

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 8168Byte，整个数据报大小为 8192Byte。

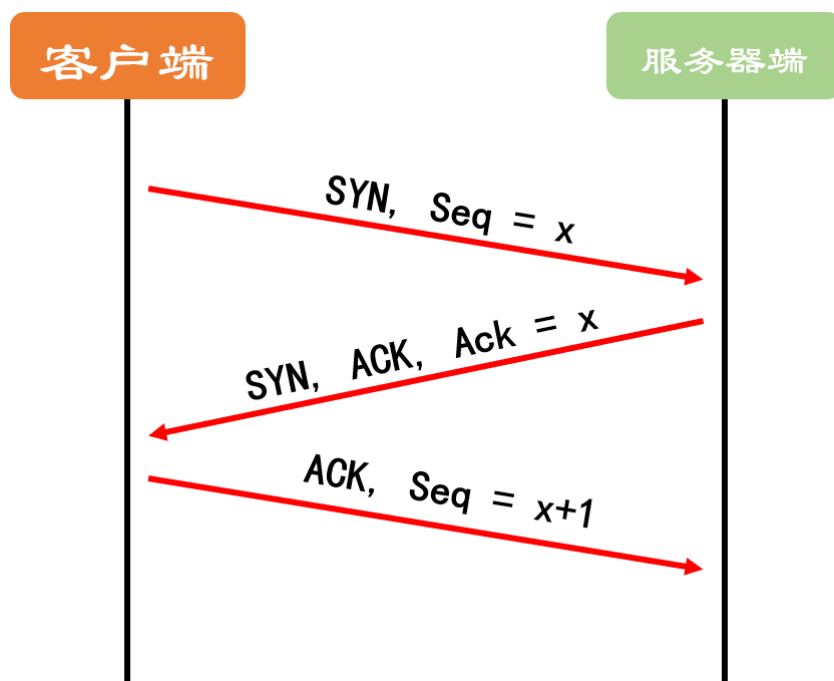
| | | | |
|----|-----------------------|---------------|----|
| 1 | 0 | 15 16 | 31 |
| 2 | ----- | | |
| 3 | | SrcPort | |
| 4 | ----- | | |
| 5 | | DestPort | |
| 6 | ----- | | |
| 7 | | Seq | |
| 8 | ----- | | |
| 9 | | Ack | |
| 10 | ----- | | |
| 11 | | Length | |
| 12 | ----- | | |
| 13 | | Flag | |
| 14 | ----- | | |
| 15 | | Data | |
| 16 | ----- | | |
| 17 | | | |
| 18 | Flag | | |
| 19 | 0 | 1 2 3 4 | 15 |
| 20 | ----- | | |
| 21 | FIN CFH ACK SYN | | |
| 22 | ----- | | |

(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

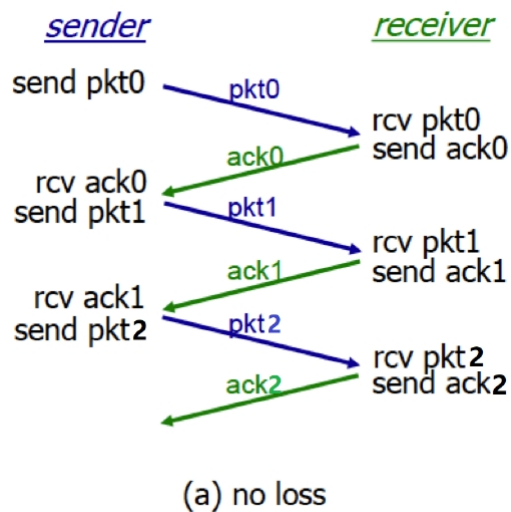
按照实验要求，本次实验需要使用停等机制进行流量控制，即发送方发送完成后，要收到接收端返回的对应 ACK 确认报文才能进行下一步传输。按照前文叙述，应接收的确认报文的 Ack 等于发送的序列号 Seq。

