计算机网络实验报告

Lab3-1 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。流量控制采用停等机制，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 1000Byte，整个数据报大小为 1024Byte。

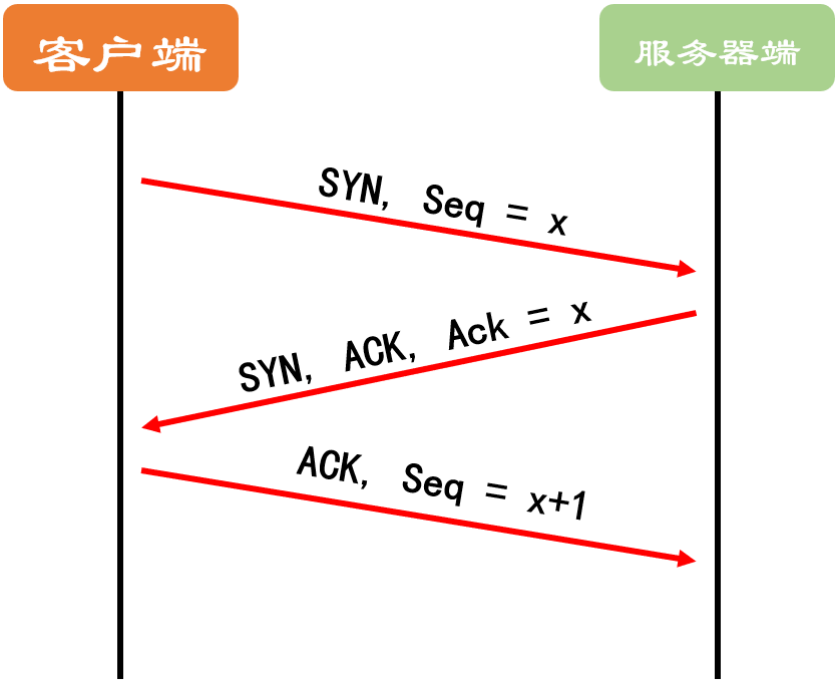
```
|0            15|16            31|
-----------------------------------
|            SrcPort              |
-----------------------------------
|           DestPort              |
-----------------------------------
|             Seq                 |
-----------------------------------
|             Ack                 |
-----------------------------------
|            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，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. 停等机制与接收确认

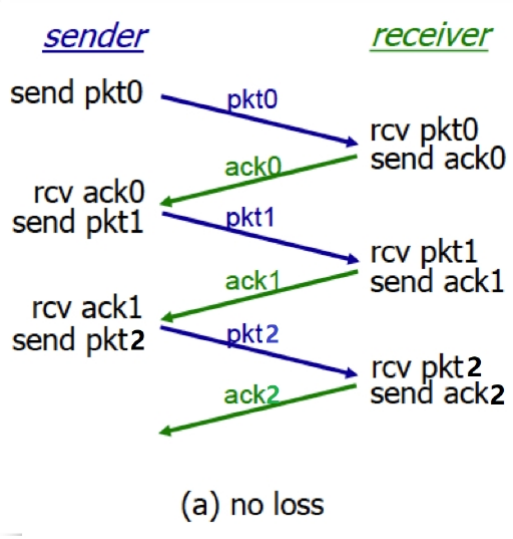
按照实验要求，本次实验需要使用停等机制进行流量控制，即发送方发送完成后，要收到接收端返回的对应 ACK 确认报文才能进行下一步传输。按照前文叙述，应接收的确认报文的 Ack 等于发送的序列号 Seq。

4. 超时重传

本次实验实现了超时重传功能以解决数据包丢失及失序问题。即，发送端每次发送数据后立刻进行计时，如果超过最大等待时间 `Wait_Time` 仍没有收到对应的接收端发送的 ACK 确认报文，将重新发送数据。

（1）理想情况

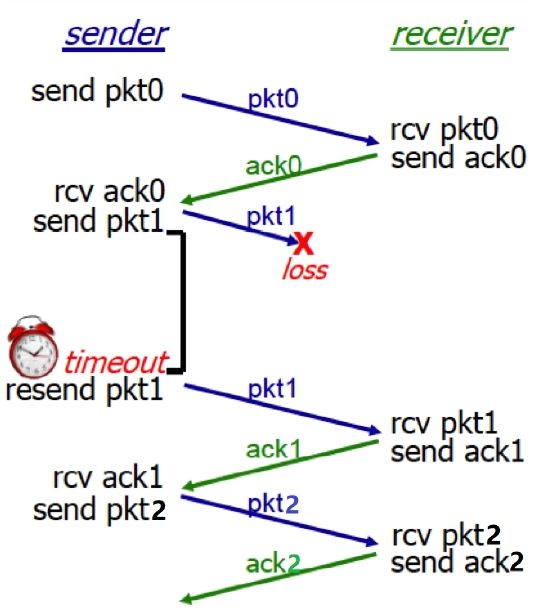
数据包正常发送，接收端正常接收，没有发生数据包丢失或失序问题。示意图如下：



(a) no loss

(2) 数据包发送丢失

数据包正常发送，但发生数据包丢失问题。示意图如下：

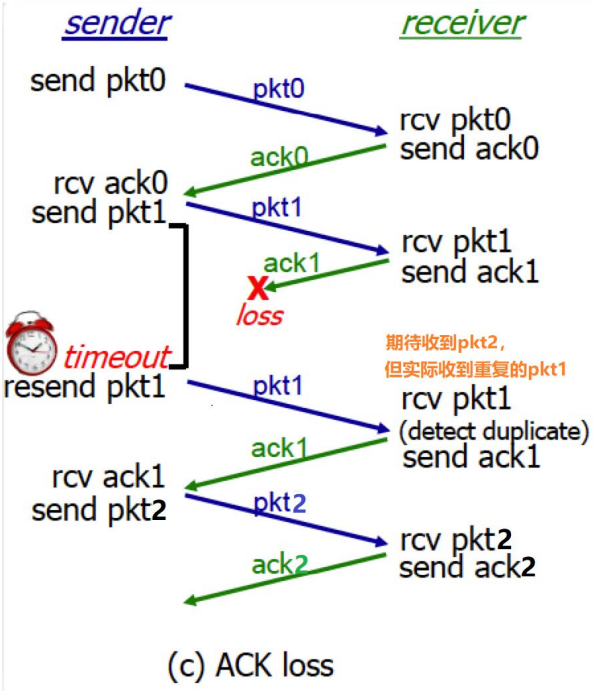


(b) packet loss

该种情况下，由于发送端发送时数据丢失，接收端没有收到消息而没有发送 ACK 确认报文，`Wait_Time` 时间后，发送端仍没有收到对应的 ACK 确认报文，此时发送端将重新发送数据。

(3) 数据包接收丢失

数据包正常发送，但发生数据包丢失问题。示意图如下：

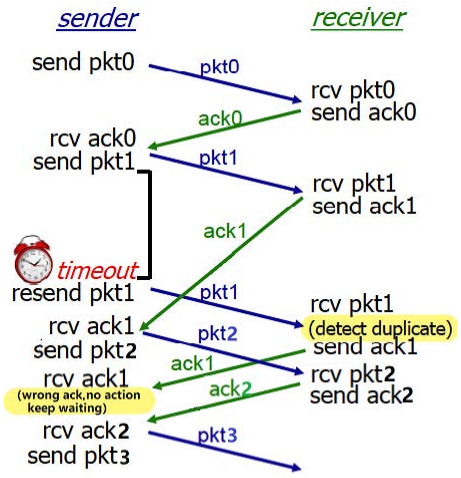


