计算机网络作业3-1实验报告：基于UDP服务设计可靠传输协议并编程实现

赵元鸣 计算机科学与技术 2211757

实验要求

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、确认重传等。流量控制采用停等机制，完成给定测试文件的传输。

实验设计

实验原理概述

本实验的核心原理是基于 **UDP 协议**，实现了 **差错检测** 和 **确认重传** 等机制，用于确保在不可靠的网络环境中，数据能够有效传输，减少丢包和错误的发生。

1. **UDP 协议**

UDP（用户数据报协议）是一个面向无连接的协议，与 TCP 相比，它没有连接的建立和拆除过程，也没有流量控制和拥塞控制机制。UDP 在传输数据时不保证数据的可靠性，可能会发生数据丢失、重复接收或乱序等问题，因此需要在应用层进行差错检测、确认重传等机制。

UDP 特性：

- **无连接**：发送数据时不需要建立连接，发送端和接收端不需要相互确认。
- **无可靠性保障**：数据包可能会丢失，接收端无法确认数据是否正确到达。
- **较低的开销**：没有像 TCP 那样复杂的控制机制，适合对实时性要求较高的场景。

2. **差错检测**

差错检测是为了确保传输过程中的数据没有损坏。在 UDP 协议中，应用层需要自行处理数据传输的差错检测。

在本实验中，差错检测通常通过 **校验和**（Checksum）机制实现。每个数据包都有一个校验和字段，用于检测数据包在传输过程中是否发生了错误。

校验和计算：

- 校验和的计算方式是将数据包中的所有 16 位字节加和，再取反得到校验值。接收方收到数据后，再对接收到的包进行相同的计算，如果计算值为零，则认为数据没有发生错误。
- 如果计算出的校验和不为零，说明数据在传输过程中发生了损坏，接收方会丢弃该数据包并请求重新发送。

3. **确认重传**

由于 UDP 本身不提供确认和重传机制，因此在本实验中实现了一个简化的 **确认重传** 机制，以保证数据可靠传输。该机制基于 **序列号** 和 **ACK 确认** 机制，确保数据包按顺序且不丢失地传输。

具体实现步骤：

- **序列号管理：**
  - 每个数据包都带有一个 **序列号**，接收方通过检查包的序列号来确定数据包的顺序。
  - 如果接收方收到一个序列号与预期不符的包（如丢失了前一个包），则丢弃该数据包并请求重传。
- **确认机制：**
  - 接收方接收到正确的数据包后，会发送一个 **ACK**（确认包）给发送方，告知发送方数据包已成功接收。
  - 如果发送方在规定时间内未收到确认响应，则会重新发送该数据包。
- **超时重传：**
  - 为了处理丢失或延迟的确认，发送方在发送数据包时会启动一个定时器。如果在超时内没有收到确认，发送方会重新发送该数据包。
- **丢包和乱序处理：**
  - 接收方根据序列号对数据包进行排序和重组。如果接收到的包是乱序的，接收方会丢弃无法按顺序接收的包，直到前一个包到达。

4. 实验步骤

在实验中，数据包的传输流程大致如下：

（1） 建立连接

- **客户端与服务器的初始化**
  - **客户端** 和 **服务器** 初始化网络套接字，准备接收和发送数据。
  - 在客户端启动之前，服务器端需要监听客户端的连接请求（通常是通过 `recvfrom()` 函数接收数据包）。
- **第一次握手：客户端发送连接请求**
  - 客户端发送一个 **SYN 包**，标志着请求建立连接。
  - 包含字段：`flag = SYN`，并带有计算出的 **校验和**。
  - 客户端发送后进入等待状态，期望从服务器收到确认包（即第二次握手的 ACK 包）。
- **第二次握手：服务器确认连接**
  - **服务器** 收到客户端的 SYN 包后，发送一个 **SYN-ACK 包**，以确认收到连接请求。
  - 包含字段：`flag = ACK` 或 `flag = ACK_SYN`，并带有计算出的 **校验和**。
  - 如果校验和正确且数据无误，服务器会确认并等待客户端发送第三次握手。
- **第三次握手：客户端确认连接**
  - **客户端** 收到服务器的 SYN-ACK 包后，发送一个 **ACK 包**，确认建立连接。
  - 包含字段：`flag = ACK`，并带有计算出的 **校验和**。

- 客户端发送后，表示连接建立成功。

（2）数据传输

- 一旦连接建立，客户端可以开始发送数据包，服务器端接收并确认。
  - 每个数据包都会包含 **序列号** 和 **校验和**。
  - 服务器接收每个数据包时会验证序列号，确保按序接收。如果序列号不对，服务器会丢弃该数据包并要求客户端重新发送。
  - 服务器收到正确的数据包后，发送一个 **ACK 包**，确认接收到的数据包，并可能通知客户端继续发送下一个数据包。
  - 如果客户端在超时时间内没有收到确认，则会重传相应的数据包。

（3）结束传输：客户端和服务器关闭连接

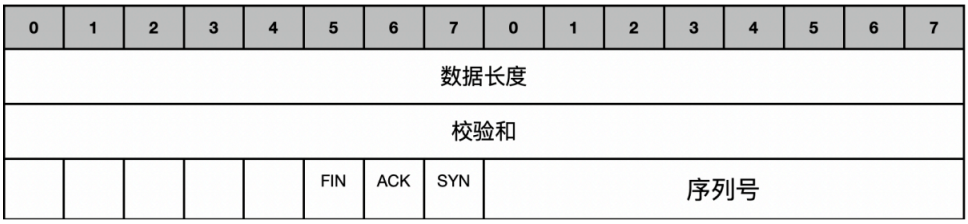
- 数据传输完成后，客户端或服务器可以发起关闭连接的请求。
  - **客户端或服务器** 发送一个 **FIN 包**（标志连接结束），等待对方确认。
  - **服务器或客户端** 接收到 FIN 包后，发送 **FIN-ACK 包** 确认。
  - 对方收到确认后，发送 **ACK 包** 以完成连接断开。
  - 此时，双方连接关闭，数据传输结束。