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。

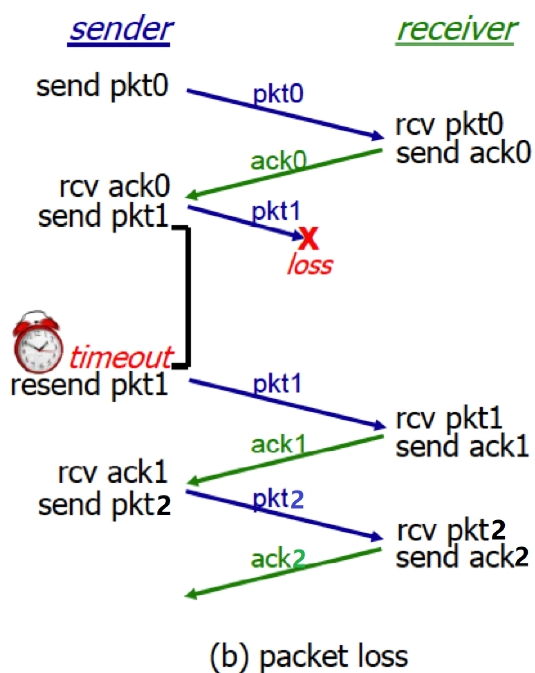
(1) 理想情况

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：



(2) 数据包发送丢失

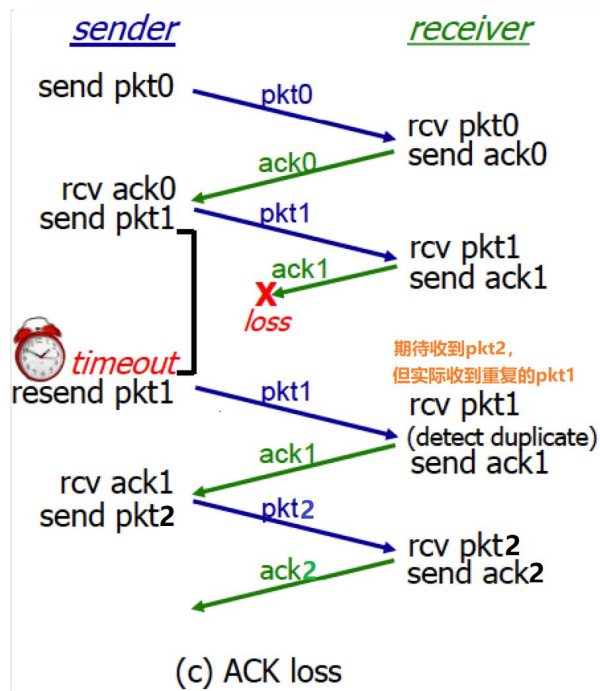
数据包正常发送，但发生数据包丢失问题。示意图如下：



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

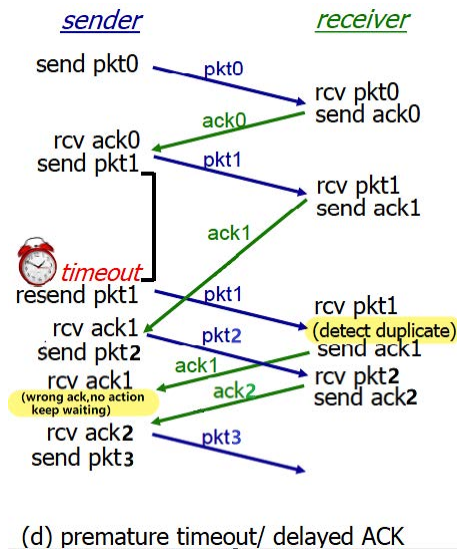
数据包正常发送，但发生数据包丢失问题。示意图如下：



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

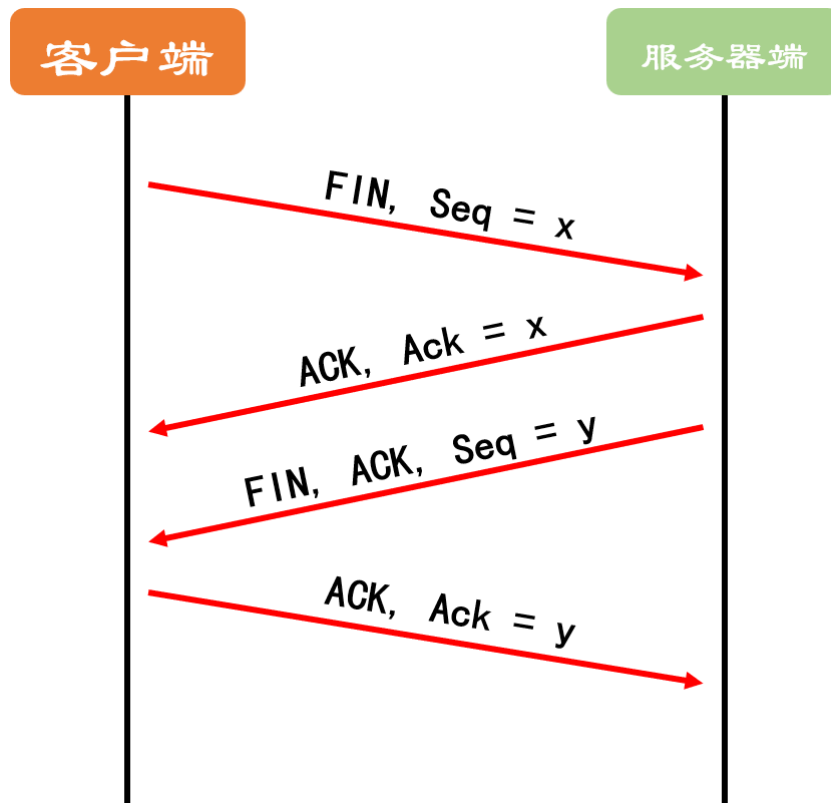
数据包正常发送，但是接收端或发送端由于种种原因发生数据包失序问题。示意图如下：



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。

```

1  #define FIN 0b1
2  #define CFH 0b10
3  #define ACK 0b100
4  #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。

```

1  #pragma pack(1)
2  struct Message
3  {
4      uint32_t SrcPort;
5      uint32_t DstPort;
6      uint32_t Seq;
7      uint32_t Ack;
8      uint32_t Length;
9      uint16_t Flag;
10     uint16_t Check;
11     char Data[MSS];
12
13     Message() : SrcPort(0), DstPort(0), Seq(0), Ack(0), Length(0), Flag(0),
14     Check(0) { memset(this->Data, 0, MSS); }
15     void Set_CFH() { this->Flag |= CFH; }
16     bool Is_CFH() { return this->Flag & CFH; }
17     void Set_ACK() { this->Flag |= ACK; }
18     bool Is_ACK() { return this->Flag & ACK; }
19     void Set_SYN() { this->Flag |= SYN; }
20     bool Is_SYN() { return this->Flag & SYN; }
21     void Set_FIN() { this->Flag |= FIN; }
22     bool Is_FIN() { return this->Flag & FIN; }
23     bool CheckValid();
24     void Print_Message();
25 };
26 #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。

```

1  void Message::Set_Check()
2  {
3      this->Check = 0;
4      uint32_t sum = 0;
5      uint16_t *p = (uint16_t *)this;
6      for (int i = 0; i < sizeof(*this) / 2; i++)
7      {
8          sum += *p++;
9          while (sum >> 16)
10         {
11             sum = (sum & 0xffff) + (sum >> 16);
12         }
13     }

```

```

14     this->Check = ~(sum & 0xffff);
15 }
16 bool Message::CheckValid()
