计算机网络实验报告

Lab3-2 基于 UDP 服务设计可靠传输协议

网络空间安全学院 信息安全专业

2112492 刘修铭 1063

https://github.com/lxmliu2002/Computer_Networking

一、实验内容

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、接收确认、超时重传等。采用基于滑动窗口的流量控制机制，接收窗口大小为 1，发送窗口大小大于 1，支持累积确认，完成给定测试文件的传输。

二、协议设计

（一）报文格式

在本次实验中，仿照 TCP 协议的报文格式进行了数据报设计，其中包括源端口号、目的端口号、序列号、确认号、消息数据长度、标志位、检测值以及数据包，其中标志位包括 FIN、CFH（是否为文件头部信息）、ACK、SYN 四位。

报文头部共 24Byte，数据段共 8168Byte，整个数据报大小为 8192Byte。

```
|0            15|16            31|
-----------------------------------
|            SrcPort              |
-----------------------------------
|            DstPort              |
-----------------------------------
|             Seq                 |
-----------------------------------
|             Ack                 |
-----------------------------------
|             Win                 |
-----------------------------------
|            Length               |
-----------------------------------
|      Flag     |      Check      |
-----------------------------------
|              Data               |
-----------------------------------

                Flag
|0  |1  |2  |3  |4              15|
```

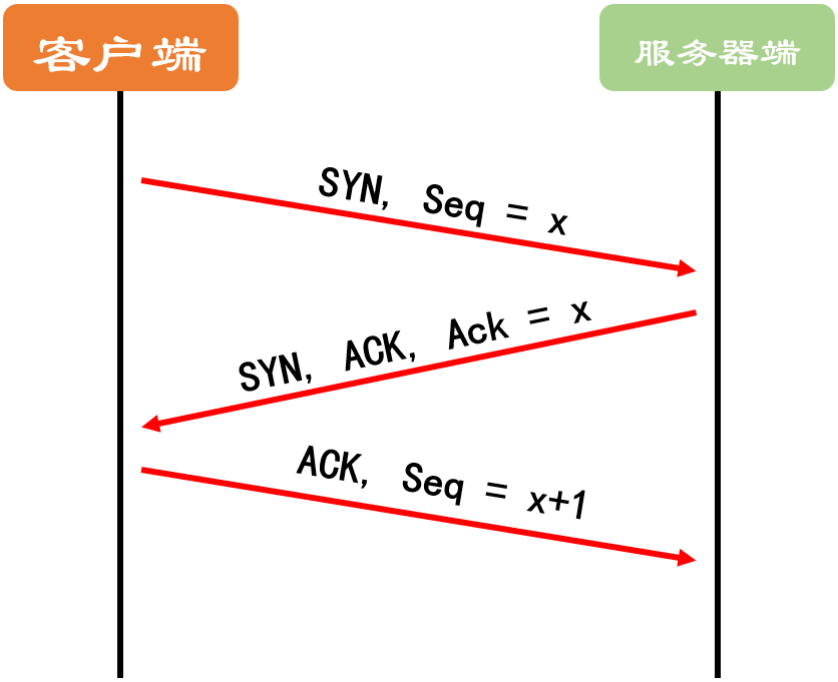
```
-----------------------------------
|FIN|CFH|ACK|SYN|                 |
-----------------------------------
```

（二）消息传输机制

本次实验对传统机制进行修改，确认消息的 Ack 为发送消息的 Seq。

1. 建立连接——三次握手

仿照 TCP 协议设计了连接的建立机制——三次握手，依旧使用停等机制，示意图如下：



2. 差错检测

为了保证数据传输的可靠性，本次实验仿照 UDP 的校验和机制设计了差错检测机制。对消息头部和数据的所有 16 位字求和，然后对结果取反。算法原理同教材，不再赘述。

3. 滑动窗口与累计确认

按照实验要求，在本次实验中，基于GBN流水线协议，使用固定窗口大小，实现了流量控制机制。

在 Defs.h 中定义了窗口大小 Windows_Size。在 Client_GBN.cpp 中，为了避免多线程导致的冲突，定义了基于原子操作的 Base_Seq 与 Next_Seq，用于标定窗口的位置。

发送端

- 发送端的发送缓冲区大小和窗口大小一致，即发送端最多一次性发送 Windows_Size 个报文。
- 通过两个指针来控制当前已经发送的报文序号和还未确认的报文序号。Base_Seq 指针表示的是当前已经确认的最大序列号，Next_Seq 指针表示的是当前已经发送的最大序列号，这之间的报文就是发送了但是还未被确认的部分。
- 每当发送端接收到回应的 ACK 的时候，需要判断一下当前回应的序号是否超过了 Base_Seq ，如果超过了 Base_Seq ，则说明从 Base_Seq 到回应序号这部分的报文已经被确认了。
- 此时可以向前滑动窗口，调整 Base_Seq 指针。而如果在一定时间内没有收到有效的 ACK，则重新发送从 Base_Seq 到 Next_Seq 之间的全部报文。

接收端

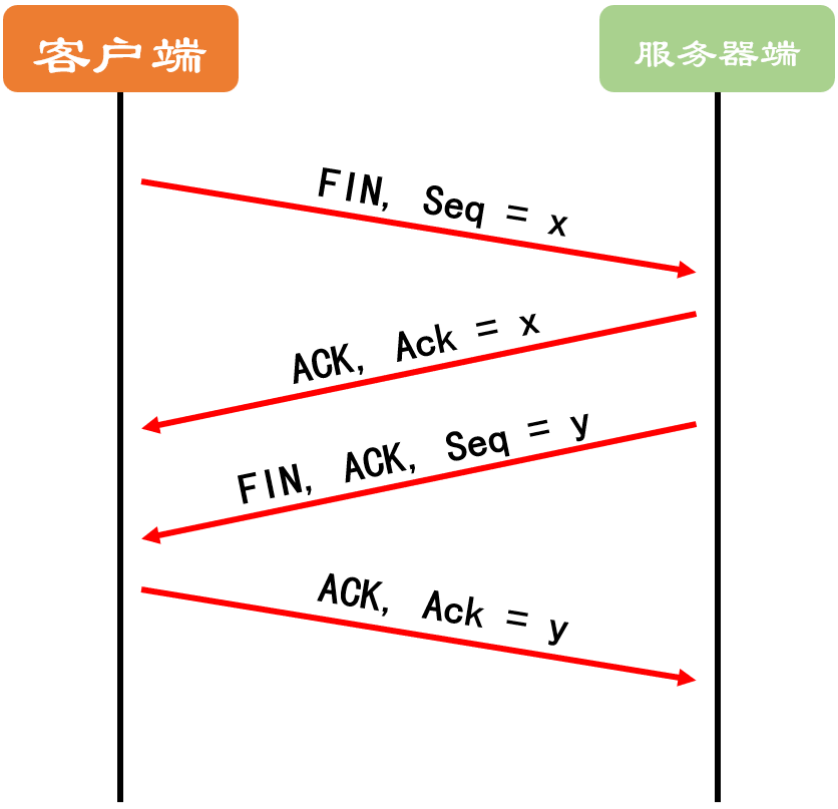
- 每次收到发送端发送来的报文的时候，判断当前的序号是否为自己期待的序号，如果相符则接收成功，窗口向后移动；如果不相符，则返回已经确认的最大序号。