错误处理和重传

- 在传输过程中，如果出现丢包或数据错误，接收方会丢弃损坏的包，并通过 **ACK 包** 请求发送方重新发送该数据包。
- 客户端根据 **ACK 包** 确认数据是否成功接收，如果在指定的超时时间内未收到确认，客户端将重发数据包，直到接收到正确的 ACK 包。

协议设计

1. 报文格式



- 报文头格式

```
| 数据长度 (16 位) | 校验和 (16 位) | 标志位 (8 位) | 序号 (8 位) |
|------------------|----------------|---------------|-------------|
| 16 位            | 16 位          | 8 位          | 8 位        |
```

(1) 数据长度（16 位）：

- 这部分记录数据区的大小（即数据字段的长度）。它占用了报文头的前 16 位，用于告知接收方该报文数据部分的长度。

(2) 校验和（16 位）：

- 这部分用于检查数据在传输过程中是否发生了错误。接收方可以通过对报文头和数据部分进行校验和运算，确保数据的完整性。
- 校验和的计算方法通常是将报文头和数据的每两个字节（16 位）进行求和，如果计算结果的高位有进位，则需要处理进位，最后取反得到校验和。

(3) 标志位（8 位）：

- 这部分表示报文的标志位，使用最低的 3 位分别表示以下几个控制标志：
  - **FIN（1 位）**：标志连接是否关闭。1 表示连接关闭，0 表示连接未关闭。
  - **ACK（1 位）**：确认标志位，用于确认接收到的报文。如果该位为 1，表示这是一个确认包。
  - **SYN（1 位）**：同步标志位，用于建立连接时进行同步。1 表示这是一个同步包，0 表示不是同步包。
- 标志位的 4-7 位（即高4位）没有使用，是预留位为未来扩展保留。

(4) 序号（8 位）：

- 序号用于标识数据包的顺序。它的范围是 0 到 255，因此会循环使用。这个序号在客户端和服务器之间保持一致，确保数据包按顺序传输。
- 每次发送数据包时，发送方会增加序号，接收方通过序号判断是否丢失数据包或出现乱序情况。

2. 连接与断开

(1)三次握手连接：

- 客户端向服务端发送数据包，其中SYN=1，ACK=0，FIN=0
- 服务端接受到数据包后，向客户端发送SYN=0，ACK=1，FIN=0
- 客户端再次接收到数据包后，向服务端发送SYN=1，ACK=1，FIN=0
- 服务端接收到数据包后，连接成功建立，可以进行数据传输

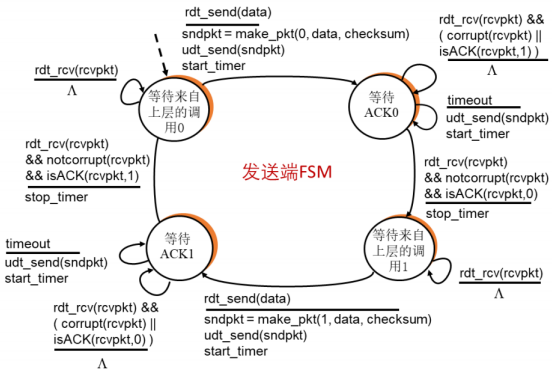
(2) 四次挥手断开：

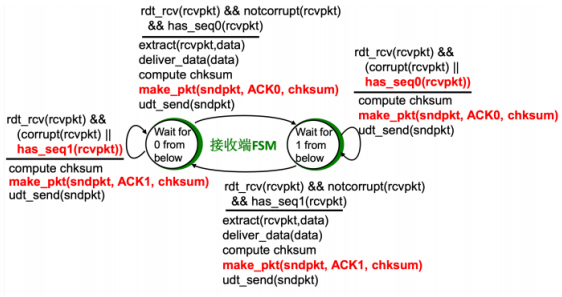
- 客户端向服务端发送数据包，其中SYN=0，ACK=0，FIN=1
- 服务端接受到数据包后，向客户端发送SYN=0，ACK=1，FIN=0
- 客户端再次接收到数据包后，向服务端发送SYN=0，ACK=1，FIN=1
- 服务端接收到数据包后，向客户端发送SYN=0，ACK=1，FIN=1
- 客户端接收到数据包后，连接成功断开

3.数据传输

发送端和接收端的接收机均采用rdt3.0

■ rdt3.0：发送端状态机





- 在文件传输过程中，文件被分割成多个数据包进行传输。每个数据包都包括数据部分和头部信息。数据头中包含了序列号、校验和、标志位等信息，用于确保数据的正确传输。
- 发送方在发送一个数据包时，接收方会在收到数据包后进行校验，若校验通过，接收方会返回一个包含正确序列号的ACK（确认包），通知发送方该包已成功接收并处理。发送方只有在收到上一包的ACK确认后，才会继续发送下一个数据包。每个数据包都必须经过确认，才能发送下一个数据包。
- 发送方会在发送数据包后开始计时，若在指定时间内没有收到ACK确认，发送方会认为数据包丢失，重新发送该包。这是为了应对网络中出现丢包的情况。
- 接收方在接收到数据包时，会检查包的序列号。如果接收到的包的序列号与上一个包的序列号相同，则认为这是一个重复的数据包（可能由于重传）。此时，接收方会丢弃重复的包，但仍然会向发送方发送该序列号的ACK确认包，以告知发送方该包仍被接收。

1. **终止传输：** 在数据传输完成后，发送方会发送一个带有SYN=1、ACK=1、FIN=1标志的数据包，表示文件传输结束。接收方收到该包后，返回一个ACK=1的确认包，通知发送方文件传输已完成。