17 {
18     uint32_t sum = 0;
19     uint16_t *p = (uint16_t *)this;
20     for (int i = 0; i < sizeof(*this) / 2; i++)
21     {
22         sum += *p++;
23         while (sum >> 16)
24         {
25             sum = (sum & 0xffff) + (sum >> 16);
26         }
27     }
28     return (sum & 0xffff) == 0xffff;
29 }

```

(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。

```

1  SOCKET ClientSocket;
2  SOCKADDR_IN ClientAddr;
3  string ClientIP = "127.0.0.1";
4  int ClientAddrLen = sizeof(ClientAddr);
5
6  bool Client_Initial()
7  {
8      WSASStartup(MAKEWORD(2, 2), &wsaData);
9      if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2)
10     {
11         perror("[Client] Error in Initializing Socket DLL!\n");
12         exit(EXIT_FAILURE);
13     }
14     cout << "[Client] "<< "Initializing Socket DLL is successful!\n";
15     ClientSocket = socket(AF_INET, SOCK_DGRAM, 0);
16     unsigned long on = 1;
17     ioctlsocket(ClientSocket, FIONBIO, &on);
18     if (ClientSocket == INVALID_SOCKET)
19     {
20         cout << "[Client] "<< "Error in Creating Socket!\n";
21         exit(EXIT_FAILURE);
22         return false;
23     }
24     cout << "[Client] "<< "Creating Socket is successful!\n";
25     ClientAddr.sin_family = AF_INET;
26     ClientAddr.sin_port = htons(Client_Port);
27     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)

```



```

25         {
26             int re = Send(con_msg[0]);
27             msg1_Send_Time = clock();
28             if (re > 0)
29             {
30                 cout << "[Client] " << "Time Out! -- Send Message to Router!
-- First-Way Handshake" << endl;
31             }
32         }
33     }
34 }
35 // * Third-way Handshake
36 con_msg[2].Ack = con_msg[1].Seq;
37 con_msg[2].Seq = ++Seq;
38 con_msg[2].Set_ACK();
39 re = Send(con_msg[2]);
40 if (re > 0) {}
41 cout<< "[Client] " << "Third-way Handshake is successful!" << endl << endl;
42 return true;
43 }