4. 超时重传

- 本次实验中发送端的超时重传有两部分组成：
  - 参考快速重传的思想实现，即当接收到三个重复 Ack 时，重新发送当前窗口的所有数据报文；
  - 当发送端超过 `Wait_Time` 时间间隔未收到 ACK 报文时，也会进行重传。
- 接收端则设置了接收时间判定，如果距离上次接收到报文的时间间隔超过 `Wait_Time`，则向发送端发送三个相同的 ACK 报文，刺激其重新发送。

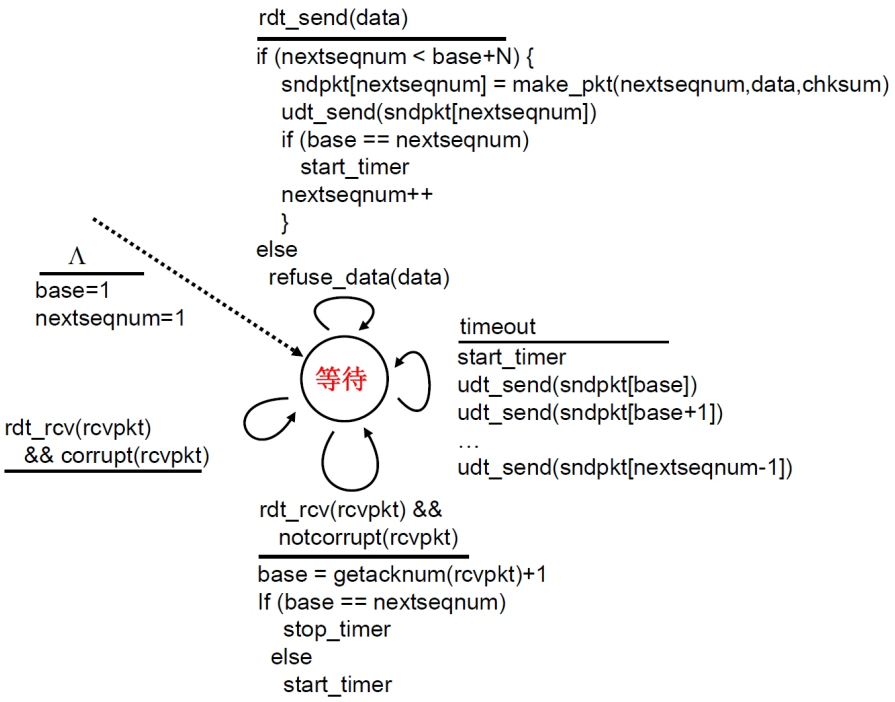
5. 断开连接——四次挥手（以发送端主动断开连接为例）

本次实验仿照 TCP 协议，设计了四次挥手断开连接机制，依旧使用停等机制，示意图如下：

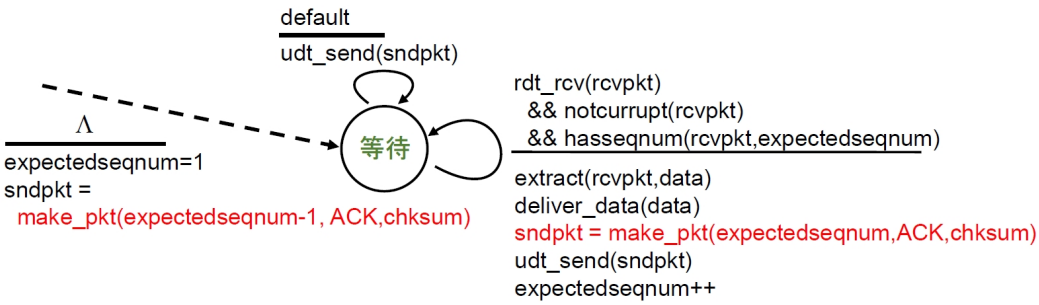


6. 状态机

（1）发送端



（2）接收端



三、代码实现

（一）协议设计

本次实验参考 oceanbase 设计，将头文件、宏定义、结构体等写入 `Defs.h` 文件中。

通过将标志位进行宏定义，便于后续使用。

```
#define FIN 0b1
#define CFH 0b10
#define ACK 0b100
#define SYN 0b1000
```

将协议报文包装成了 `Message` 结构体，并编写了系列函数，用来初始化结构体、设置标志位、差错检测等。

```
#pragma pack(1)
struct Message
{
    uint32_t SrcPort;
    uint32_t DstPort;
    uint32_t Seq;
    uint32_t Ack;
    uint32_t Length;
    uint16_t Flag;
    uint16_t Check;
    char Data[MSS];

    Message() : SrcPort(0), DstPort(0), Seq(0), Ack(0), Length(0), Flag(0), Check(0) { memset(this->Data, 0, MSS); }
    void Set_CFH() { this->Flag |= CFH; }
    bool Is_CFH() { return this->Flag & CFH; }
    void Set_ACK() { this->Flag |= ACK; }
    bool Is_ACK() { return this->Flag & ACK; }
    void Set_SYN() { this->Flag |= SYN; }
    bool Is_SYN() { return this->Flag & SYN; }
    void Set_FIN() { this->Flag |= FIN; }
    bool Is_FIN() { return this->Flag & FIN; }
    bool CheckValid();
    void Print_Message();
};
#pragma pack()
```

按照前面叙述，实现了校验位的设置与检测函数。其原理同理论课相同，不再赘述。

```
void Message::Set_Check()
{
    this->Check = 0;
    uint32_t sum = 0;
    uint16_t *p = (uint16_t *)this;
    for (int i = 0; i < sizeof(*this) / 2; i++)
    {
        sum += *p++;
        while (sum >> 16)
        {
            sum = (sum & 0xffff) + (sum >> 16);
        }
    }
    this->Check = ~(sum & 0xffff);
}
bool Message::CheckValid()
{
    uint32_t sum = 0;
    uint16_t *p = (uint16_t *)this;
```

```
    for (int i = 0; i < sizeof(*this) / 2; i++)
    {
        sum += *p++;
        while (sum >> 16)
        {
            sum = (sum & 0xffff) + (sum >> 16);
        }
    }
    return (sum & 0xffff) == 0xffff;
}
```