(c) ACK loss

该种情况下，由于发送端接收时数据丢失，接收端收到消息发送 ACK 确认报文，但该报文丢失，`Wait_Time` 时间后，发送端仍没有收到对应的 ACK 确认报文，此时发送端将重新发送数据。同时，接收端将对接收到的消息的 Seq 进行验证，如果与预期不符，将丢弃该数据包，并输出日志，接着继续接收其他报文。

（4）数据包失序

数据包正常发送，但是接收端或发送端由于种种原因发生数据包失序问题。示意图如下：

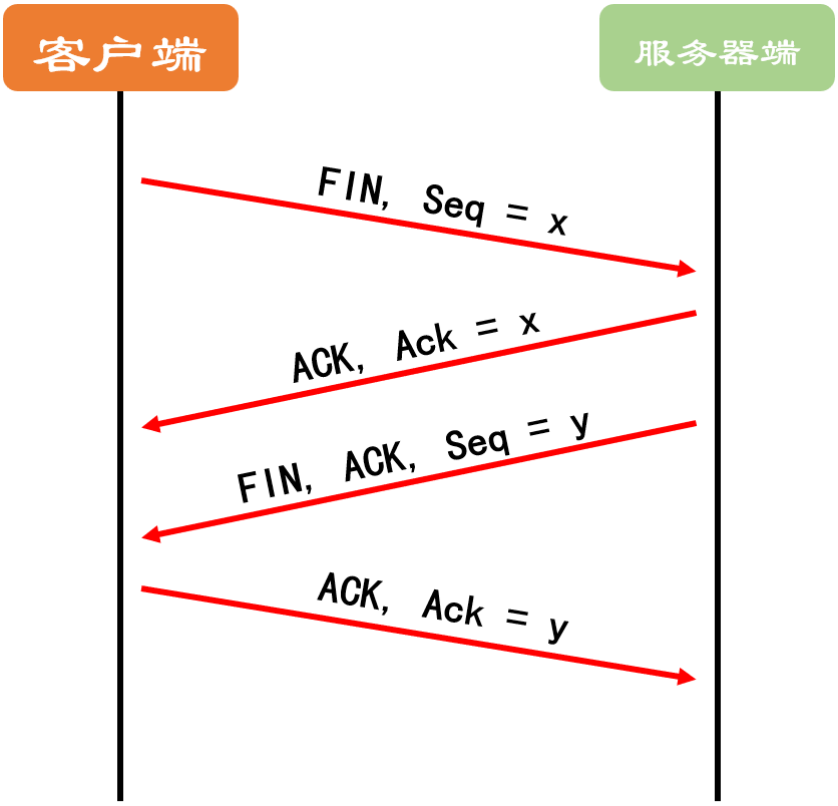


(d) premature timeout/ delayed ACK

针对该情况，每个数据包发送时都设置相应的定时器与 Seq，接收时需要同时检验时间与 Seq，如果超时未收到对应的 ACK 确认报文，将重新发送数据；如果收到不符合预期的 Seq 报文，将丢弃报文，并输出日志，接着继续接收其他报文。

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，示意图如下：



6. 状态机

（1）发送端

- 建立连接，发送报文，Seq = x，启动计时器，等待回复
  - 超时未收到 ACK 确认报文：重新发送数据并重新计时
  - 收到 ACK 确认报文，但 Ack 不匹配：丢弃报文，输出日志，继续等待
- 收到 ACK 确认报文，且 Ack 及相关标志位匹配成功：继续发送下一个报文或关闭连接

（2）接收端

- 建立连接，等待接收
  - 收到报文，但 Seq 或相关标志位不匹配：丢弃报文，输出日志，继续等待
- 收到报文，且 Seq 或相关标志位匹配：接收报文，发送对应 Ack，继续等待下一个报文或关闭连接

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());
```

```
28
29      if (bind(ClientSocket, (SOCKADDR *)&ClientAddr, sizeof(SOCKADDR)) ==
SOCKET_ERROR)
30      {
31          cout <<"[Client] "<< "Error in Binding Socket!\n";
32          exit(EXIT_FAILURE);
33          return false;
34      }
35      cout <<"[Client] "<< "Binding Socket to port "<< Client_Port<<" is
successful!\n\n";
36      RouterAddr.sin_family = AF_INET;
37      RouterAddr.sin_port = htons(Router_Port);
38      RouterAddr.sin_addr.S_un.S_addr = inet_addr(RouterIP.c_str());
39      return true;
40  }
```