```

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`。

```

1 int Send(Message &msg)
2 {
3     msg.SrcPort = Client_Port;
4     msg.DstPort = Router_Port;
5     msg.Set_Check();
6     return sendto(ClientSocket, (char *)&msg, sizeof(msg), 0, (SOCKADDR
*)&RouterAddr, RouterAddrLen);
7 }

```

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。

```

1 strcpy(file_name, "");
2 size_t found = file_path.find_last_of("\\");
3 string file_name = file_path.substr(found + 1);

```

```

4 ifstream file(file_path, ios::binary);
5 if (!file.is_open())
6 {
7     cout << "[Client] " << "Error in Opening File!" << endl;
8     exit(EXIT_FAILURE);
9 }
10 file.seekg(0, ios::end);
11 file_length = file.tellg();
12 file.seekg(0, ios::beg);
13 if(file_length > pow(2,32))
14 {
15     cout<<"[Client] " << "File is too large!" << endl;
16     exit(EXIT_FAILURE);
17 }

```

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。

```

1 Message send_msg;
2 strcpy(send_msg.Data, file_name.c_str());
3 send_msg.Data[strlen(send_msg.Data)] = '\0';
4 send_msg.Length = file_length;
5 send_msg.Seq = ++Seq;
6 send_msg.Set_CFH();
7 float last_time;
8 int re = Send(send_msg);
9 float msg1_Send_Time = clock();
10 if (re > 0)
11 {
12     cout << "[Client] " << "Send Message to Router! -- File Header" << endl;
13 }
14
15 while(true)
16 {
17     Message tmp;
18     if (recvfrom(ClientSocket, (char *)&tmp, sizeof(tmp), 0, (SOCKADDR
19 *)&RouterAddr, &RouterAddrLen) > 0)
20     {
21         cout << "[Client] " << "Receive Message from Router! -- File Header" <<
22         endl;
23         if (tmp.Is_ACK() && tmp.CheckValid() && tmp.Seq == Seq + 1)
24         {
25             Seq = tmp.Seq;
26             last_time = clock() - msg1_Send_Time;
27             break;
28         }
29         else if (tmp.CheckValid() && tmp.Seq != Seq + 1)
30         {
31             Message reply_msg;
32             reply_msg.Ack = tmp.Seq;
33             reply_msg.Set_ACK();

```

```

32         if(Send(reply_msg)>0)
33         {
34             cout<<"!Repeatedly! [Client]"<< "Receive Seq = "<<tmp.Seq<<"
Reply Ack = "<<reply_msg.Ack<<endl;
35         }
36     }
37 }
38 else if (clock()-msg1_Send_Time > last_time)
39 {
40     int re = sendto(ClientSocket, (char *)&send_msg, sizeof(send_msg), 0,
(SOCKADDR *)&RouterAddr, RouterAddrLen);
41     msg1_Send_Time = clock();
42     if (re > 0)
43     {
44         cout <<"[Client] "<< "Time Out! -- Send Message to Router! -- File
Header" << endl;
45     }
46     else
47     {
48         cout<<"[Client] "<<"Error in Sending Message! -- File Header"
<<endl;
49         exit(EXIT_FAILURE);
50     }
51 }
52 }

```

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

```

1 struct timeval complete_time_start, complete_time_end;
2 gettimeofday(&complete_time_start, NULL);
3 float complete_time = clock();
4 int complete_num = file_length / MSS;
5 int last_length = file_length % MSS;
6 cout <<"[Client] "<< "Start to Send Message to Router! -- File" << endl;
7 for(int i=0;i<=complete_num;i++)
8 {
9     Message data_msg;
10    if (i!=complete_num)
11    {
12        file.read(data_msg.Data, MSS);
13        data_msg.Length = MSS;
14        data_msg.Seq = ++Seq;
15        int re = Send(data_msg);
16        struct timeval every_time_start, every_time_end;
17        long long every_time_usec;
18        gettimeofday(&every_time_start, NULL);

