIZE;
            // 判断单个数据包是否超时
            if (((clock() - packetTimes[index]) > timeout.count()) &&
                timeused[i]) {
                std::lock_guard<std::mutex> lock(mtx); // 在代码块中自动上
                    锁，出作用域时自动释放
                SetColor(11, 0); // 红色文本，蓝色背景
                cout << "Client:␣超时重传␣" << window[index].seq << "␣号数据
                    包" << endl;
                sendmessage(client, serveraddr, window[index]);
                packetTimes[index] = clock(); // 重新记录发送时间
            }
        }
        // 延时一段时间再次检查
        std::this_thread::sleep_for(std::chrono::milliseconds(500)); // 假设
            延时100毫秒
    }
}
```

3. 接收端

函数 ReceiveFile 主要用于在服务器端接收来自客户端的文件数据。其核心部分是通过套接字 serverSocket 接收数据，并对接收到的数据包进行处理。

首先我初始化了一些变量：

- expectedSeq 表示期望的初始序列号。
- last_ack 记录最后确认的序列号。
- 一个窗口大小 WINDOW_SIZE 的数组 acked，用于标记已确认的数据包。
- buffer 用来缓存接收到的数据包。

然后开始处理接收到的数据包，我将处理逻辑放在一下无限循环 `while(1)`，用于持续接收消息，并根据不同的条件执行相应的操作。

1. **接收消息：**使用 `recvmessage` 函数从 `serverSocket` 接收消息 `msg`。
2. **检查消息类型：**

- 如果消息没有内容（`isEXT()` 为假），则跳过这次循环，继续接收下一条消息。
- 如果消息是结束类型（`isEND()` 为真），则向客户端发送确认消息，并输出成功接收文件的信息。随后关闭文件流，设置非阻塞模式，并调用 `ReceiveName` 函数进行下一个文件的接收。

3. **数据包处理：**

- 如果收到的数据包序号在窗口范围内，并且未被确认过，则进行处理：
  - 随机确定是否模拟丢包。

  - 存储接收到的数据包。将其缓存在 buffer 中。
  - 发送对应的 ACK 消息给客户端，确认接收到的数据包。
  - 输出当前窗口的占用情况。
  - 如果收到的序号与基准值相同，则处理窗口内连续已确认的数据包，写入文件，并滑动窗口。
- 如果收到的数据包序号在错误的区间（不在窗口范围内但是在基准值之前的窗口范围内），则发送空的 ACK 消息。

4. **其他情况处理：**由于窗口长度必须小于或等于序号空间的一半，因此可以将接收窗口接收到的可能的文件序号空间以 base 为界分为两部分：$[base, base-1]$ $[base, base+N-1]$，所以若出现在第一个区间，会进行“发送对应的 ACK，不做处理”的操作。对于其他区间的数据包不进行处理。

接收文件

```
while (1) {
    SetColor(15, 0);
    msg = recvmessage(serverSocket, clientaddr);

    if (!msg.isEXT()) {
        continue;
    }

    if (msg.isEND()) {
        message ackMsg;
        ackMsg.setACK();
        ackMsg.ack = msg.ack + 1;
        ackMsg.setEND();
        sendmessage(server, clientaddr, ackMsg);
        SetColor(13, 0);
        cout << "Server:␣接收文件" << receivedFilePath << "成功!!" <<
            endl << endl;
        cout << "************************************************" <<
            endl;
        outFile.close();
        outFile.clear();
        int iMode = 0; //1: 非阻塞, 0: 阻塞
        ioctlsocket(server, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
        SetColor(15, 0);
        return ReceiveName(server, clientaddr);
    }

    else if (msg.seq >= base && msg.seq < base + WINDOW_SIZE) {

        double random_value = (double)rand() / RAND_MAX;
        if (random_value < loss_rate) {
            SetColor(11, 0);
            cout << endl;
```