（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

```
1   bool Connect()
2   {
3       Message con_msg[3];
4       // * First-Way Handshake
5       con_msg[0].Seq = ++Seq;
6       con_msg[0].Set_SYN();
7       int re = Send(con_msg[0]);
8       float msg1_Send_Time = clock();
9       if (re > 0)
10      {
11          // * Second-Way Handshake
12          while(true)
13          {
14              if (recvfrom(ClientSocket, (char *)&con_msg[1], sizeof(con_msg[1]),
0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
15              {
16                  if (!(con_msg[1].Is_ACK() && con_msg[1].Is_SYN()
&&con_msg[1].CheckValid() && con_msg[1].Ack == con_msg[0].Seq))
17                  {
18                      cout <<"[Client] "<< "Error Message!" << endl;
19                      exit(EXIT_FAILURE);
20                  }
21                  Seq = con_msg[1].Seq;
22                  break;
23              }
24              if ((clock() - msg1_Send_Time) > Wait_Time)
```

```
            {
                int re = Send(con_msg[0]);
                msg1_Send_Time = clock();
                if (re > 0)
                {
                    cout <<"[Client] "<< "Time Out! -- Send Message to Router! -- First-Way Handshake" << endl;
                }
            }
        }
    }
    // * Third-Way Handshake
    con_msg[2].Ack = con_msg[1].Seq;
    con_msg[2].Seq = ++Seq;
    con_msg[2].Set_ACK();
    re = Send(con_msg[2]);
    if (re > 0) {}
    cout<<"[Client] "<< "Third-Way Handshake is successful!" << endl <<endl;
    return true;
}
```

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

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。

```
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);
}
```

接着将文件的名称写入 `Message` 的 `Data` 数据段，将文件的大小写入 `Length`，然后将该信息发送出去，作为发送文件的头部信息。此处发送启用超时重传机制。

```
Message send_msg;
strcpy(send_msg.Data, file_name.c_str());
send_msg.Data[strlen(send_msg.Data)] = '\0';
send_msg.Length = file_length;
send_msg.Seq = ++Seq;
send_msg.Set_CFH();
float last_time;
int re = Send(send_msg);
float msg1_Send_Time = clock();
if (re > 0)
{
    cout <<"[Client] "<< "Send Message to Router! -- File Header" << endl;
}

while(true)
{
    Message tmp;
    if (recvfrom(ClientSocket, (char *)&tmp, sizeof(tmp), 0, (SOCKADDR
*)&RouterAddr, &RouterAddrLen) > 0)
    {
        cout <<"[Client] "<< "Receive Message from Router! -- File Header" <<
endl;
        if (tmp.Is_ACK() && tmp.CheckValid() && tmp.Seq == Seq + 1)
        {
            Seq = tmp.Seq;
            last_time = clock() - msg1_Send_Time;
            break;
        }
        else if (tmp.CheckValid() && tmp.Seq != Seq + 1)
        {
            Message reply_msg;
            reply_msg.Ack = tmp.Seq;
            reply_msg.Set_ACK();
            if(Send(reply_msg)>0)
```

```
            {
                cout<<"!Repeatedly! [Client]"<< "Receive Seq = "<<tmp.Seq<<" Reply Ack = "<<reply_msg.Ack<<endl;
            }
        }
    }
    else if (clock()-msg1_Send_Time > last_time)
    {
        int re = sendto(ClientSocket, (char *)&send_msg, sizeof(send_msg), 0, (SOCKADDR *)&RouterAddr, RouterAddrLen);
        msg1_Send_Time = clock();
        if (re > 0)
        {
            cout <<"[Client] "<< "Time Out! -- Send Message to Router! -- File Header" << endl;
        }
        else
        {
            cout<<"[Client] "<<"Error in Sending Message! -- File Header" <<endl;
            exit(EXIT_FAILURE);
        }
    }
}
```