（二）初始化

发送端与接收端结构相同，此处以发送端为例进行说明。

在 `Defs.h` 中定义了相关套接字等信息，在 `Client.cpp` 中编写了 `Client_Initial` 函数，初始化发送端网络连接与套接字，并按照连接状态，适时进行错误检测，输出运行日志。

```
SOCKET ClientSocket;
SOCKADDR_IN ClientAddr;
string ClientIP = "127.0.0.1";
int ClientAddrLen = sizeof(ClientAddr);

bool Client_Initial()
{
    WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2)
    {
        perror("[Client] Error in Initializing Socket DLL!\n");
        exit(EXIT_FAILURE);
    }
    cout <<"[Client] "<< "Initializing Socket DLL is successful!\n";
    ClientSocket = socket(AF_INET, SOCK_DGRAM, 0);
    unsigned long on = 1;
    ioctlsocket(ClientSocket, FIONBIO, &on);
    if (ClientSocket == INVALID_SOCKET)
    {
        cout <<"[Client] "<< "Error in Creating Socket!\n";
        exit(EXIT_FAILURE);
        return false;
    }
    cout <<"[Client] "<< "Creating Socket is successful!\n";
    ClientAddr.sin_family = AF_INET;
    ClientAddr.sin_port = htons(Client_Port);
    ClientAddr.sin_addr.S_un.S_addr = inet_addr(ClientIP.c_str());

    if (bind(ClientSocket, (SOCKADDR *)&ClientAddr, sizeof(SOCKADDR)) ==
SOCKET_ERROR)
    {
        cout <<"[Client] "<< "Error in Binding Socket!\n";
        exit(EXIT_FAILURE);
```

```
        return false;
    }
    cout <<"[Client] "<< "Binding Socket to port "<< Client_Port<<" is successful!\n\n";
    RouterAddr.sin_family = AF_INET;
    RouterAddr.sin_port = htons(Router_Port);
    RouterAddr.sin_addr.S_un.S_addr = inet_addr(RouterIP.c_str());
    return true;
}
```

（三）建立连接——三次握手

1. 发送端

- 发送第一次握手消息，并开始计时，申请建立连接，然后等待接收第二次握手消息。
  - 如果超时未收到，则重新发送。
- 收到正确的第二次握手消息后，发送第三次握手消息。

```
void Connect()
{
    Message con_msg[3];
    // * First-Way Handshake
    con_msg[0].Seq = ++Seq;
    con_msg[0].Set_SYN();
    int re = Send(con_msg[0]);
    float msg1_Send_Time = clock();
    if (re)
    {
        con_msg[0].Print_Message();
        // * Second-Way Handshake
        while (true)
        {
            if (recvfrom(ClientSocket, (char *)&con_msg[1], sizeof(con_msg[1]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
            {
                con_msg[1].Print_Message();
                if (!(con_msg[1].Is_ACK() && con_msg[1].Is_SYN() && con_msg[1].CheckValid() && con_msg[1].Ack == con_msg[0].Seq))
                {
                    cout << "Error Message!" << endl;
                    exit(EXIT_FAILURE);
                }
                Seq = con_msg[1].Seq;
                break;
            }
            if ((clock() - msg1_Send_Time) > Wait_Time)
            {
                int re = Send(con_msg[0]);
                msg1_Send_Time = clock();
                if (re)
```

```
                {
                    SetConsoleTextAttribute(hConsole, 12);
                    cout << "!Time Out! -- Send Message to Router! -- First-Way Handshake" << endl;
                    con_msg[0].Print_Message();
                    SetConsoleTextAttribute(hConsole, 7);
                }
            }
        }
    }
    // * Third-Way Handshake
    con_msg[2].Ack = con_msg[1].Seq;
    con_msg[2].Seq = ++Seq;
    con_msg[2].Set_ACK();
    if (Send(con_msg[2]);) con_msg[2].Print_Message();
    cout << "Third-Way Handshake is successful!\n\n";
}
```

2. 接收端

- 接收正确的第一次握手消息，发送第二次握手消息，并开始计时，等待接收第三次握手消息。
  - 如果超时未收到，则重新发送。
- 接收到正确的第三次握手消息，连接成功建立。

此处代码结构与发送端基本一致，为避免报告冗长，不再展示。

（四）数据传输

为避免代码冗余，首先包装了消息发送函数 `Send`。

```
int Send(Message &msg)
{
    msg.SrcPort = Client_Port;
    msg.DstPort = Router_Port;
    msg.Set_Check();
    return sendto(ClientSocket, (char *)&msg, sizeof(msg), 0, (SOCKADDR *)&RouterAddr, RouterAddrLen);
}
```

1. 发送端

在 Defs.h 中定义了窗口大小 Windows_Size：

```
#define Windows_Size 20
```

为了避免多线程引起的数据读写冲突，定义了一系列基于原子操作的操作数，同时也定义了一个 mutex 用于控制输出。

```
atomic_int Base_Seq(1);
atomic_int Next_Seq(1);
atomic_int Header_Seq(0);
atomic_int Count(0);
int Msg_Num = 0;
atomic_bool Re_Send(false);
atomic_bool Finish(false);
mutex mtx;
atomic_int Send_Time(0);
```

编写了 `Send_Message` 函数用于数据发送。首先输入文件路径，按照路径寻找文件，获取到文件的名称及大小等信息，并以二进制方式读取文件数据。

```
strcpy(file_name, "");
size_t found = file_path.find_last_of("/\\");
string file_name = file_path.substr(found + 1);
ifstream file(file_path, ios::binary);
if (!file.is_open())
{
    cout <<"[Client] "<< "Error in Opening File!" << endl;
    exit(EXIT_FAILURE);
}
file.seekg(0, ios::end);
file_length = file.tellg();
file.seekg(0, ios::beg);
if(file_length > pow(2,32))
{
    cout<<"[Client] "<< "File is too large!" << endl;
    exit(EXIT_FAILURE);
}
```