```

```

19     float time = clock();
20     if (re > 0)
21     {
22         Message tmp;
23         while(true)
24         {
25             if (recvfrom(ClientSocket, (char *)&tmp, sizeof(tmp), 0,
(SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
26             {
27                 if (tmp.Is_ACK() && tmp.CheckValid() && tmp.Seq == Seq +
1)
28                 {
29                     Seq = tmp.Seq;
30                     gettimeofday(&every_time_end, NULL);
31                     every_time_usec = (every_time_end.tv_usec -
every_time_start.tv_usec);
32                     if(i % 1 == 0)
33                     {
34                         gettimeofday(&complete_time_end, NULL);
35                         long long complete_time_usec =
(complete_time_end.tv_usec - complete_time_start.tv_usec);
36                         time_txt << (complete_time_usec) << "," <<
(every_time_usec ) << "," << ((double)(MSS * i)/(complete_time_usec)*1000)
<<endl;
37                     }
38                     break;
39                 }
40                 else if (tmp.CheckValid() && tmp.Seq != Seq + 1)
41                 {
42                     Message reply_msg;
43                     reply_msg.Ack = tmp.Seq;
44                     reply_msg.Set_ACK();
45                     if(Send(reply_msg)>0)
46                     {
47                         cout<<"!Repeatedly! [Client]"<< "Receive Seq = "
<<tmp.Seq<<" Reply Ack = "<<reply_msg.Ack<<endl;
48                     }
49                 }
50             }
51             else if (clock()-time > every_time_usec)
52             {
53                 int re = sendto(ClientSocket, (char *)&data_msg,
sizeof(data_msg), 0, (SOCKADDR *)&RouterAddr, RouterAddrLen);
54                 time = clock();
55                 if (re > 0)
56                 {
57                     cout << "[Client] " << "Time Out! -- Send Message to
Router! Part " << i << "-- File" << endl;
58                 }
59                 else
60                 {

```

```

61         cout<<"[Client] "<<"Error in Sending Message! Part "
<<i<<" -- File"<<endl;
62         exit(EXIT_FAILURE);
63     }
64 }
65 }
66 }
67 }
68 else
69 {
70     Message data_msg;
71     file.read(data_msg.Data, last_length);
72     data_msg.Length = last_length;
73     data_msg.Seq = ++Seq;
74     int re = Send(data_msg);
75     // float every_time_start = clock();
76     struct timeval every_time_start, every_time_end;
77     long long every_time_usec;
78     gettimeofday(&every_time_start, NULL);
79     float time = clock();
80     if (re > 0)
81     {
82         Message tmp;
83         while(true)
84         {
85             if (recvfrom(ClientSocket, (char *)&tmp, sizeof(tmp), 0,
(SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
86             {
87                 if (tmp.Is_ACK() && tmp.CheckValid() && tmp.Seq == Seq +
1)
88                 {
89                     Seq = tmp.Seq;
90                     gettimeofday(&every_time_end, NULL);
91                     every_time_usec = (every_time_end.tv_usec -
every_time_start.tv_usec);
92                     if(i % 1 == 0)
93                     {
94                         gettimeofday(&complete_time_end, NULL);
95                         long long complete_time_usec =
(complete_time_end.tv_usec - complete_time_start.tv_usec);
96                         time_txt << (complete_time_usec) << "," <<
(every_time_usec) << "," << ((double)
(last_length)/(complete_time_usec)*1000);
97                     }
98                     break;
99                 }
100                 else if (tmp.CheckValid() && tmp.Seq != Seq + 1)
101                 {
102                     Message reply_msg;
103                     reply_msg.Ack = tmp.Seq;
104                     reply_msg.Set_ACK();
105                     if(Send(reply_msg)>0)

```

```

106         {
107             cout<<"!Repeatedly! [Client]"<< "Receive Seq = "
108             <<tmp.Seq<<" Reply Ack = "<<reply_msg.Ack<<endl;
109         }
110     }
111     else if (clock()-time > every_time_usec)
112     {
113         int re = sendto(ClientSocket, (char *)&data_msg,
114             sizeof(data_msg), 0, (SOCKADDR *)&RouterAddr, RouterAddrLen);
115         time = clock();
116         if (re > 0)
117         {
118             cout <<"[Client] " << "Time Out! -- Send Message to
119             Router! Part " << i << "-- File" << endl;
120         }
121         else
122         {
123             cout<<"[Client] " <<"Error in Sending Message! Part "
124             <<i<<" -- File"<<endl;
125             exit(EXIT_FAILURE);
126         }
127     }
128 }

```

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。

为避免报告冗长，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

```

1 void Disconnect() // * Client端主动断开连接
2 {
3     Message discon_msg[4];