```
32                  cout << "Server:␣模拟丢包，未发送数据包" << endl;
33                  cout << endl;
34                  //sendmessage(server, serveraddr, message()); // 返回空消息
35                  continue;
36              }
37
38              if (msg.seq >= base && msg.seq < base + WINDOW_SIZE) {
39                  buffer[msg.seq] = msg; // 存储接收到的数据包
40              }
41
42              if (acked[msg.seq % WINDOW_SIZE]) {
43                  // 如果 ACK 对应的数据包已确认过，不做处理
44                  cout << "[Server:]已接受过"<< msg.seq<<"号数据包" << endl;
45                  continue;
46              }
47
48              // 收到正确的数据包，发送 ACK
49              message ackMsg;
50              ackMsg.setACK();
51              ackMsg.ack = msg.seq + 1;
52              last_ack = ackMsg.ack;
53              acked[msg.seq % WINDOW_SIZE] = true;
54
55              sendmessage(server, clientaddr, ackMsg);
56              cout << endl;
57              msg.output();
58              cout << "Server:␣发送确认收到的数据包(对应的ack)" << endl;
59
60              // 输出当前窗口占用情况
61              cout << "当前窗口占用情况: " << endl;
62              int i = 0;
63              for (int j = base; j <base + WINDOW_SIZE; ++i,j++) {
64                  if (acked[j % WINDOW_SIZE]) {
65                      SetColor(13, 0);
66                      cout << "[␣" << i << "␣]" << "␣";
67                  }
68                  else {
69                      if (j <= msg.seq) {
70                          SetColor(14, 0);
71                          cout << "[␣" << i << "␣]" << "␣";
72                      }
73                      else {
74                          SetColor(15, 0);
75                          cout << "[␣" << i << "␣]" << "␣";
76                      }
77                  }
78              }
79
```