接着编写了一个函数，用于多线程接收 Ack，并根据接收情况进行一些判定。

- 定义了一个用于记录接收的 Ack 的值的 Err_Ack_Num。
- 如果超过 `Wait_Time` 时间间隔未收到 ACK 报文，则需要重新发送，Re_Send 置为 true。
- 如果接收到的 Ack 大于目前窗口的 Base_Seq，则窗口向后移动。
- 如果接收到的 Ack 与发送的消息总数 Msg_Num 相等，则说明发送完成，Finish 置为 true。
- 如果接收到重复三个 Ack （此时 Count 的值为 3），则需要重新发送，Re_Send 置为 true。

```
void Receive_Ack()
{
    int Err_Ack_Num = 0;
    while (true)
    {
        Message ack_msg;
        if (clock() - Send_Time > Wait_Time)
        {
            Re_Send = true;
        }
```

```
        if (recvfrom(ClientSocket, (char *)&ack_msg, sizeof(ack_msg), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen))
        {
            if (ack_msg.Is_ACK() && ack_msg.CheckValid())
            {
                lock_guard<mutex> lock(mtx);
                cout << "Receive Ack = " << ack_msg.Ack << endl;
                if (ack_msg.Ack > Base_Seq + Header_Seq) Base_Seq = ack_msg.Ack - Header_Seq + 1;
                Print_Window(Base_Seq + Header_Seq, Next_Seq + Header_Seq);
                if (ack_msg.Ack - Header_Seq == Msg_Num + 1)
                {
                    Finish = true;
                    return;
                }
                if (Err_Ack_Num != ack_msg.Ack)
                {
                    Err_Ack_Num = ack_msg.Ack;
                    Count = 0;
                }
                else
                {
                    Count++;
                    if (Count == 3)
                    {
                        Re_Send = true;
                        Count = 0;
                    }
                }
            }
        }
    }
    return;
}
```

在发送文件时，首先按照给定路径对文件进行读取。

```
size_t found = file_path.find_last_of("/\\");
string file_name = file_path.substr(found + 1);
ifstream file(file_path, ios::binary);
if (!file.is_open())
{
    cout << "Error in Opening File!" << endl;
    exit(EXIT_FAILURE);
}
file.seekg(0, ios::end);
file_length = file.tellg();
file.seekg(0, ios::beg);
if (file_length > pow(2, 32))
{
    cout << "File is too large!" << endl;
```

```
    exit(EXIT_FAILURE);
}
```

接着计算待发送的消息总数 Msg_Num，创建新线程用于接收 Ack，同时按照计算出来的消息总数创建一个 Message 数组，作为缓冲区记录发送的数据。

```
int complete_num = file_length / MSS;
int last_length = file_length % MSS;
Header_Seq = Seq;
if (last_length != 0)
{
    Msg_Num = complete_num + 1;
}
else
{
    Msg_Num = complete_num;
}
thread Receive_Ack_Thread(Receive_Ack);
Message *data_msg = new Message[Msg_Num + 1];
```

接着循环发送数据：

- 如果 Finish 为 true，说明文件发送完毕，则直接终止发送。
- 如果 Re_Send 为 true，说明出现丢失，需要重新发送。
    - 此时借助前面定义的 Message 缓冲区，定位并重新发送窗口中的所有未确认的数据。

```
while (!Finish)
{
    if (Re_Send)
    {
        for (int i = 0; i < Next_Seq - Base_Seq; i++)
        {
            lock_guard<mutex> lock(mtx);
            if (Send(data_msg[Base_Seq + i - 1]))
            {
                SetConsoleTextAttribute(hConsole, 12);
                cout << "!Re_Send! -- Send Message to Router!" << endl;
                SetConsoleTextAttribute(hConsole, 7);
                data_msg[Base_Seq + i - 1].Print_Message();
            }
            else
            {
                cout << "Error in Sending Message!" << endl;
                exit(EXIT_FAILURE);
            }
        }
        Re_Send = false;
    }
    if (Next_Seq < Base_Seq + Windows_Size && Next_Seq - 1 <= Msg_Num)
    {
```

```
        if (Next_Seq == 1)
        {
            lock_guard<mutex> lock(mtx);
            strcpy(data_msg[Next_Seq - 1].Data, file_name.c_str());
            data_msg[Next_Seq - 1].Data[strlen(data_msg[Next_Seq - 1].Data)] =
'\0';
            data_msg[Next_Seq - 1].Length = file_length;
            data_msg[Next_Seq - 1].Seq = ++Seq;
            data_msg[Next_Seq - 1].Set_CFH();
        }
        else if (Next_Seq - 1 == Msg_Num && last_length)
        {
            lock_guard<mutex> lock(mtx);
            file.read(data_msg[Next_Seq - 1].Data, last_length);
            data_msg[Next_Seq - 1].Length = last_length;
            data_msg[Next_Seq - 1].Seq = ++Seq;
        }
        else
        {
            lock_guard<mutex> lock(mtx);
            file.read(data_msg[Next_Seq - 1].Data, MSS);
            data_msg[Next_Seq - 1].Length = MSS;
            data_msg[Next_Seq - 1].Seq = ++Seq;
        }
        if (Next_Seq % 17 == 0)
        {
            Next_Seq++;
            continue;
        }
        if (Next_Seq % 19 == 0)
        {
            Sleep(10);
            Next_Seq++;
            continue;
        }
        if (Send(data_msg[Next_Seq - 1]))
        {
            lock_guard<mutex> lock(mtx);
            data_msg[Next_Seq - 1].Print_Message();
            Next_Seq++;
            // Print_Window(Base_Seq + Header_Seq, Next_Seq + Header_Seq);
        }
        else
        {
            lock_guard<mutex> lock(mtx);
            cout << "Error in Sending Message!" << endl;
            exit(EXIT_FAILURE);
        }
    }
}
Receive_Ack_Thread.join();
```

最后重置相关操作数。

```
lock_guard<mutex> lock(mtx);
Next_Seq = 1;
Base_Seq = 1;
Finish = false;
Re_Send = false;
Header_Seq = 0;
Msg_Num = 0;
```

2. 接收端

- 首先接收发送端发送的文件头部信息，并根据 `CFH` 标志位进行确认。
- 接着计算出待接收的消息的总数，并以此创建 Message 数组作为接收缓冲区。
- 接着循环接收数据报，并将其数据写入到 Message 缓冲区中。
    - 特别的，为了避免进入“睡眠”状态，引入了 Sleep_Time，如果接收到的错误 Seq 数量超过 10，则向发送端发送三个相同的已经接收的最大 Ack，刺激发送端重新发送数据。

```
else if (Sleep_Time == 10)
{
    Message reply_msg;
    reply_msg.Ack = Waiting_Seq - 1;
    reply_msg.Set_ACK();
    for (int j = 0; j < 3; j++)
    {
        if (Send(reply_msg))
        {
            SetConsoleTextAttribute(hConsole, 12);
            cout << "Time Out! Trying to Get New Message! Waiting_Seq = " << Waiting_Seq << endl;
            SetConsoleTextAttribute(hConsole, 7);
        }
    }
    Sleep_Time = 0;
}
```