在收到接收端发送的正确的确认报文后，进行后续文件的传输。

- 按照文件大小，结合协议的设计中预留数据段的大小，计算完整的数据段个数以及不完全的数据段大小
- 循环发送，并实时接收确认报文
- 同时，设定计时器，计算往返时延，根据传输带宽确定等待时长；实时计算吞吐率与往返时延，设定日志输出

```
struct timeval complete_time_start, complete_time_end;
gettimeofday(&complete_time_start, NULL);
float complete_time = clock();
int complete_num = file_length / MSS;
int last_length = file_length % MSS;
cout <<"[Client] "<< "Start to Send Message to Router! -- File" << endl;
for(int i=0;i<=complete_num;i++)
{
    Message data_msg;
    if (i!=complete_num)
    {
        file.read(data_msg.Data, MSS);
        data_msg.Length = MSS;
        data_msg.Seq = ++Seq;
        int re = Send(data_msg);
        struct timeval every_time_start, every_time_end;
        long long every_time_usec;
        gettimeofday(&every_time_start, NULL);
        float time = clock();
```

```
        if (re > 0)
        {
            Message tmp;
            while(true)
            {
                if (recvfrom(ClientSocket, (char *)&tmp, sizeof(tmp), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
                {
                    if (tmp.Is_ACK() && tmp.CheckValid() && tmp.Seq == Seq + 1)
                    {
                        Seq = tmp.Seq;
                        gettimeofday(&every_time_end, NULL);
                        every_time_usec = (every_time_end.tv_usec - every_time_start.tv_usec);
                        if(i % 1 == 0)
                        {
                            gettimeofday(&complete_time_end, NULL);
                            long long complete_time_usec = (complete_time_end.tv_usec - complete_time_start.tv_usec);
                            time_txt << (complete_time_usec) << "," << (every_time_usec ) << "," << ((double)(MSS * i)/(complete_time_usec)*1000) <<endl;
                        }
                        break;
                    }
                    else if (tmp.CheckValid() && tmp.Seq != Seq + 1)
                    {
                        Message reply_msg;
                        reply_msg.Ack = tmp.Seq;
                        reply_msg.Set_ACK();
                        if(Send(reply_msg)>0)
                        {
                            cout<<"!Repeatedly! [Client]"<< "Receive Seq = " <<tmp.Seq<<" Reply Ack = "<<reply_msg.Ack<<endl;
                        }
                    }
                }
                else if (clock()-time > every_time_usec)
                {
                    int re = sendto(ClientSocket, (char *)&data_msg, sizeof(data_msg), 0, (SOCKADDR *)&RouterAddr, RouterAddrLen);
                    time = clock();
                    if (re > 0)
                    {
                        cout <<"[Client] "<< "Time Out! -- Send Message to Router! Part " << i << "-- File" << endl;
                    }
                    else
                    {
                        cout<<"[Client] "<<"Error in Sending Message! Part " <<i<<" -- File"<<endl;
```

```
                        exit(EXIT_FAILURE);
                    }
                }
            }
        }
    }
    else
    {
        Message data_msg;
        file.read(data_msg.Data, last_length);
        data_msg.Length = last_length;
        data_msg.Seq = ++Seq;
        int re = Send(data_msg);
        // float every_time_start = clock();
        struct timeval every_time_start, every_time_end;
        long long every_time_usec;
        gettimeofday(&every_time_start, NULL);
        float time = clock();
        if (re > 0)
        {
            Message tmp;
            while(true)
            {
                if (recvfrom(ClientSocket, (char *)&tmp, sizeof(tmp), 0,
(SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
                {
                    if (tmp.Is_ACK() && tmp.CheckValid() && tmp.Seq == Seq +
1)
                    {
                        Seq = tmp.Seq;
                        gettimeofday(&every_time_end, NULL);
                        every_time_usec = (every_time_end.tv_usec -
every_time_start.tv_usec);
                        if(i % 1 == 0)
                        {
                            gettimeofday(&complete_time_end, NULL);
                            long long complete_time_usec =
(complete_time_end.tv_usec - complete_time_start.tv_usec);
                            time_txt << (complete_time_usec) << "," <<
(every_time_usec ) << "," << ((double)
(last_length)/(complete_time_usec)*1000);
                        }
                        break;
                    }
                    else if (tmp.CheckValid() && tmp.Seq != Seq + 1)
                    {
                        Message reply_msg;
                        reply_msg.Ack = tmp.Seq;
                        reply_msg.Set_ACK();
                        if(Send(reply_msg)>0)
                        {
```

```
                            cout<<"!Repeatedly! [Client]"<< "Receive Seq = "
<<tmp.Seq<<" Reply Ack = "<<reply_msg.Ack<<endl;
                        }
                    }
                }
                else if (clock()-time > every_time_usec)
                {
                    int re = sendto(ClientSocket, (char *)&data_msg,
sizeof(data_msg), 0, (SOCKADDR *)&RouterAddr, RouterAddrLen);
                    time = clock();
                    if (re > 0)
                    {
                        cout <<"[Client] "<< "Time Out! -- Send Message to
Router! Part " << i << "-- File" << endl;
                    }
                    else
                    {
                        cout<<"[Client] "<<"Error in Sending Message! Part "
<<i<<" -- File"<<endl;
                        exit(EXIT_FAILURE);
                    }
                }
            }
        }
    }
}
```