```
80              cout << endl;
81
82              if (msg.seq == base) {
83                  while (acked[base % WINDOW_SIZE]) {
84                      auto it = buffer.find(base);
85                      if (it != buffer.end()) {
86                          message nextMsg = it->second;
87                          cout << "写入: " << nextMsg.seq << endl;
88                          outFile.write(nextMsg.data, nextMsg.len);
89                          num--;
90                          expectedSeq++;
91                          acked[base % WINDOW_SIZE] = false;
92                          buffer.erase(it);
93                          base++;
94                      }
95                      else {
96                          // 没有找到对应序号的数据包，可能出现了丢包或其他异常
                               情况
97                          break;
98                      }
99                  }
100                 // 滑动窗口
101                 while (acked[base % WINDOW_SIZE]) {
102                     acked[base % WINDOW_SIZE] = false;
103                     base ++;
104                 }
105             }
106         }
107         else if (msg.seq >= base - WINDOW_SIZE && msg.seq < base - 1) {
108             // 收到错误区间的数据包，发送 ACK
109             message ackMsg;
110             ackMsg.setACK();
111             ackMsg.ack = msg.seq + 1;
112             SetColor(15, 0);
113             cout << "[Server:]收到错误区间的数据包" << endl;
114             sendmessage(server, clientaddr, ackMsg);
115         }
116
117     }
```

三、 结果分析