    - 设置了一个 Last_Time 变量，记录上一次收到数据的时间。如果超过 Wait_Time 为收到数据，则向发送端发送数据，刺激其重新发送数据。

```
if (clock() - Last_Time > Wait_Time)
{
    Message reply_msg;
    reply_msg.Ack = Waiting_Seq - 1;
    reply_msg.Set_ACK();
    for (int j = 0; j < 3; j++)
    {
        if (Send(reply_msg))
        {
            SetConsoleTextAttribute(hConsole, 12);
```

```
            cout << "Time Out! Trying to Get New Message! Waiting_Seq = " << Waiting_Seq << endl;
            SetConsoleTextAttribute(hConsole, 7);
        }
    }
}
```

- 接收完成后，将数据写入到文件中。

（五）断开连接——四次挥手（以发送端主动断开连接为例）

1. 发送端

- 发送第一次挥手消息，并开始计时，提出断开连接，然后等待接收第二次挥手消息。
  - 如果超时未收到，则重新发送。
- 收到正确的第二次挥手消息后，等待接收第三次挥手消息。
- 接收到正确的第三次挥手消息，输出日志，准备断开连接。
- 再等待 `2 * Wait_Time` 时间（确保消息发送完毕），断开连接。

```
void Disconnect() // * Client端主动断开连接
{
    Message discon_msg[4];

    // * First-Way Wavehand
    discon_msg[0].Seq = ++Seq;
    discon_msg[0].Set_FIN();
    int re = Send(discon_msg[0]);
    float dismsg0_Send_Time = clock();
    if (re) discon_msg[0].Print_Message();
    // * Second-Way Wavehand
    while (true)
    {
        if (recvfrom(ClientSocket, (char *)&discon_msg[1], sizeof(discon_msg[1]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            discon_msg[1].Print_Message();
            if (discon_msg[1].Seq < Seq + 1) continue;
            else if (!(discon_msg[1].Is_ACK() && discon_msg[1].CheckValid() && discon_msg[1].Seq == Seq + 1 && discon_msg[1].Ack == discon_msg[0].Seq))
            {
                cout << "Error Message!" << endl;
                exit(EXIT_FAILURE);
            }
            Seq = discon_msg[1].Seq;
            break;
        }
        if ((clock() - dismsg0_Send_Time) > Wait_Time)
        {
            SetConsoleTextAttribute(hConsole, 12);
```

```
            cout << "!Time Out! -- First-Way Wavehand" << endl;
            int re = Send(discon_msg[0]);
            dismsg0_Send_Time = clock();
            if (re)
            {
                discon_msg[0].Print_Message();
                SetConsoleTextAttribute(hConsole, 7);
            }
        }
    }
    // * Third-Way Wavehand
    while (true)
    {
        if (recvfrom(ClientSocket, (char *)&discon_msg[2], sizeof(discon_msg[2]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            discon_msg[2].Print_Message();
            if (!(discon_msg[2].Is_ACK() && discon_msg[2].Is_FIN() && discon_msg[2].CheckValid() && discon_msg[2].Seq == Seq + 1 && discon_msg[2].Ack == discon_msg[1].Seq))
            {
                cout << "Error Message!" << endl;
                exit(EXIT_FAILURE);
            }
            Seq = discon_msg[2].Seq;
            break;
        }
    }
    // * Fourth-Way Wavehand
    discon_msg[3].Ack = discon_msg[2].Seq;
    discon_msg[3].Set_ACK();
    discon_msg[3].Seq = ++Seq;
    if (Send(discon_msg[3]))
    {
        discon_msg[3].Print_Message();
    }
    cout << "Fourth-Way Wavehand is successful!" << endl;
    Wait_Exit();
}
void Wait_Exit()
{
    Message exit_msg;
    float exit_msg_time = clock();
    while (clock() - exit_msg_time < 2 * Wait_Time)
    {
        if (recvfrom(ClientSocket, (char *)&exit_msg, sizeof(exit_msg), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen))
        {
            Seq = exit_msg.Seq;
            exit_msg.Ack = exit_msg.Seq;
            exit_msg.Set_ACK();
            exit_msg.Seq = ++Seq;
```

```
            Send(exit_msg);
        }
    }
    closesocket(ClientSocket);
    WSACleanup();
    cout << "Client is closed!" << endl;
}
```

2. 接收端

- 接收正确第一次挥手消息，发送第二次挥手消息，同意断开连接。
- 发送第三次挥手消息，并开始计时，然后等待接收第四次挥手消息。
    - 如果超时未收到，则重新发送。
- 接收到正确的第四次挥手消息，输出日志，断开连接。

```
void Disconnect() // * Router端主动断开连接
{
    Message discon_msg[4];
    while (true)
    {
        // * First-Way Wavehand
        if (recvfrom(ServerSocket, (char *)&discon_msg[0], sizeof(discon_msg[0]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            discon_msg[0].Print_Message();
            if (!(discon_msg[0].Is_FIN() && discon_msg[0].CheckValid()))
            {
                cout << "Error Message!" << endl;
                exit(EXIT_FAILURE);
            }
            Seq = discon_msg[0].Seq;
        }
        // * Second-Way Wavehand
        discon_msg[1].Ack = discon_msg[0].Seq;
        discon_msg[1].Seq = ++Seq;
        discon_msg[1].Set_ACK();
        if (Send(discon_msg[1])) discon_msg[1].Print_Message();
        break;
    }
    // * Third-Way Wavehand
    discon_msg[2].Ack = discon_msg[1].Seq;
    discon_msg[2].Seq = ++Seq;
    discon_msg[2].Set_ACK();
    discon_msg[2].Set_FIN();
    int re = Send(discon_msg[2]);
    float dismsg3_Send_Time = clock();
    if (re) discon_msg[2].Print_Message();
    // * Fourth-Way Wavehand
    while (true)
    {
```

```
        if (recvfrom(ServerSocket, (char *)&discon_msg[3], sizeof(discon_msg[3]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            discon_msg[3].Print_Message();
            if (discon_msg[3].Seq < Seq + 1) continue;
            else if (!(discon_msg[3].Is_ACK() && discon_msg[3].CheckValid() && discon_msg[3].Seq == Seq + 1 && discon_msg[3].Ack == discon_msg[2].Seq))
            {
                cout << "Error Message!" << endl;
                exit(EXIT_FAILURE);
            }
            Seq = discon_msg[3].Seq;
            cout << "Fourth-Way Wavehand is successful!\n\n";
            break;
        }
        if ((clock() - dismsg3_Send_Time) > Wait_Time)
        {
            int re = Send(discon_msg[2]);
            dismsg3_Send_Time = clock();
            if (re)
            {
                HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
                cout << "!Time Out! -- Send Message to Router! -- Third-Way Wavehand" << endl;
                discon_msg[2].Print_Message();
                SetConsoleTextAttribute(hConsole, 7);
            }
        }
    }
    Exit();
}
void Exit()
{
    closesocket(ServerSocket);
    WSACleanup();
    cout << "Server is closed!" << endl;
}
```