通过这些机制，RDT 3.0 实现了可靠的数据传输，确保即使在网络环境不稳定、存在丢包或重复数据包的情况下，数据仍能正确无误地传输到接收方。

代码实现

数据头和标志位信号定义

```
enum PacketFlags : unsigned char {
    INIT = 0x0,      //初始值
    SYN = 0x1,       // SYN = 1
    ACK = 0x2,       // ACK = 1
    ACK_SYN = 0x3,   // SYN = 1, ACK = 1
    FIN = 0x4,       // FIN = 1
    FIN_ACK = 0x5,   // FIN = 1, ACK = 1
    OVER = 0x7       // 结束
};

struct Header
{
    //u_short16位
    u_short checkSum = 0;
    u_short dataLen = 0;
```

```
    PacketFlags flag;

    unsigned char seqNum = 0;

    Header() {
        checkSum = 0;
        dataLen = 0;
        flag =INIT;
        seqNum = 0;
    }
};
```

计算校验和

- 发送方生成检验和

    1. 将发送的进行检验和运算的数据分成若干个16位的位串，每个位串看成一个二进制数
    2. 将首部中的检验和字段置为0，该字段也参与检验和运算。
    3. 对这些16位的二进制数进行1的补码和运算，累加的结果再取反码即生成了检验码。将检验码放入检验和字段中。其中1的补码和运算，即带循环进位的加法，最高位有进位应循环进到最低位

- 接收方校验检验和

    1. 接收方将接收的数据(包括检验和字段)按发送方的同样的方法进行1的补码和运算，累加的结果再取反码。
    2. 校验，如果上步的结果为0，表示传输正确；否则，说明传输有差错。

```
u_short getCheckSum(u_short* tempHeader, int size) {
    int count = (size + 1) / 2;
    u_short* buf = (u_short*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, tempHeader, size);
    u_long checkSum = 0;

    for (int i = 0; i < count; i++) {
        checkSum += buf[i];
        if (checkSum & 0xffff0000) {  // 检查进位
            checkSum &= 0xffff;
            checkSum++;
        }
    }
    return ~(checkSum & 0xffff);
}
```

三次握手

客户端

- 握手实现

```
  int clientShake(Header header,char* Buffer, SOCKET& socketClient,
SOCKADDR_IN& servAddr, int& servAddrlen,int seqOfShake) {

  if (seqOfShake == 1) {
      header.flag = PacketFlags::SYN;
  }
  else if(seqOfShake == 3){
      header.flag = PacketFlags::ACK_SYN;
  }
  else
  {
      cout<<"[client]"<< "sequence of shake error!" << endl;
      return -1;
  }

  header.checkSum = 0;//校验和置零
  u_short temp = getCheckSum((u_short*)&header, sizeof(header));
  header.checkSum = temp;//计算校验和
  if (seqOfShake == 1) {
      memcpy(Buffer, &header, sizeof(header));//将首部放入缓冲区
      if (sendto(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen) == -1)
      {
          return -1;
      }
  }
  else {
      if (sendto(socketClient, (char*)&header, sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen) == -1)
      {
          return -1;
      }
  }


  return 0;}
```

- 连接函数

```
  int Connect(SOCKET& socketClient, SOCKADDR_IN& servAddr, int&
servAddrlen)//三次握手建立连接
  {
  Header header;
  char* Buffer = new char[sizeof(header)];

  u_short checkSum;
```

```
  clientShake(header, Buffer, socketClient, servAddr, servAddrlen,1);
  clock_t start = clock();
  u_long mode = 1;
  ioctlsocket(socketClient, FIONBIO, &mode);

  //第二次握手
  while (recvfrom(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, &servAddrlen) <= 0)
  {
      if (clock() -start > timeLimit)//超时重传第一次握手
      {
          cout<<"[client]"<< "second shake from server time out,resending
first shake" << endl;
          clientShake(header, Buffer, socketClient, servAddr,
servAddrlen,1);
          start = clock();

      }
  }


  //校验和检验
  memcpy(&header, Buffer, sizeof(header));
  if (header.flag == ACK && getCheckSum((u_short*)&header, sizeof(header) ==
0))
  {
      cout<<"[client]"<< "second shake from server received." << endl;
  }
  else
  {
      cout<<"[client]"<< "Connection error!Please restart the client." <<
endl;
      return -1;
  }

  clientShake(header, Buffer, socketClient, servAddr, servAddrlen, 3);
  cout<<"[client]"<< "Successfully connected to server.Permission for
sending data accepted" << endl;
  return 1;}
```

服务端

- 握手实现

```
  int serverShake(Header header,char*Buffer,SOCKET& serverSock, SOCKADDR_IN&
clientAddr, int& clientAddrLen,int seqOfShake)    {
  if (seqOfShake == 2) {
      header.flag = ACK;
```

```
  }
  else {
      cout << "server shake error!" << endl;
      return -1;

  }

  header.checkSum = 0;
  u_short temp = getCheckSum((u_short*)&header, sizeof(header));
  header.checkSum = temp;
  memcpy(Buffer, &header, sizeof(header));
  if (sendto(serverSock, Buffer, sizeof(header), 0, (sockaddr*)&clientAddr,
clientAddrLen) == -1)
  {
      return -1;
  }
  return 0;}
```

- 连接函数

```
  int Connect(SOCKET& serverSock, SOCKADDR_IN& clientAddr, int&
clientAddrLen)
  {

  Header header;
  char* Buffer = new char[sizeof(header)];

  //监听第一次握手
  while (1 == 1)
  {
      if (recvfrom(serverSock, Buffer, sizeof(header), 0,
(sockaddr*)&clientAddr, &clientAddrLen) == -1)
      {
          return -1;
      }
      memcpy(&header, Buffer, sizeof(header));
      if (header.flag == SYN && getCheckSum((u_short*)&header,
sizeof(header)) == 0)
      {
          cout << "first wave from client received." << endl;
          break;
      }
  }

  //发送第二次握手信息

  serverShake(header, Buffer, serverSock, clientAddr, clientAddrLen, 2);
  clock_t start = clock();

  //接收第三次握手
  while (recvfrom(serverSock, Buffer, sizeof(header), 0,
(sockaddr*)&clientAddr, &clientAddrLen) <= 0)
```

```
  {
      if (clock() - start > timeLimit)
      {
          cout << "third wave from client time out,resending second wave" <<
endl;
          serverShake(header, Buffer, serverSock, clientAddr, clientAddrLen,
2);

      }
  }

  Header temp1;
  memcpy(&temp1, Buffer, sizeof(header));
  if (temp1.flag == ACK_SYN && getCheckSum((u_short*)&temp1, sizeof(temp1)
== 0))
  {
      cout << "Connection established!Data transmission is permitted. " <<
endl;
  }
  else
  {
      cout << "Connection error from Client, restart for Client is
required!" << endl;
      return -1;
  }
  return 1;}
```