(一) 传输日志

下面我将在窗口大小为 10，丢包率为 0.03 的条件下以 1.jpg 为例，来介绍传输日志。首先，我们来看建立连接和发送端打印窗口大小：

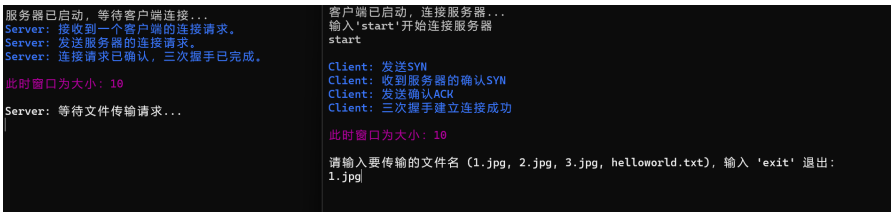

图 4: 传输准备

接下来是传输过程，接收端和发送端都打印传输日志，相较于 3-1 和 3-2 主要的改进是：

- 接收端接收到文件时，根据选择确认后的结果，完成窗口滑动与变化，并打印当前窗口（紫色：已缓存，白色：可接受，黄色：期待）。
- 发送端发送文件时，打印已发送数据包在窗口中的位置，用来反映当前窗口剩余空间。

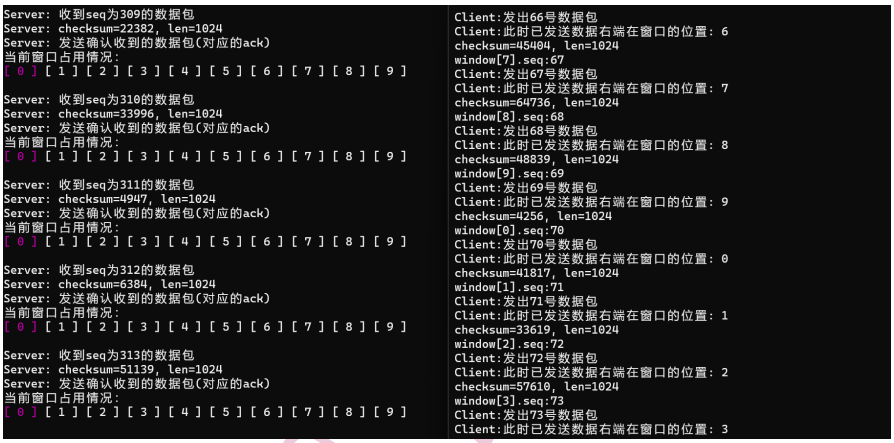

图 5: 传输过程

再来分析一下窗口移动和重传效果：

- 接收端由于丢包，未收到 334 号数据包，但是可以乱序接收其他数据包。

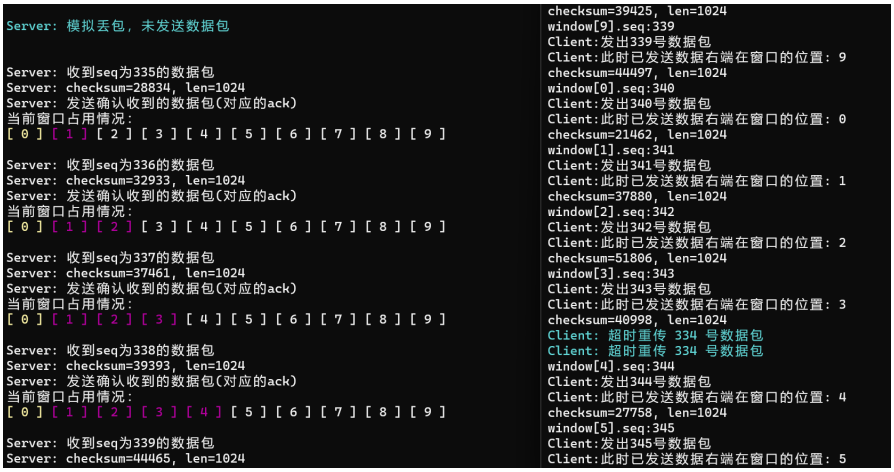

图 6: 窗口移动

- 发送端的 334 号数据包由于未被确认而超时重传：立刻将已发送未确认（334）数据包重发。

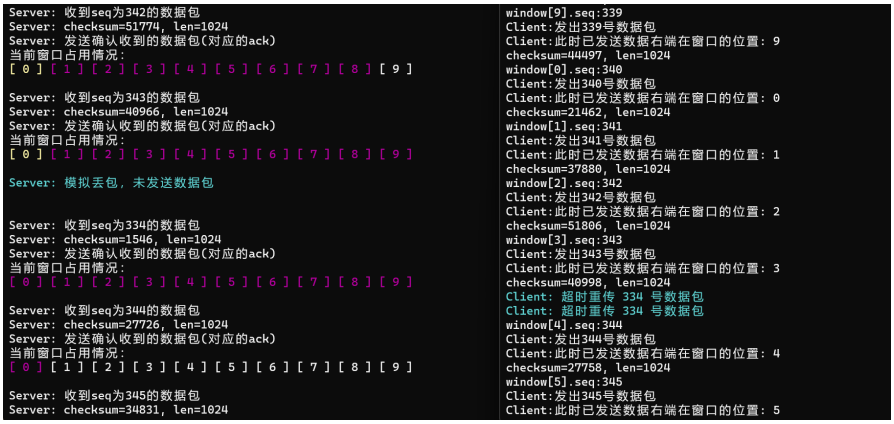

图 7: 超时重传

最后，文件接收成功，打印日志，并输出传输时间和吞吐率。之后是断开连接的日志。

图 8: 传输成功

(二) 文件传输结果

传输其他三个文件后，都可以接收到相应的文件：

图 9: 2.jpg

图 10: 3.jpg

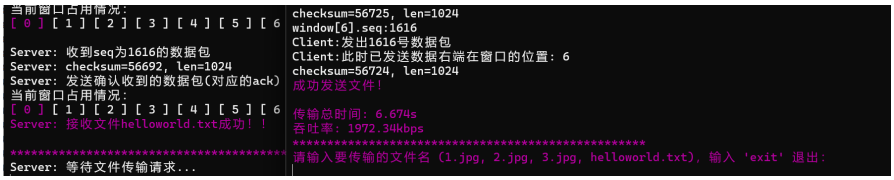

图 11: helloworld.txt

可以看到，四个文件都可以被 Server 端正确接收：

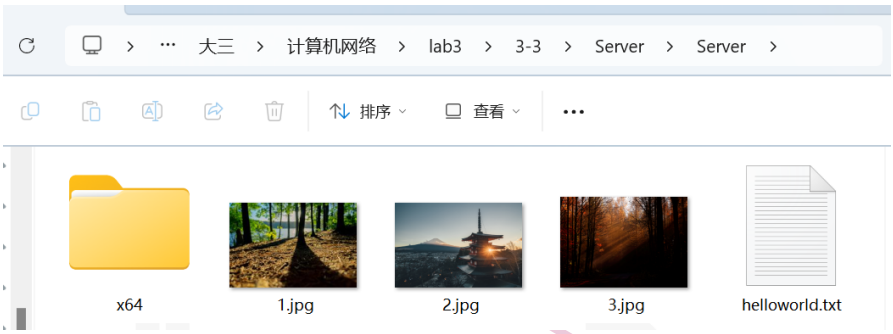

图 12: 传输效果

(三) 传输性能测试结果

我对四个文件都进行了传输性能测试，接下来展示在丢包率和延时为 0 的情况下的传输结果：

文件名	窗口大小	数据包数（个）	传输时间 (s)	吞吐率（kbps）
1.jpg	4	1813	2.374	5783
2.jpg	8	5760	9.204	5100.72
3.jpg	16	11688	21.709	4383.24
helloworld.txt	32	1616	3.509	3751.32

表 3: 传输文件统计

与 3-2 类似，为了探究滑动窗口大小对传输性能的影响，我在传输 1.jpg，丢包率为 0.01，超时时间为 0.3s 时，测试了 SR 协议下不同窗口大小下的传输性能：

窗口大小	传输时间 (s)	吞吐率（kbps）
1	21.681	681.108
4	13.109	1126.49
8	13.215	1117.45
16	12.744	1158.75

表 4: 1.jpg 传输数据

分析结果可知：

- SR 滑动窗口先对于依次传输，传输时间显著减少。但是当窗口大于 1 时，随着窗口大小改变，传输时间并没有很大的改变。这可能是因为我的超时时间较长导致一般只有窗口满之后才会造成超时重传。
- 吞吐率变化：从窗口大小 1 到 16，吞吐率接近提升一般。这表明滑动窗口使得网络的利用率提高，单位时间内传输的数据量也随之增加。但是更大的窗口对于吞吐率也没有显著提升。

四、 总结与反思

本次实验成功利用 SR 协议实现滑动窗口功能，对选择确认、超时重传等概念有了深入的理解，继续改进了多线程编程，实验完成度较高。但是也存在一定的问题，主要是超时时间的设定还需要更多的探索。