四、传输测试与性能分析

（一）传输测试

本次实验中，手动编写了丢包与时延模拟。

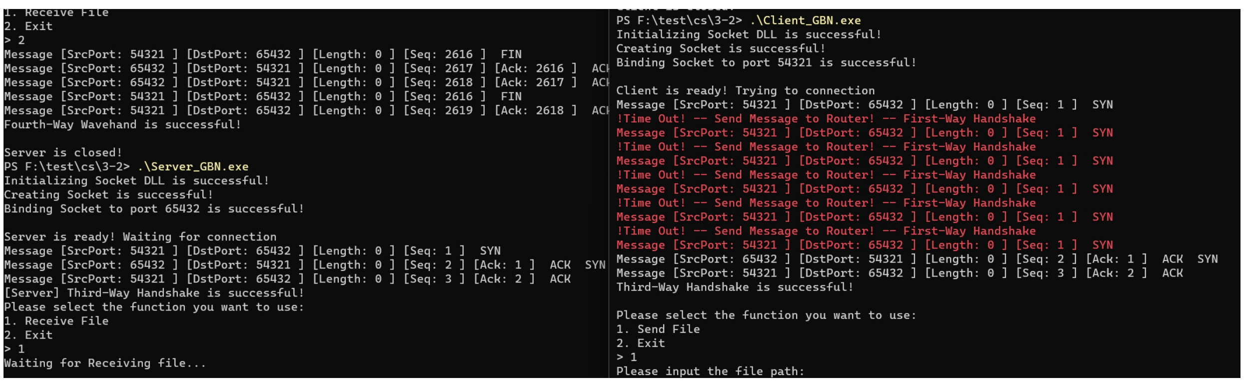
```
if (Next_Seq % 17 == 0)
{
    Next_Seq++;
    continue;
}
if (Next_Seq % 19 == 0)
{
    Sleep(10);
    Next_Seq++;
    continue;
}
```