传输数据(同时检测RTT，吞吐率等数据)

- 发送数据包

```
  void sendDataPack(SOCKET& socketClient, SOCKADDR_IN& servAddr, int&
servAddrlen, char* message, int len, int& order)
  {
  Header header;
  char* buffer = new char[maxBuffSize + sizeof(header)];
  header.dataLen = len;
  header.seqNum = unsigned char(order);  // 序列号
  memcpy(buffer, &header, sizeof(header));
  memcpy(buffer + sizeof(header), message, len);
  u_short temp = getCheckSum((u_short*)buffer, sizeof(header) + len); // 计算
校验和
  header.checkSum = temp;
  memcpy(buffer, &header, sizeof(header));

  auto sendStart = high_resolution_clock::now();  // 记录发送开始时间

  sendto(socketClient, buffer, len + sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen); // 发送
  auto sendEnd = high_resolution_clock::now();  // 记录发送结束时间
  duration<double> sendDuration = sendEnd - sendStart;  // 计算发送时延
```

```
  cout << "[client]Message sent from client---------Information:
length(Bytes): " << len << " seqNum: " << int(header.seqNum) << " flag: " <<
int(header.flag) << endl;
  cout << "Send Round-trip time: " << sendDuration.count() * 1000 << " ms"
<< endl;

  double throughput = (len * 8) / (sendDuration.count());  // 单位为bits/秒
  cout << "Throughput: " << throughput << " bits/sec" << endl;

  clock_t start = clock(); // 初始化start，避免重置
  int retryCount = 0;
  const int maxRetries = 5; // 最大重试次数

  while (retryCount < maxRetries) {
      u_long mode = 1; // socket进入非阻塞模式
      ioctlsocket(socketClient, FIONBIO, &mode);

      int recvResult = recvfrom(socketClient, buffer, maxBuffSize, 0,
(sockaddr*)&servAddr, &servAddrlen);
      if (recvResult > 0) {
          memcpy(&header, buffer, sizeof(header));  // 缓冲区读取信息
          u_short check = getCheckSum((u_short*)&header, sizeof(header));
          if (header.seqNum == u_short(order) && header.flag == ACK) {
              cout << "[client]Send has been confirmed!---------Information:
Flag:" << int(header.flag) << " seqNum:" << int(header.seqNum) << endl;
              break; // 收到ACK，跳出循环
          }
      }

      // 检查超时
      if (clock() - start > timeLimit) {
          retryCount++;
          cout << "[client]Time Out, resending message " << retryCount << "
of " << maxRetries << endl;

          // 重发逻辑
          sendto(socketClient, buffer, len + sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen); // 发送
          start = clock();  // 重新记录发送时间
      }

      // 延时等待，避免占用过多CPU
      Sleep(100); // 或使用 select() 等方法
  }

  if (retryCount >= maxRetries) {
      cout << "[client]Max retries reached, aborting." << endl;
  }

  u_long mode = 0;
  ioctlsocket(socketClient, FIONBIO, &mode); }
```

- 发送文件及结束信息

```
 void send(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen,
char* message, int len)
  {
  int packNum = len / maxBuffSize + (len % maxBuffSize != 0);
  int tempSeqNum = 0;
  auto sendStart = high_resolution_clock::now();  // 记录总发送开始时间
  int totalBytesSent = 0;

  for (int i = 0; i < packNum; i++)
  {
      sendDataPack(socketClient, servAddr, servAddrlen, message + i *
maxBuffSize, i == packNum - 1 ? len - (packNum - 1) * maxBuffSize :
maxBuffSize, tempSeqNum);
      totalBytesSent += (i == packNum - 1 ? len - (packNum - 1) *
maxBuffSize : maxBuffSize);
      tempSeqNum++;
      if (tempSeqNum > 255)
      {
          tempSeqNum = tempSeqNum - 256;
      }
  }

  auto sendEnd = high_resolution_clock::now();  // 记录总发送结束时间
  duration<double> sendDuration = sendEnd - sendStart;  // 计算总发送时延

  // 计算往返时延
  double roundTripTime = sendDuration.count() * 1000;  // 转换为毫秒

  // 输出总发送的字节数、往返时延和吞吐率
  cout << "[client]" << "Total Bytes Sent: " << totalBytesSent << " Bytes"
<< endl;
  cout << "[client]" << "Round-trip time: " << roundTripTime << " ms" <<
endl;

  double throughput = (totalBytesSent * 8) / sendDuration.count();  // 吞吐
率：比特/秒
  cout << "[client]" << "Throughput: " << throughput << " bits/sec" << endl;



  //发送结束信息
  Header header;
  char* Buffer = new char[sizeof(header)];
  header.flag = PacketFlags::OVER;
  header.checkSum = 0;
  u_short temp = getCheckSum((u_short*)&header, sizeof(header));
  header.checkSum = temp;
  memcpy(Buffer, &header, sizeof(header));
  sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr,
servAddrlen);
  cout<<"[client]"<< "ending-message sent to server" << endl;
```

```
    clock_t start = clock();
    while (1 == 1)
    {
        u_long mode = 1;
        ioctlsocket(socketClient, FIONBIO, &mode);
        while (recvfrom(socketClient, Buffer, maxBuffSize, 0,
  (sockaddr*)&servAddr, &servAddrlen) <= 0)
        {
            if (clock() - start > timeLimit)
            {
                cout<<"[client]"<< "Time Out,start ending-message resending "
  << " ";
                char* Buffer = new char[sizeof(header)];
                header.flag = PacketFlags::OVER;
                header.checkSum = 0;
                u_short temp = getCheckSum((u_short*)&header, sizeof(header));
                header.checkSum = temp;
                memcpy(Buffer, &header, sizeof(header));
                sendto(socketClient, Buffer, sizeof(header), 0,
  (sockaddr*)&servAddr, servAddrlen);
                cout<<"[client]"<< "resend ended" << endl;
                start = clock();
            }
        }



        //缓冲区读取
        memcpy(&header, Buffer, sizeof(header));
        u_short check = getCheckSum((u_short*)&header, sizeof(header));
        if (header.flag == OVER)
        {
            cout<<"[client]"<< "server received ending-message" << endl;
            break;
        }
        else
            continue;

    }
    u_long mode = 0;
    ioctlsocket(socketClient, FIONBIO, &mode);}
```