```

```

4      // * First-way Wavehand
5      discon_msg[0].Seq = ++Seq;
6      discon_msg[0].Set_FIN();
7      int re = Send(discon_msg[0]);
8      float dismsg0_Send_Time = clock();
9      if (re > 0) {}
10     // * Second-way Wavehand
11     while (true)
12     {
13         if(discon_msg[0].Seq < Seq + 1)
14         {
15             continue;
16         }
17         if (recvfrom(ClientSocket, (char *)&discon_msg[1],
18 sizeof(discon_msg[1]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
19         {
20             if (!(discon_msg[1].Is_ACK() && discon_msg[1].CheckValid() &&
21 discon_msg[1].Seq == Seq + 1 && discon_msg[1].Ack == discon_msg[0].Seq))
22             {
23                 cout << "[Client] " << "Error Message!" << endl;
24                 exit(EXIT_FAILURE);
25             }
26             Seq = discon_msg[1].Seq;
27             break;
28         }
29         if ((clock() - dismsg0_Send_Time) > wait_Time)
30         {
31             cout << "[Client] " << "Time Out! -- First-way wavehand" << endl;
32             int re = Send(discon_msg[0]);
33             dismsg0_Send_Time = clock();
34             if (re > 0) {}
35         }
36     }
37     // * Third-way Wavehand
38     while (true)
39     {
40         if (recvfrom(ClientSocket, (char *)&discon_msg[2],
41 sizeof(discon_msg[2]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
42         {
43             if (!(discon_msg[2].Is_ACK() && discon_msg[2].Is_FIN() &&
44 discon_msg[2].CheckValid() && discon_msg[2].Seq == Seq + 1 && discon_msg[2].Ack
45 == discon_msg[1].Seq))
46             {
47                 cout << "[Client] " << "Error Message!" << endl;
48                 exit(EXIT_FAILURE);
49             }
50             Seq = discon_msg[2].Seq;
51             break;
52         }
53     }
54     // * Fourth-way Wavehand
55     discon_msg[3].Ack = discon_msg[2].Seq;

```

```

51     discon_msg[3].Set_ACK();
52     discon_msg[3].Seq = ++Seq;
53     re = Send(discon_msg[3]);
54     if (re > 0) {}
55     cout << "[Client] " << "Fourth-Way Wavehand is successful!" << endl <<
endl;
56     Wait_Exit();
57     return;
58 }
59 void Wait_Exit()
60 {
61     Message exit_msg;
62     float exit_msg_time = clock();
63     while (clock() - exit_msg_time < 2 * wait_Time)
64     {
65         if (recvfrom(ClientSocket, (char *)&exit_msg, sizeof(exit_msg), 0,
(SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
66         {
67             Seq = exit_msg.Seq;
68             exit_msg.Ack = exit_msg.Seq;
69             exit_msg.Set_ACK();
70             exit_msg.Seq = ++Seq;
71             Send(exit_msg);
72         }
73     }
74     closesocket(ClientSocket);
75     WSACleanup();
76     cout << "[Client] " << "Client is closed!" << endl;
77     system("pause");
78 }

```

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

```

1 void Disconnect() // * Router端主动断开连接
2 {
3     Message discon_msg[4];
4     while (true)
5     {
6         // * First-Way Wavehand
7         if (recvfrom(ServerSocket, (char *)&discon_msg[0],
sizeof(discon_msg[0]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
8         {
9             if (!(discon_msg[0].Is_FIN() && discon_msg[0].CheckValid() &&
discon_msg[0].Seq == Seq + 1))
10            {

```



```

11         cout << "[Server] " << "Error Message!" << endl;
12         exit(EXIT_FAILURE);
13     }
14     Seq = discon_msg[0].Seq;
15 }
16 // * Second-way wavehand
17 discon_msg[1].Ack = discon_msg[0].Seq;
18 discon_msg[1].Seq = ++Seq;
19 discon_msg[1].Set_ACK();
20 int re = Send(discon_msg[1]);
21 if (re > 0) {}
22 break;
23 }
24 // * Third-way wavehand
25 discon_msg[2].Ack = discon_msg[1].Seq;
26 discon_msg[2].Seq = ++Seq;
27 discon_msg[2].Set_ACK();
28 discon_msg[2].Set_FIN();
29 int re = Send(discon_msg[2]);
30 float dismsg3_Send_Time = clock();
31 if (re > 0) {}
32 // * Fourth-way wavehand
33 while (true)
34 {
35     if (recvfrom(ServerSocket, (char *)&discon_msg[3],
sizeof(discon_msg[3]), 0, (SOCKADDR *)&RouterAddr, &RouterAddrLen) > 0)
36     {
37         if (discon_msg[3].Seq < Seq + 1)
38         {
39             continue;
40         }
41         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))
42         {
43             cout << "[Server] " << "Error Message!" << endl;
44             exit(EXIT_FAILURE);
45         }
46         Seq = discon_msg[3].Seq;
47         cout << "[Server] " << "Fourth-way wavehand is successful!" <<
endl;
48         break;
49     }
50     if ((clock() - dismsg3_Send_Time) > wait_Time)
51     {
52         int re = Send(discon_msg[2]);
53         dismsg3_Send_Time = clock();
54         if (re > 0)
55         {
56             cout << "[Server] " << "Time Out! -- Send Message to Router! --
Third-way wavehand" << endl;
57         }
58     }