1. 本机测试

本次实验在本机上实现了给定测试文件的测试，设置窗口大小为 `20`。

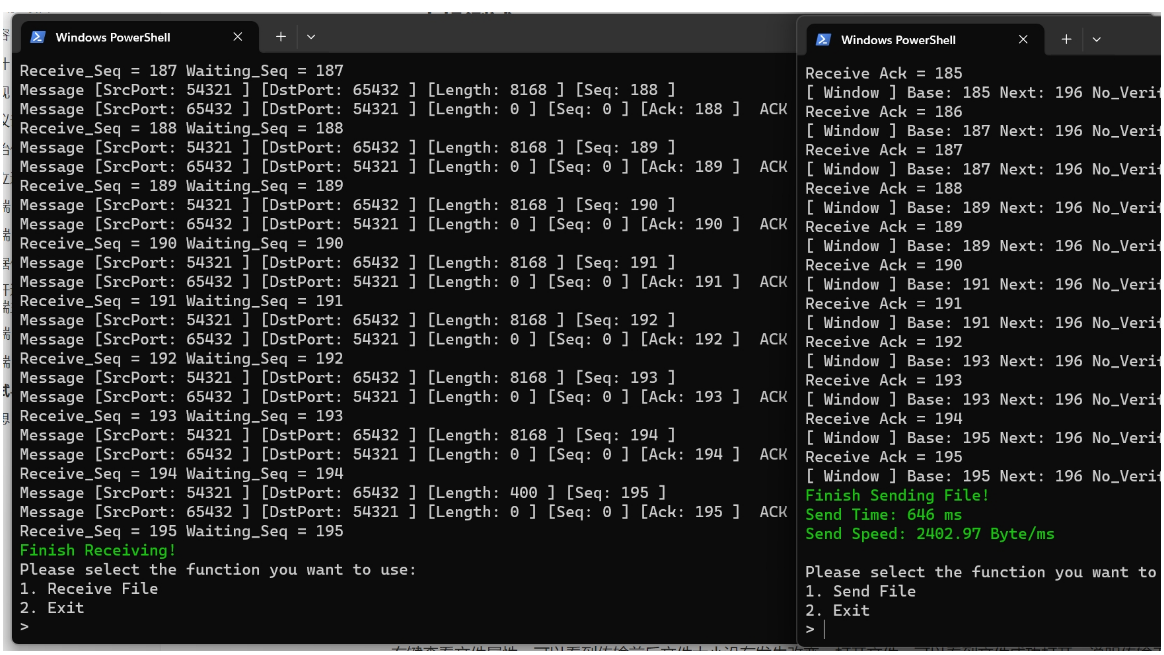
（1）三次握手

可以看到，发送端与接收端成功完成三次握手，建立连接。



（2）helloworld.txt

传输完成，可以看到，累计用时 646 ms，吞吐率为 2402.97 $Byte/ms$。

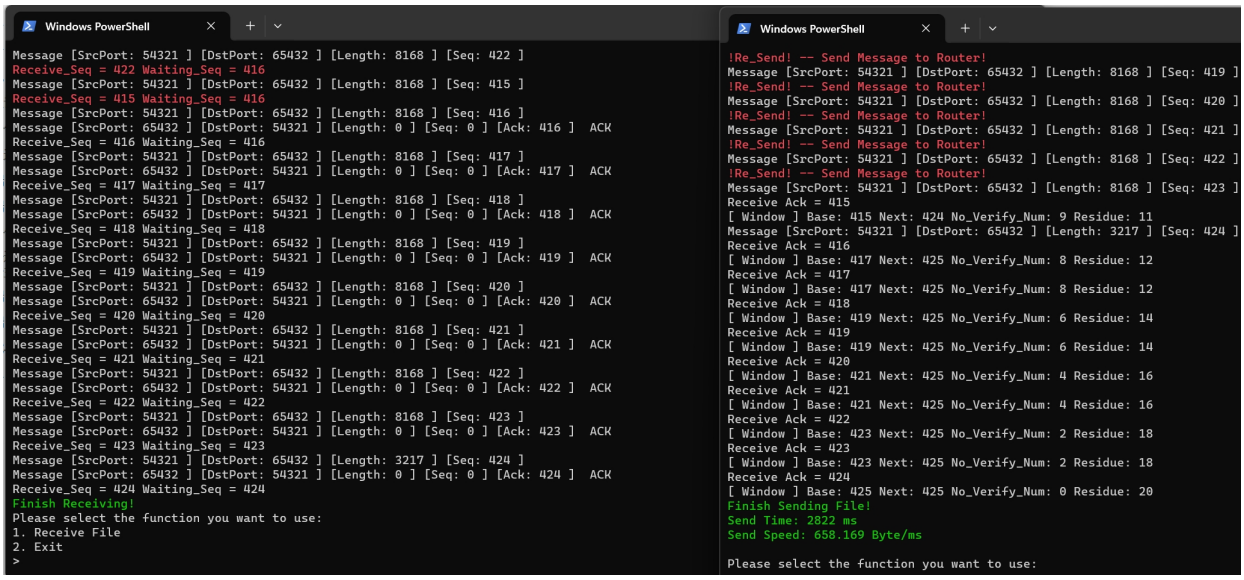


右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。

（3）1.jpg

红色输出即为丢包部分，可以看到能够在丢包情况下实现数据传输。

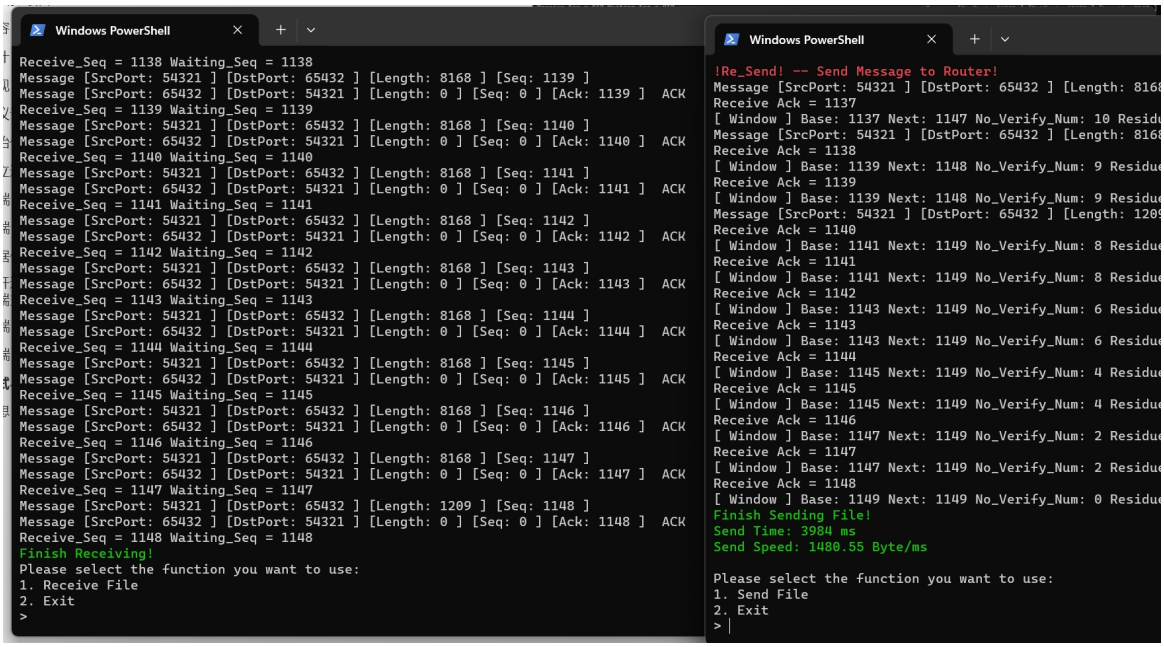
传输完成，可以看到，累计用时 2822 ms，吞吐率为 658.169 $Byte/ms$。



右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。

（4）2.jpg

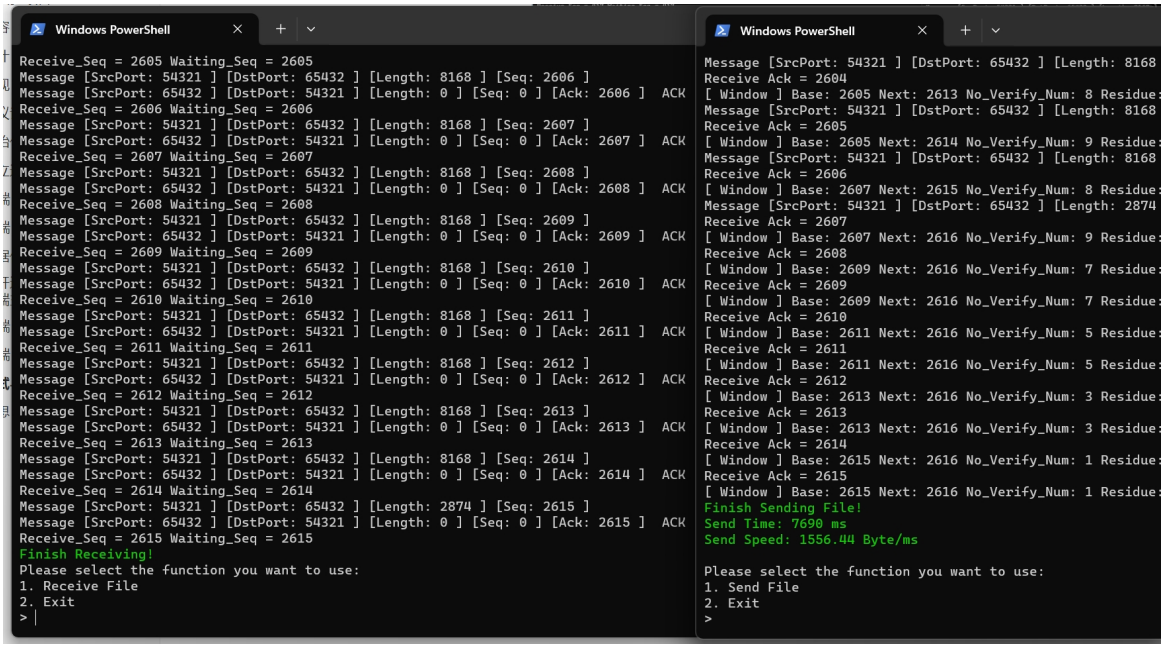
传输完成，可以看到，累计用时 3984 ms，吞吐率为 1480.55 $Byte/ms$。



右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。

（5）3.jpg

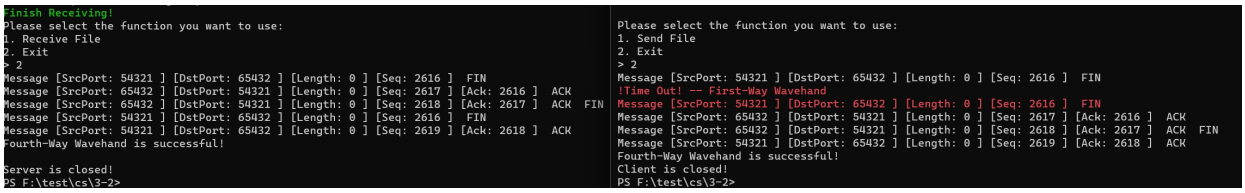
传输完成，可以看到，累计用时 7690 ms，吞吐率为 1556.44 $Byte/ms$。