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。
- 接着按照接收到的文件头部信息，以二进制方式打开文件，便于写入。然后循环接收报文消息，实时写入文件中。

未避免报告冗长，此处代码不再展示。

（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 > 0) {}
    // * Second-Way Wavehand
    while (true)
    {
        if(discon_msg[0].Seq < Seq + 1)
        {
            continue;
        }
        if (recvfrom(ClientSocket, (char *)&discon_msg[1], sizeof(discon_msg[1]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            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 << "[Client] " << "Error Message!" << endl;
                exit(EXIT_FAILURE);
            }
            Seq = discon_msg[1].Seq;
            break;
        }
        if ((clock() - dismsg0_Send_Time) > Wait_Time)
        {
            cout << "[Client] " << "Time Out! -- First-Way Wavehand" << endl;
            int re = Send(discon_msg[0]);
            dismsg0_Send_Time = clock();
            if (re > 0) {}
        }
    }
    // * Third-Way Wavehand
    while (true)
    {
        if (recvfrom(ClientSocket, (char *)&discon_msg[2], sizeof(discon_msg[2]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            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 << "[Client] " << "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;
    re = Send(discon_msg[3]);
    if (re > 0) {}
    cout << "[Client] " << "Fourth-Way Wavehand is successful!" << endl << endl;
    Wait_Exit();
    return;
}
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) > 0)
        {
            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] " << "Client is closed!" << endl;
    system("pause");
}
```

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)
        {
            if (!(discon_msg[0].Is_FIN() && discon_msg[0].CheckValid() && discon_msg[0].Seq == Seq + 1))
            {
                cout << "[Server] " << "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();
        int re = Send(discon_msg[1]);
        if (re > 0) {}
        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 > 0) {}
    // * Fourth-Way Wavehand
    while (true)
    {
        if (recvfrom(ServerSocket, (char *)&discon_msg[3],
sizeof(discon_msg[3]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
        {
            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 << "[Server] " << "Error Message!" << endl;
                exit(EXIT_FAILURE);
            }
            Seq = discon_msg[3].Seq;
            cout << "[Server] " << "Fourth-Way Wavehand is successful!" <<
endl;
            break;
        }
        if ((clock() - dismsg3_Send_Time) > Wait_Time)
        {
            int re = Send(discon_msg[2]);
            dismsg3_Send_Time = clock();
            if (re > 0)
            {
                cout << "[Server] " << "Time Out! -- Send Message to Router! --
Third-Way Wavehand" << endl;
            }
        }
    }
```

```
60      Exit();
61      return;
62  }
63  void Exit()
64  {
65      closesocket(ServerSocket);
66      WSACleanup();
67      cout << "[Server] " << "Server is closed!" << endl;
68      system("pause");
69  }
```