```

```
59     }
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。

```
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
```

```
[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
```

*Adapter for loopback traffic capture

件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(I) 帮助(H)

dst == 127.0.0.1 or ip.src == 127.0.0.1) and (udp.srcport == 54321 or udp.srcport == 65432 or udp.dstport == 54321 or udp.dstport == 65432)

| | Time | Source | Destination | Protocol | Length | Info |
|-----|-----------|-----------|-------------|----------|--------|------------------------|
| 376 | 25.533341 | 127.0.0.1 | 127.0.0.1 | UDP | 1056 | 54321 → 65432 Len=1024 |
| 377 | 25.533415 | 127.0.0.1 | 127.0.0.1 | UDP | 1056 | 65432 → 54321 Len=1024 |
| 378 | 25.533440 | 127.0.0.1 | 127.0.0.1 | UDP | 1056 | 54321 → 65432 Len=1024 |

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

设置好路由转发，丢包率 10%，时延 100ms。

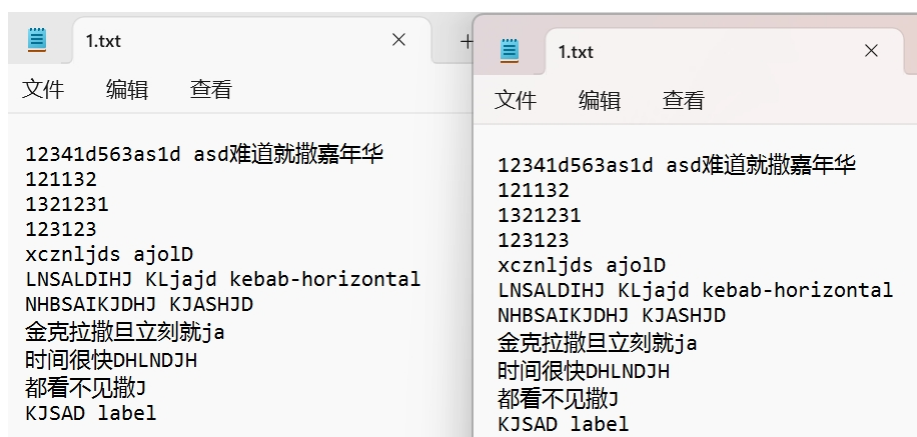


传输完成，可以看到，累计用时 120 ms ，吞吐率为 1.825 Byte/ms 。

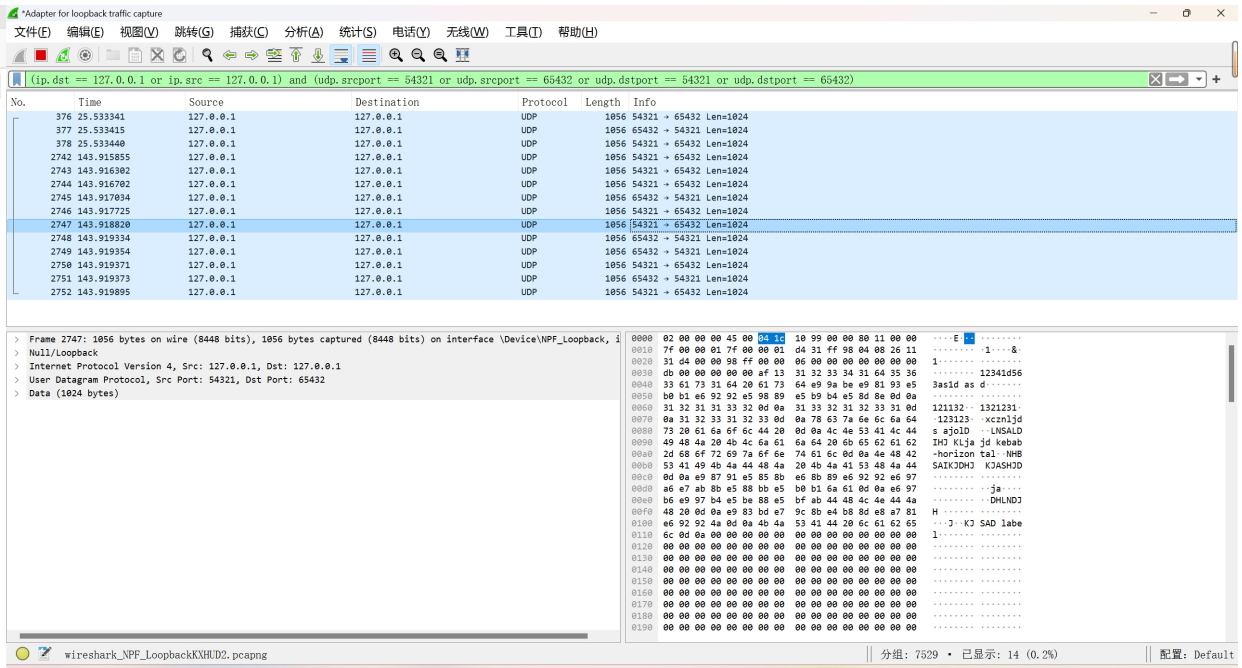
```
[Client] Please input the file path:
1\1.txt
[Client] Send Message to Router! -- File Header
[Client] Time Out! -- Send Message to Router! -- File Header
[Client] Time Out! -- Send Message to Router! -- File Header
[Client] Receive Message from Router! -- File Header
[Client] Start to Send Message to Router! -- File
[Client] Finish Sending File!
[Client] Send Time: 120 ms
[Client] Send Speed: 1.825 Byte/ms