右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。

（6）四次挥手

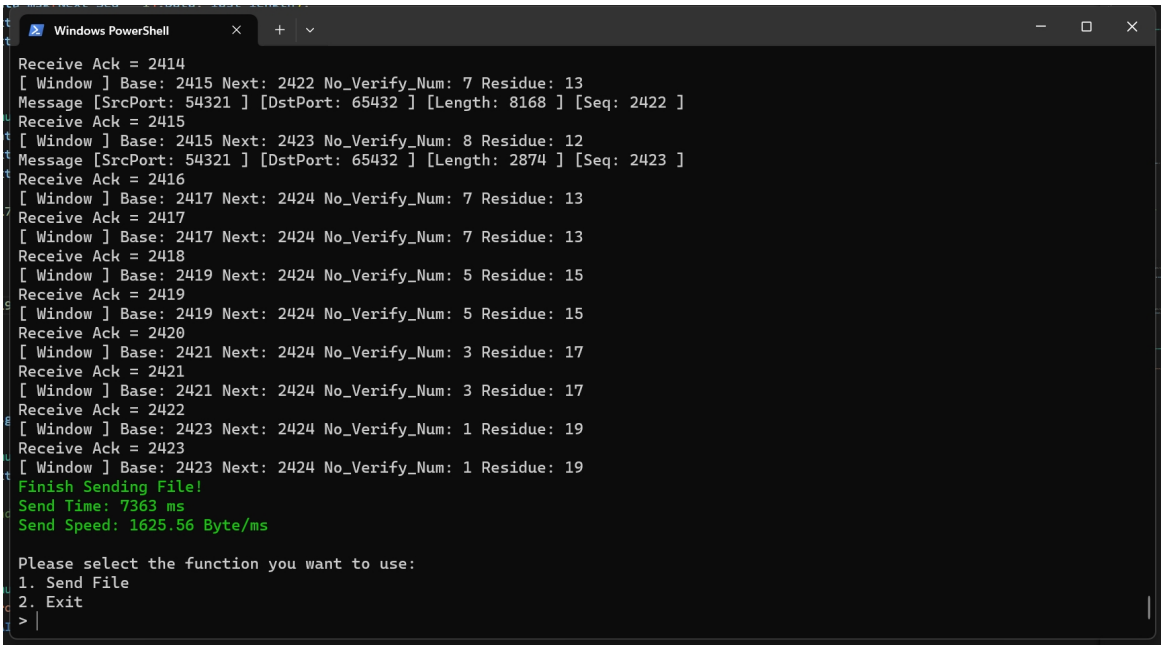
可以看到发送端与接收端成功完成四次挥手，断开连接。



2. 局域网下联机测试

本次实验中，还借助局域网进行联机测试。仅以 3.jpg 为例进行展示。

传输完成，可以看到，累计用时 7363 ms，吞吐率为 1625.56 $Byte/ms$。

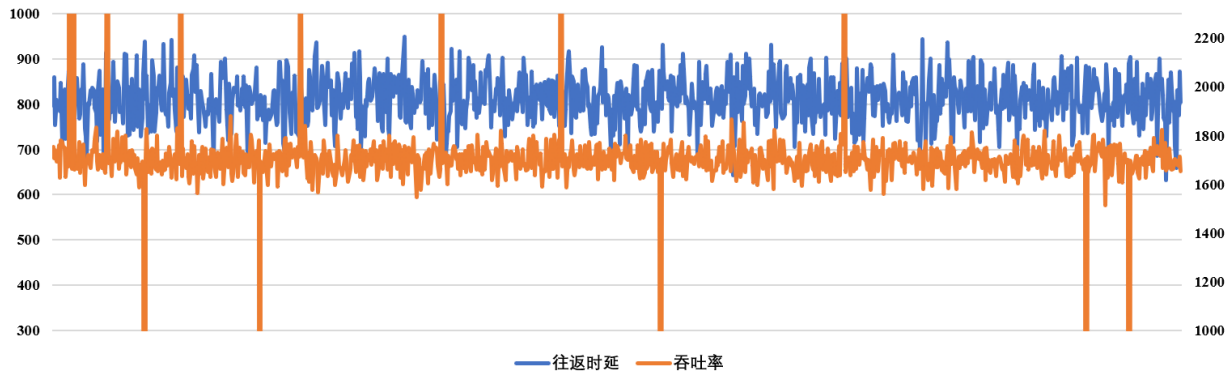


右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。

经过测试，文件在丢包及时延条件下均能够正确传输，说明程序设计成功。

（二）性能分析

在上面的传输测试中，添加日志输出，将传输时间、吞吐率、往返时延予以输出，进行数据清洗，然后使用 `excel` 绘制了实时吞吐率与实时往返时延的数据分析折线图。



可以直观看到，实时吞吐率逐渐提升并稳定在 1500 $Byte/ms$，而实时往返时延稳定在约 800 μs，偶尔会有波动。

> 以上数据仅对本次实验负责。

五、问题反思

（一）协议格式错乱

在结构体定义代码处的 `#pragma pack(1)` 是一个编译器指令，用于告诉编译器以最小的字节对齐单位对结构体进行打包，该处即以 16 字节进行对齐。如果不加该指令，编译的时候可能会由于优化等过程改变原有的数据报文设计。

（二）创新超时重传机制

本次实验中借鉴快速重传机制重新设计了超时重传机制，这样既可以保证数据传输的完整可靠，又实现了按照网络带宽实时确定重传时间，保证了传输效率。

（三）全 0 数据

数据传输时，有时会出现数据全 0 的情况，即源端口号、目的端口号、数据段等均为 0 的情况，目前仍未曾知晓产生原因。为了避免其对运行的干扰，在接收端做了对应的修改。即，当接收到全 0 数据时，将向发送端发送等待的最大 Ack，刺激其进行快速重传操作。

```
if (rec_msg.Seq == 0)
{
    Message reply_msg;
    reply_msg.Ack = Waiting_Seq - 1;
    reply_msg.Set_ACK();
    for (int j = 0; j < 3; j++)
    {
        if (Send(reply_msg))
        {
            SetConsoleTextAttribute(hConsole, 12);
            cout << "Zero Seq! Trying to Get New Message! Waiting_Seq = " <<
Waiting_Seq << endl;
            SetConsoleTextAttribute(hConsole, 7);
        }
    }
}
```