四、传输测试与性能分析

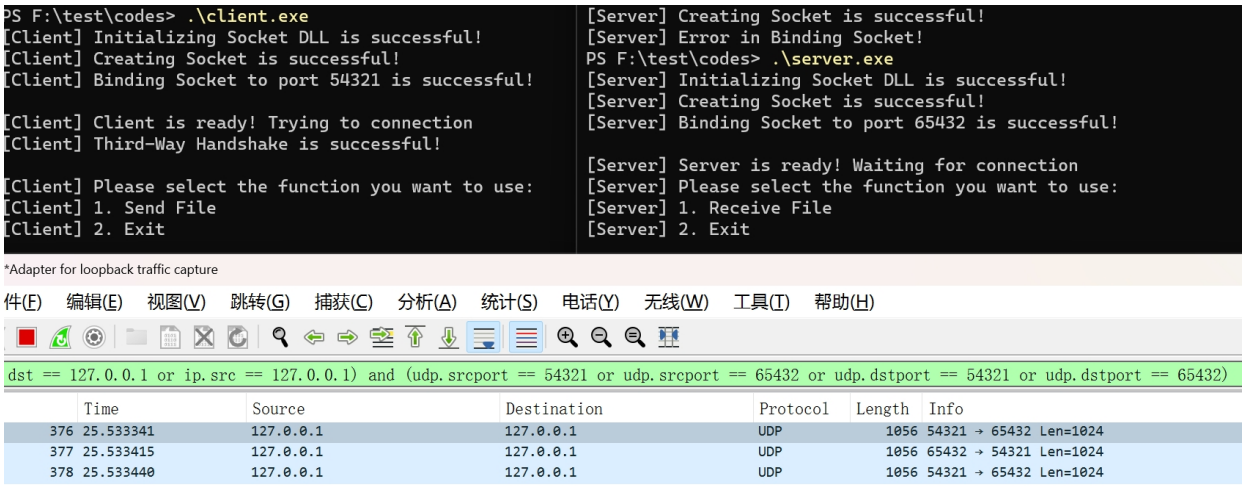
（一）传输测试

1. 本机测试

本次实验使用大中小三种类型文件进行传输测试，并使用 `wireshark` 进行抓包辅助测试。

（1）小文件

运行 `wireshark` ，设置过滤条件，接着启动发送端与接收端，首先可以看到我们设计的三次握手信息。



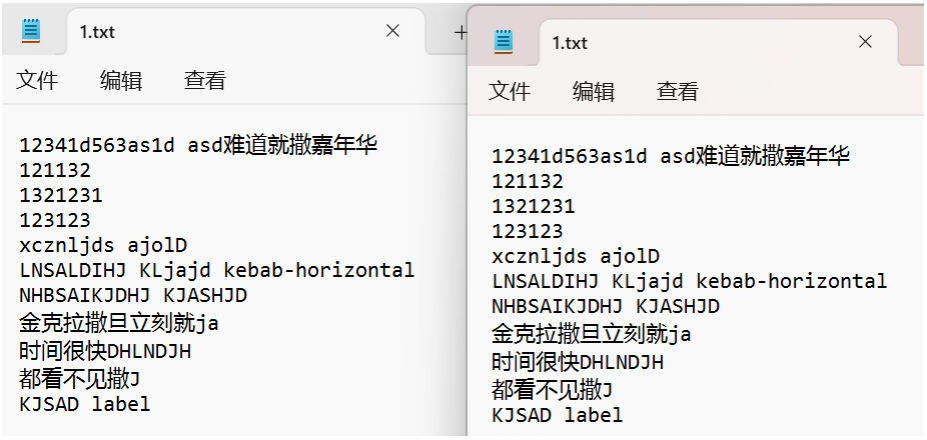
接着设置好发生端与接收端信息，开始传输文件。此处以大小为 219Byte 的文件 `1.txt` 为例。

传输完成，可以看到，累计用时 2 ms，吞吐率为 109.5 $Byte/ms$。

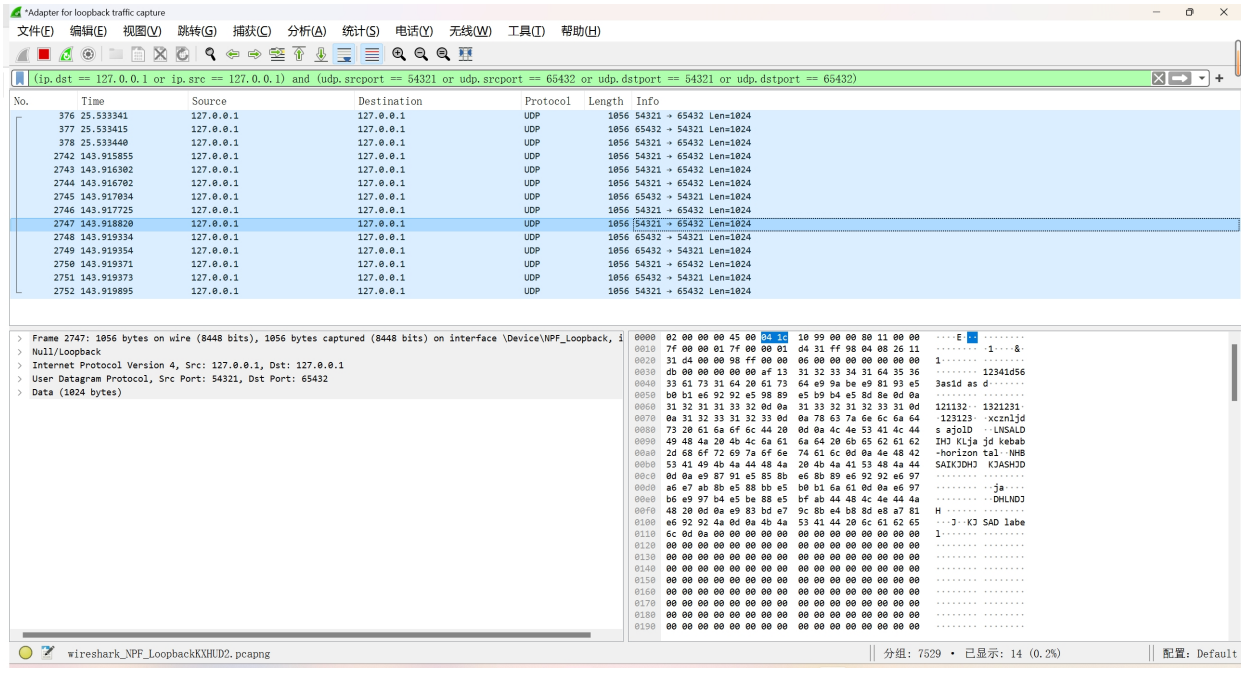
```
PS F:\test\codes> .\client.exe
[Client] Initializing Socket DLL is successful!
[Client] Creating Socket is successful!
[Client] Binding Socket to port 54321 is successful!

[Client] Client is ready! Trying to connection
[Client] Third-Way Handshake is successful!

[Client] Please select the function you want to use:
[Client] 1. Send File
[Client] 2. Exit
1
[Client] Please input the file path:
1\1.txt
[Client] Send Message to Router! -- File Header
[Client] Time Out! -- Send Message to Router! -- File He
[Client] Time Out! -- Send Message to Router! -- File He
[Client] Receive Message from Router! -- File Header
[Client] Start to Send Message to Router! -- File
[Client] Time Out! -- Send Message to Router! Part 0-- F
!Repeatedly! [Client]Receive Seq = 0 Reply Ack = 0
!Repeatedly! [Client]Receive Seq = 0 Reply Ack = 0
[Client] Finish Sending File!
[Client] Send Time: 2 ms
[Client] Send Speed: 109.5 Byte/ms

[Client] Please select the function you want to use:
[Client] 1. Send File
[Client] 2. Exit
```

```
Message [SrcPort: 54321 ] [DstPort: 65432 ] [Seq: 4
0
1
4
7
0
6
[Server] Error Message!
PS F:\test\codes> .\server.exe
[Server] Initializing Socket DLL is successful!
[Server] Creating Socket is successful!
[Server] Error in Binding Socket!
PS F:\test\codes> .\server.exe
[Server] Initializing Socket DLL is successful!
[Server] Creating Socket is successful!
[Server] Binding Socket to port 65432 is successful!

[Server] Server is ready! Waiting for connection
[Server] Please select the function you want to use:
[Server] 1. Receive File
[Server] 2. Exit
1
[Server] Waiting for Receiving file...
[Server] Receive File Name: 1.txt File Size: 219
[Server] Start Receiving File!
[Server] Finish Receiving!
[Server] Please select the function you want to use:
[Server] 1. Receive File
[Server] 2. Exit
```