服务器接收数据

```
    int ListenRecv(SOCKET& serverSock, SOCKADDR_IN& clientAddr, int&
clientAddrLen, char* message)
{
    long int packLen = 0;//文件长度
    Header header;
    char* Buffer = new char[maxBuffSize + sizeof(header)];
    int seq = 0;
```

```
    while (1 == 1)
    {
        int length = recvfrom(serverSock, Buffer, sizeof(header) + maxBuffSize, 0,
(sockaddr*)&clientAddr, &clientAddrLen);//接收报文长度

        memcpy(&header, Buffer, sizeof(header));
        //判断是否结束
        if (header.flag == OVER && getCheckSum((u_short*)&header, sizeof(header))
== 0)
        {
            cout << "Data package received." << endl;
            break;
        }
        if (header.flag == INIT && getCheckSum((u_short*)Buffer, length -
sizeof(header)))
        {
            //是否接受的不是指定的包
            if (seq != int(header.seqNum))
            {
                sendACK(header, Buffer, serverSock, clientAddr, clientAddrLen,
seq);
                continue;//丢弃
            }
            seq = int(header.seqNum);
            if (seq > 255)
            {
                seq = seq - 256;
            }
            //取buffer存的的数据
            cout << "Send message " << endl;
            cout<<"      information: length(Bytes):" <<length - sizeof(header) <<
"   seqNum :" << int(header.seqNum) << "   Flag: " << int(header.flag) << endl;
            char* temp = new char[length - sizeof(header)];
            memcpy(temp, Buffer + sizeof(header), length - sizeof(header));

            memcpy(message + packLen, temp, length - sizeof(header));
            packLen = packLen + int(header.dataLen);


            sendACK(header, Buffer, serverSock, clientAddr, clientAddrLen, seq);
            seq++;
            if (seq > 255)
            {
                seq = seq - 256;
            }
        }
    }

    sendOver(header, Buffer, serverSock, clientAddr, clientAddrLen);
    return packLen;
}
```

四次挥手

客户端

- 挥手实现

```
  int clientWave(Header header ,char* Buffer, SOCKET& socketClient,
SOCKADDR_IN& servAddr, int& servAddrlen,int seqOfWave) {
  if (seqOfWave == 1) {
      header.flag = PacketFlags::FIN;
  }
  else {
      header.flag = PacketFlags::FIN_ACK;
  }

  header.checkSum = 0;
  u_short temp = getCheckSum((u_short*)&header, sizeof(header));
  header.checkSum = temp;
  memcpy(Buffer, &header, sizeof(header));

  if (sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr,
servAddrlen) == -1)
  {
      return -1;
  }
  return 0;}
```

- 总函数

```
  int disconnect(SOCKET& socketClient, SOCKADDR_IN& servAddr, int&
servAddrlen)
  {
  Header header;
  char* Buffer = new char[sizeof(header)];
  u_short checkSum;
  clientWave(header, Buffer, socketClient, servAddr, servAddrlen,1);
  clock_t start = clock();
  u_long mode = 1;
  ioctlsocket(socketClient, FIONBIO, &mode);
  //接收第二次挥手
  while (recvfrom(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, &servAddrlen) <= 0)
  {
      if (clock() - start > timeLimit)//超时重新传输
      {

          clientWave(header, Buffer, socketClient, servAddr, servAddrlen,1);
          start = clock();
          cout << "second wave from server time out, resending first wave" <<
```

```
endl;
        }
    }
    //校验和检验
    memcpy(&header, Buffer, sizeof(header));
    if (header.flag == ACK && getCheckSum((u_short*)&header, sizeof(header) ==
0))
    {
        cout << "second wave received from server" << endl;
    }
    else
    {
        cout << "connection error, quiting now" << endl;
        return -1;
    }
    //第三次挥手
    clientWave(header, Buffer, socketClient, servAddr, servAddrlen,3);
    start = clock();
    //接收第四次挥手
    while (recvfrom(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, &servAddrlen) <= 0)
    {
        if (clock() - start > timeLimit)//超时重新传输第三次挥手
        {
            cout << "fourth wave from server time out.resending third wave" <<
endl;
            clientWave(header, Buffer, socketClient, servAddr, servAddrlen,3);
            start = clock();
            cout << "" << endl;
        }
    }
    cout << "fourth wave from server received,disconnection completed." <<
endl;
    return 1;}
```

服务端

- 挥手实现

```
    int serverWave(Header header, char* Buffer, SOCKET& serverSock,
SOCKADDR_IN& clientAddr, int& clientAddrLen, int seqOfWave) {
    if (seqOfWave == 2)
    {
        header.flag = ACK;
    }
    else {
        header.flag = FIN_ACK;
    }

    header.checkSum = 0;
    u_short temp = getCheckSum((u_short*)&header, sizeof(header));
```

```
  header.checkSum = temp;
  memcpy(Buffer, &header, sizeof(header));
  if (sendto(serverSock, Buffer, sizeof(header), 0, (sockaddr*)&clientAddr,
clientAddrLen) == -1)
  {
      return -1;
  } }
```