[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，不使用路由转发。

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

| | | | |
|---|--|---|--|
| <pre>[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] Receive Message from Router! -- File Header [Client] Start to Send Message to Router! -- File [Client] Finish Sending File! [Client] Send Time: 18204 ms [Client] Send Speed: 22913.1 Byte/ms</pre> | | <pre>[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: 2.mp4 File Size: 417109478 [Server] Start Receiving File! [Server] Finish Receiving! [Server] Please select the function you want to use: [Server] 1. Receive File [Server] 2. Exit 1</pre> | |
|---|--|---|--|

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，不使用路由转发。

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

```

[Client] Please select the function you want to use:
[Client] 1. Send File
[Client] 2. Exit
1
[Client] Please input the file path:
1\3.mp4
[Client] 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: 143841 ms
[Client] Send Speed: 22203.4 Byte/ms

[Client] Please select the function you want to use:
[Client] 1. Send File
[Client] 2. Exit
[Client] Error in Selecting!

[Server] Waiting for Receiving file...
[Server] Receive File Name: 2.mp4 File Size: 417109478
[Server] Start Receiving File!
[Server] Finish Receiving!
[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: 3.mp4 File Size: 3193755074
[Server] Start Receiving File!
[Server] Finish Receiving!
[Server] Please select the function you want to use:
[Server] 1. Receive File
[Server] 2. Exit
[Server] Error in Selecting!
PS F:\test\codes>

```

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!

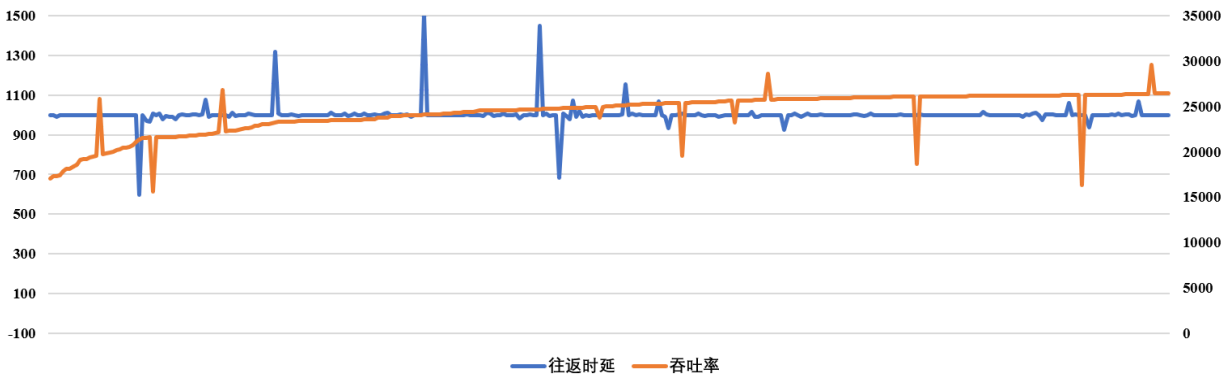
```

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。

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，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 $26000 \mu s$ ，而实时往返时延稳定在 $1000 \mu s$ ，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。