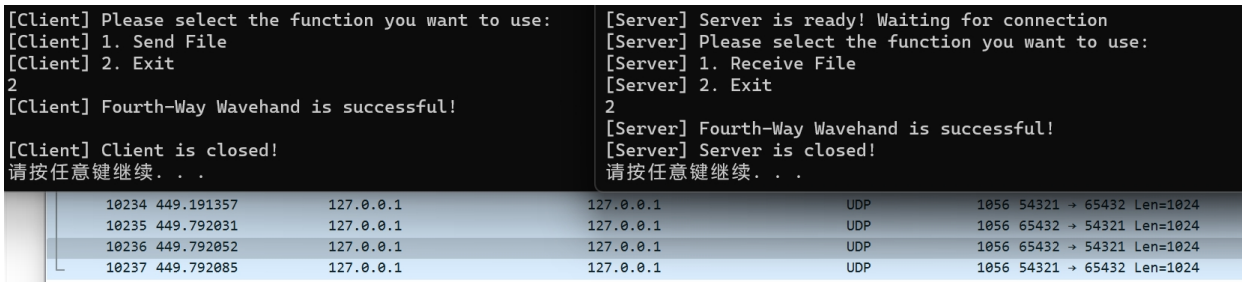
右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。



这是 `wireshark` 抓取的发送的数据包。



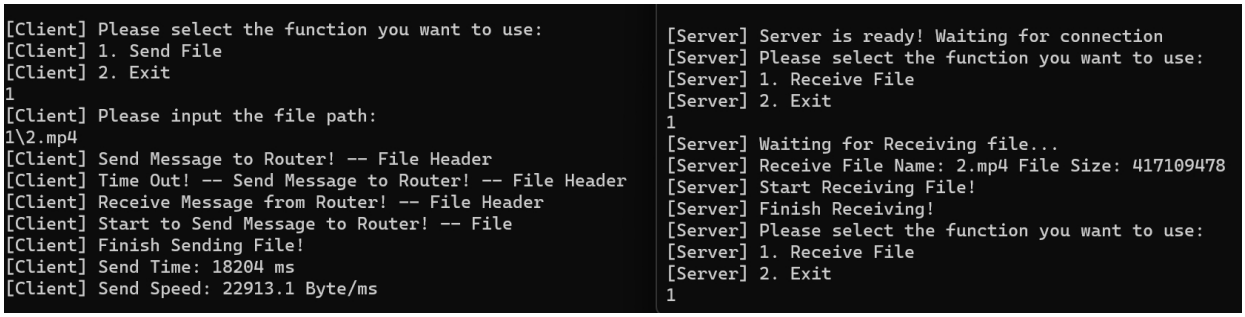
接着断开连接，可以看到 `wireshark` 上抓取到我们设计的四次挥手相关信息。