- 总函数

```
  int disconnect(SOCKET& serverSock, SOCKADDR_IN& clientAddr, int&
clientAddrLen)
  {
  Header header;
  char* Buffer = new char[sizeof(header)];
  while (1 == 1)
  {
      int length = recvfrom(serverSock, Buffer, sizeof(header) +
maxBuffSize, 0, (sockaddr*)&clientAddr, &clientAddrLen);//接收报文长度
      memcpy(&header, Buffer, sizeof(header));
      if (header.flag == FIN && getCheckSum((u_short*)&header,
sizeof(header)) == 0)
      {
          cout << "receieved first wave from client" << endl;
          break;
      }
  }
  //发送第二次挥手
  serverWave(header, Buffer, serverSock, clientAddr, clientAddrLen, 2);
  clock_t start = clock();

  //接收第三次挥手
  while (recvfrom(serverSock, Buffer, sizeof(header), 0,
(sockaddr*)&clientAddr, &clientAddrLen) <= 0)
  {
      if (clock() - start > timeLimit)
      {
          cout << "third wave from client TIME OUT, resending second wave
now" << endl;
          serverWave(header, Buffer, serverSock, clientAddr, clientAddrLen,
2);

      }
  }

  Header temp1;
  memcpy(&temp1, Buffer, sizeof(header));
  if (temp1.flag == FIN_ACK && getCheckSum((u_short*)&temp1, sizeof(temp1)
== 0))
  {
      cout << "received third wave" << endl;
  }
```

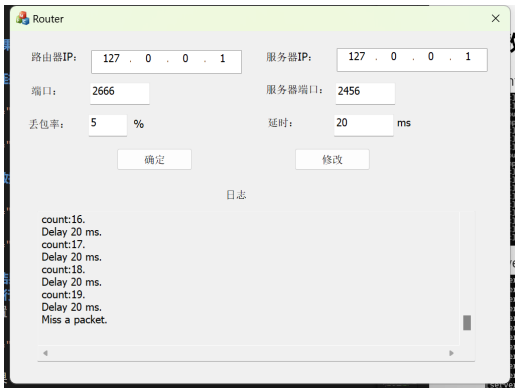
```
    else
    {
        cout << "error(third wave receive)" << endl;
        return -1;
    }

    //发送第四次挥手
    serverWave(header, Buffer, serverSock, clientAddr, clientAddrLen, 4);
    cout << "Completed:fourth wave,disconnected!" << endl;
    return 1; }
```

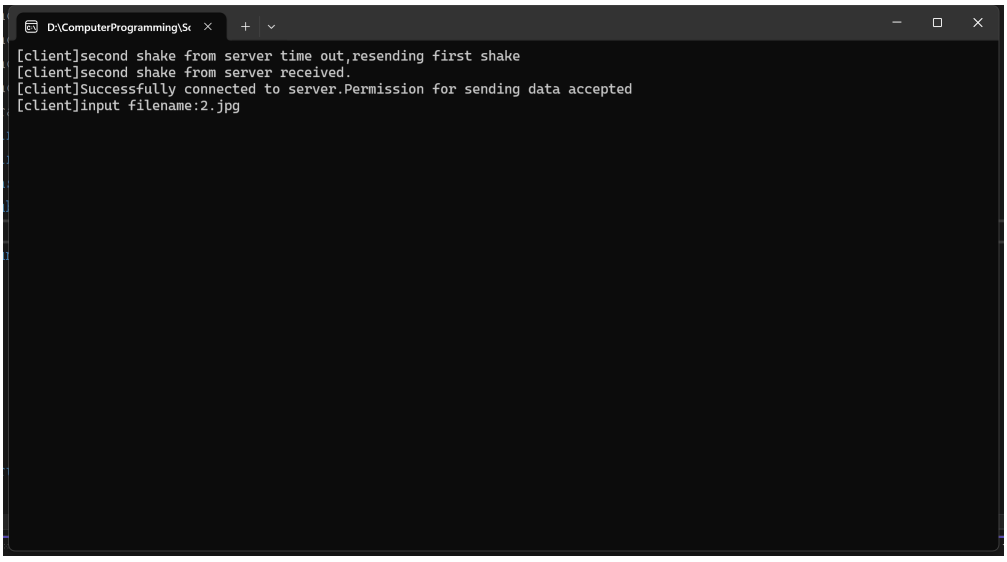
实验结果展示

建立连接(三次挥手)

- router设置



- client



丢包显示（丢包时客户端会显示timeout，然后重传）

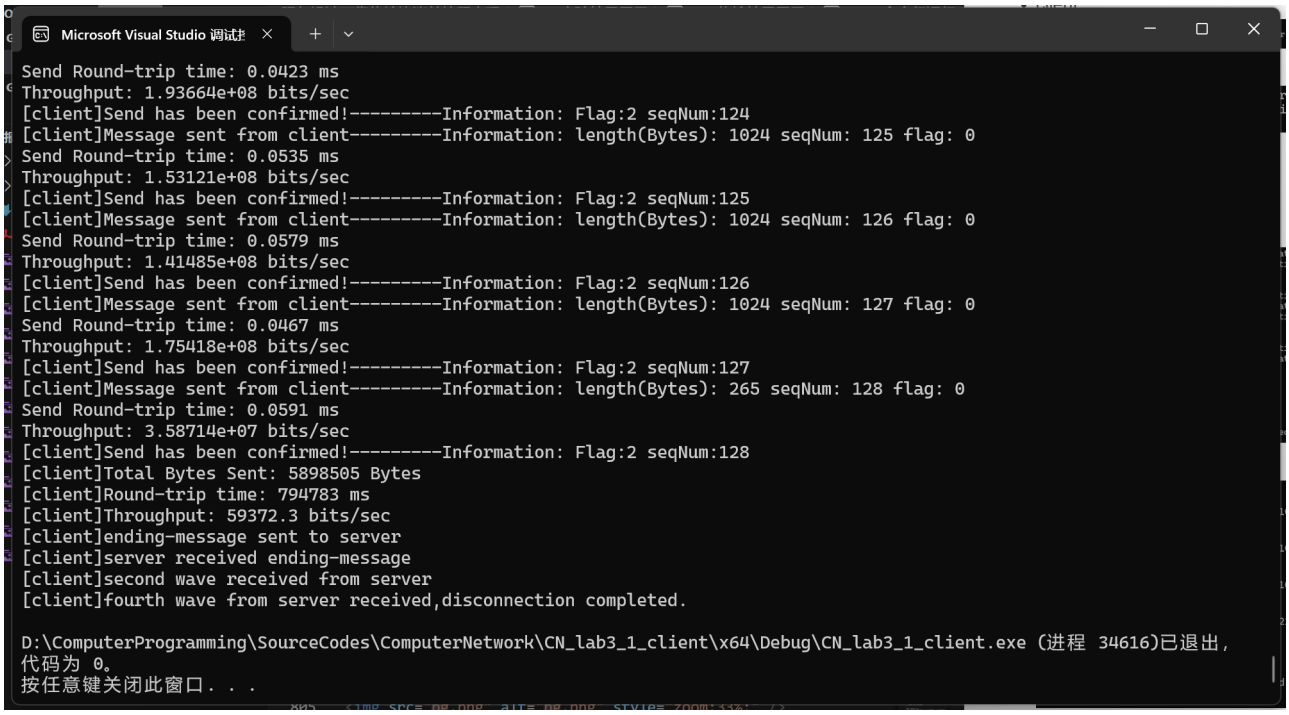
```
[client]Time Out, resending message 1 of 5
[client]Send has been confirmed!---------Information: Flag:2 seqNum:16
[client]Message sent from client---------Information: length(Bytes): 1024 seqNum: 17 flag: 0
Send Round-trip time: 0.0452 ms
Throughput: 1.81239e+08 bits/sec
```