（2）中文件

此处以大小为 417109478 Byte 的文件 `2.mp4` 为例。

传输完成，可以看到，累计用时 18204 ms，吞吐率为 22913.1 $Byte/ms$。

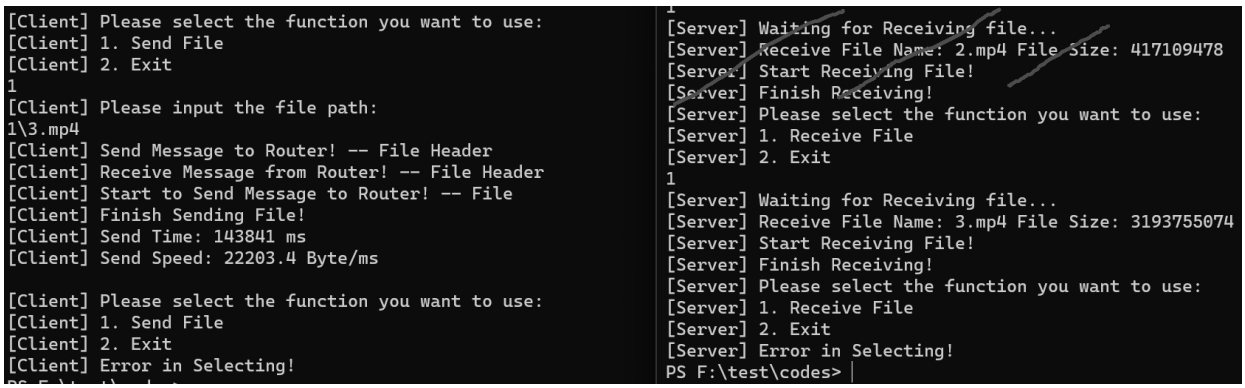


右键查看文件属性，可以看到传输前后文件大小没有发生改变；打开文件，可以看到文件成功打开，说明传输无误。

（3）大文件

此处以大小为 3193755074 Byte 的文件 `3.mp4` 为例。

传输完成，可以看到，累计用时 143841ms，吞吐率为 22203.4 $Byte/ms$。



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。

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。此处仅以上文提到的中文件传输为例进行测试说明。

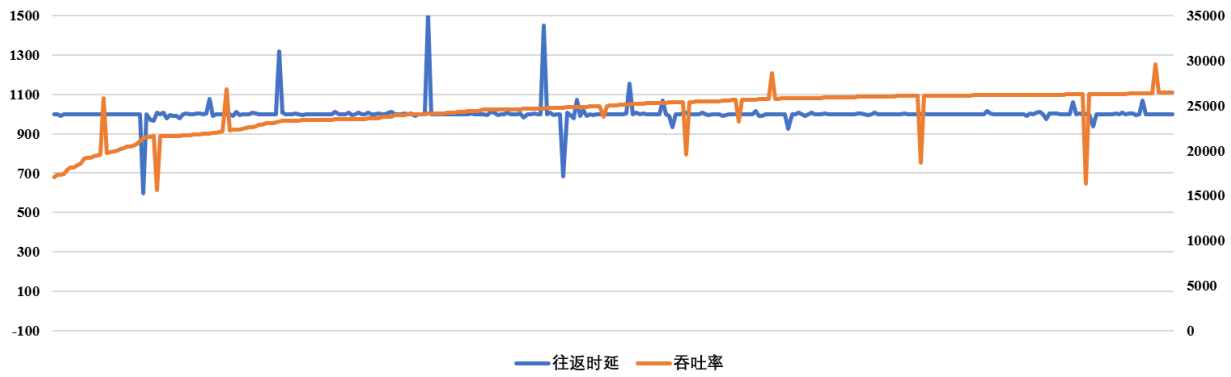
传输完成，可以看到，累计用时 1.48844e+06 ms，吞吐率为 280.232 $Byte/ms$。

```
PS F:\test\codes> .\client.exe 10.130.51.222 10.130.108.86
[Client] Initializing Socket DLL is successful!
[Client] Creating Socket is successful!
[Client] Binding Socket to port 54321 is successful!

[Client] Client is ready! Trying to connection
[Client] Third-Way Handshake is successful!

[Client] Please select the function you want to use:
[Client] 1. Send File
[Client] 2. Exit
1
[Client] Please input the file path:
1\2.mp4
[Client] Send Message to Router! -- File Header
[Client] Time Out! -- Send Message to Router! -- File Header
[Client] Time Out! -- Send Message to Router! -- File Header
[Client] Receive Message from Router! -- File Header
[Client] Start to Send Message to Router! -- File
[Client] Finish Sending File!
[Client] Send Time: 1.48844e+06 ms
[Client] Send Speed: 280.232 Byte/ms

[Client] Please select the function you want to use:
[Client] 1. Send File
[Client] 2. Exit
2
[Client] Time Out! -- First-Way Wavehand
[Client] Fourth-Way Wavehand is successful!

[Client] Client is closed!
```

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。

（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，借助 `python` 进行数据清洗，然后借助 `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 26000 μs，而实时往返时延稳定在 1000 μs，偶尔会有波动。

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）设置缓冲区，导致无法传输大文件

原本设想设定一个巨大的缓冲区，将文件全部读入其中后再传，全部接收完成后再一起写入新文件。但是这样一来浪费空间，二来无法传输大型文件。

改进后，去掉了缓冲区，采取读一点发一点写一点的策略，按照设计的 `MSS` 读取文件，然后写入数据段发送，接收端收到数据后，以 `MSS` 为单位写入新文件，跳过缓冲区的使用。

（三）根据带宽情况，实时调整等待时间

程序原本统一使用宏定义的 `Wait_Time` 作为重传等待时间，但是这样就会导致效率低下等问题。参考教材，修改了重传等待机制，将上一个消息的往返时延作为当前消息的重传等待时间。这样就实现了按照网络带宽实时确定重传等待时间，提高了传输效率。

（四）联机状态下传输慢

在使用局域网联机测试时，传输效率较低，推测可能由于协议数据段较小，每次传输数据较少，需要多次传输。