- server

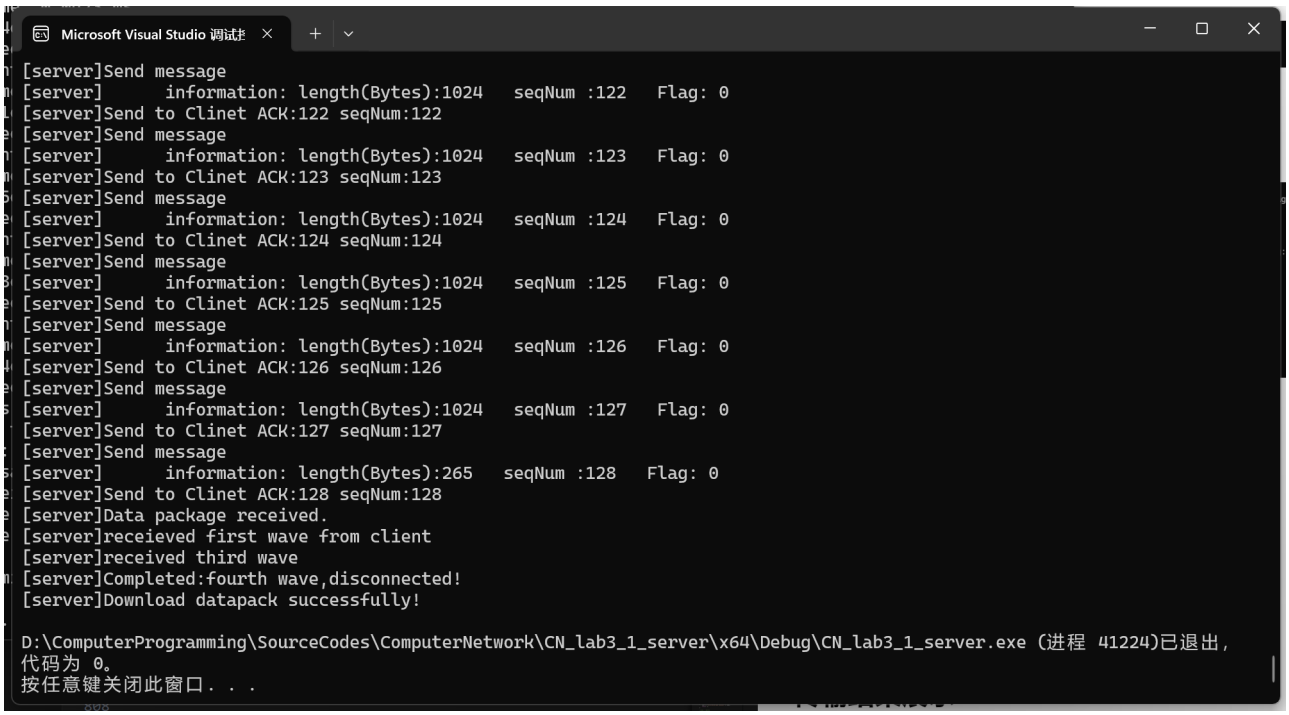
```
[server]Listening for client to start.....
[server]first wave from client received.
[server]Connection established!Data transmission is permitted.
```

传输数据,并断开连接

- client



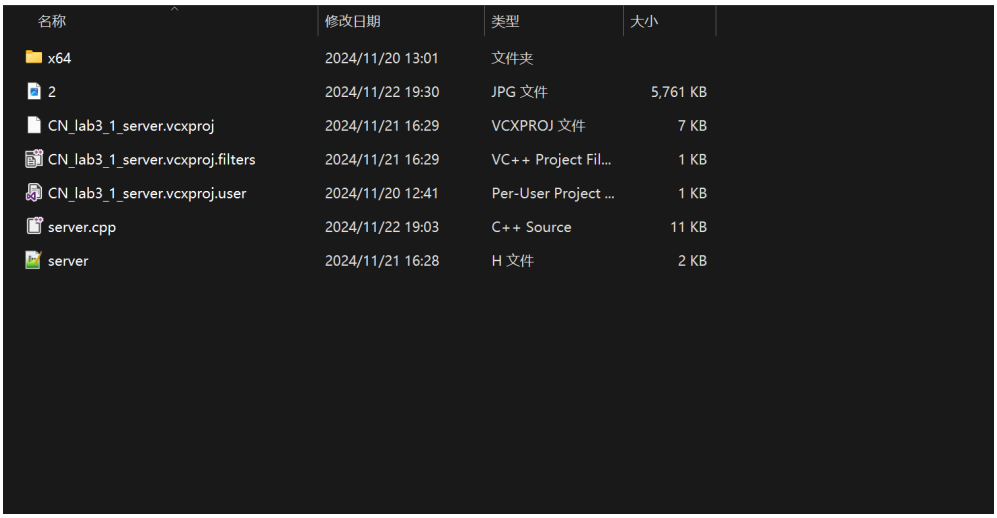
- server



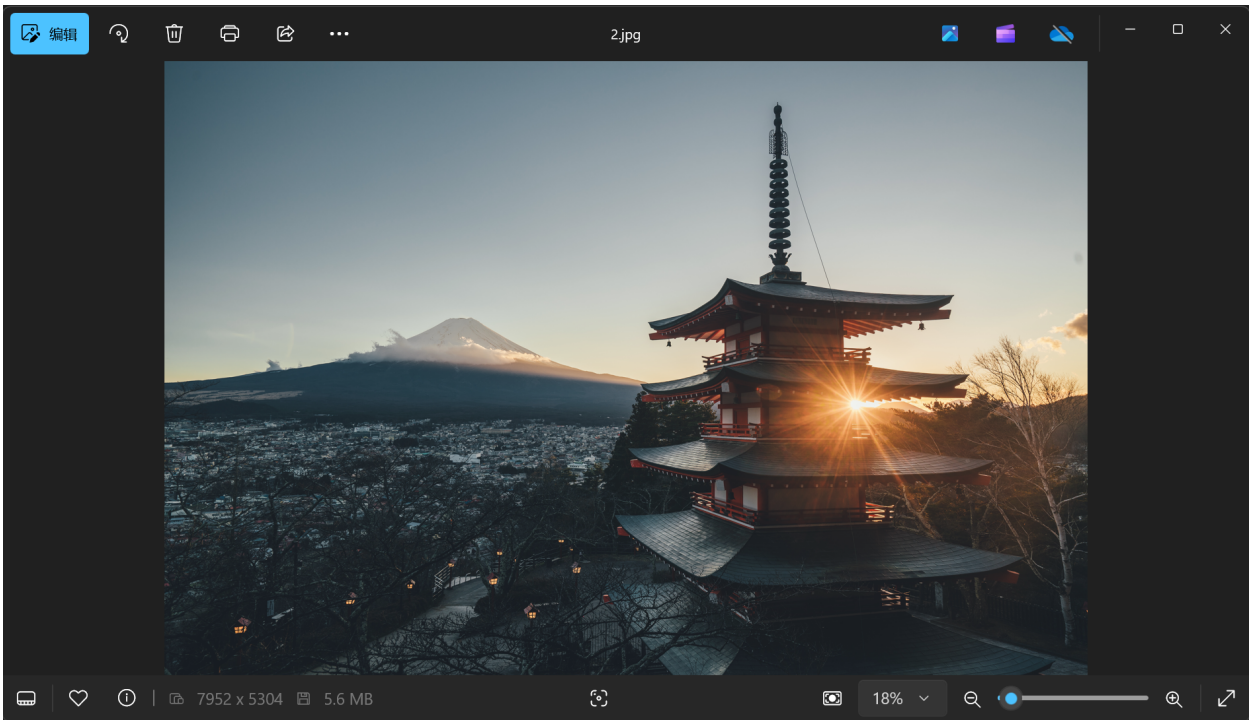
传输结果展示

命令行运行

- 传输位置



- 传输结果



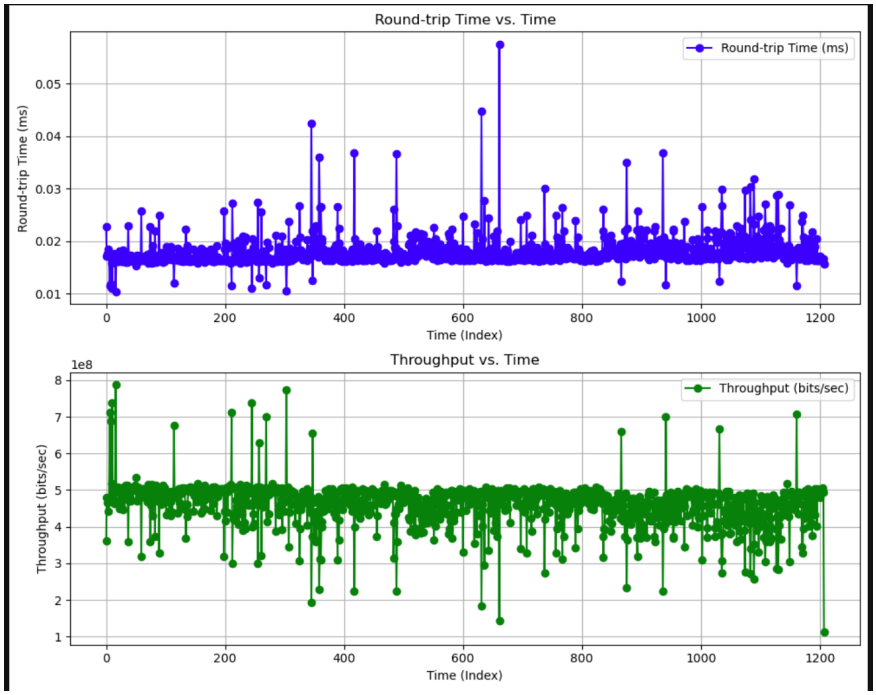
与原图完全相同，传输成功！

数据分析（图形结果）

我使用python对client端得出的RTT，吞吐率等数据进行清洗，并使用python绘制随时间变化情况，可以看到：

- RTT随时间稳定在0.018ms左右

- 吞吐率稳定在4.6-5.0e+08 bits/ms之间



总结

通过这次实验，学习了基于UDP协议的数据传输实现，深入理解了自定义报文头的设计，掌握了差错检测机制，如使用校验和验证数据完整性，以及通过标志位（如SYN、ACK、FIN）实现数据的确认和重传。实验中还加深了对连接管理的理解，模拟了握手和断开过程，更好地理解了面向无连接协议的通信流程。