计算机网络实验报告
实验3-1：基于UDP服务设计可靠传输协议并编程实现

姓名：谢雯菲 学号：2110803

作业要求

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、接受确认、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。

实验原理

流式套接字（TCP）和数据报套接字（UDP）的区别

TCP

TCP是面向连接的通信协议，通过三次握手建立连接，通讯完成时要拆除连接，只能用于端到端的通讯。

UDP

UDP通讯时不需要接受方确认，属于不可靠的传输，可能会出现丢包现象，实际应用中要求程序员编程验证。

流式套接字

流式套接字提供了面向连接的、可靠的、数据无措并且无重复的数据发送服务，并且发送数据时按顺序被接受。所有利用该套接字进行传输的数据均被视为连续的字节流且无长度限制。TCP协议使用该类接口。

数据报套接字

数据报套接字提供了面向无连接的服务。它独立地以数据包形式发送数据，不提供正确性检查，也不保证各数据包的发送顺序，因此可能出现数据的重发、丢失等现象，并且接受顺序由具体路由决定。UDP使用该类接口。

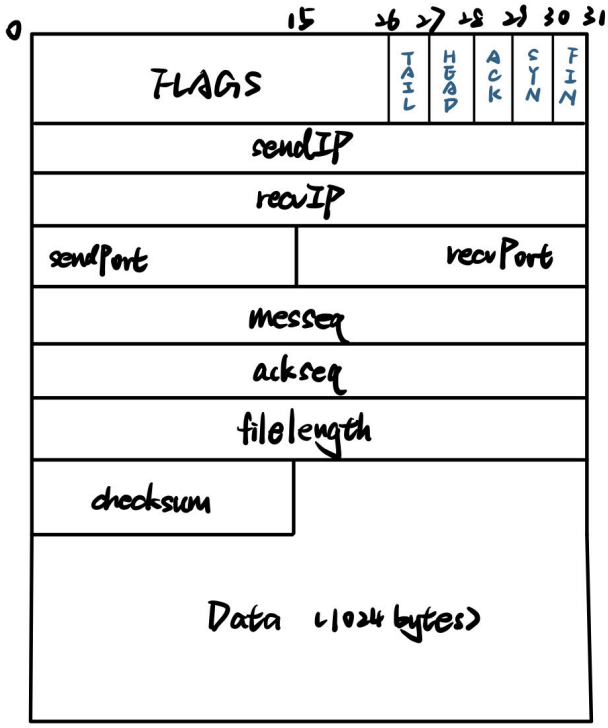
协议设计

- rdt1.0：假设下层通道是完全可靠的，发送端和接收端各只有一个状态。
- rdt2.0：考虑下层通道可能造成某些位的出错，需要增加差错检测、接收端反馈和发送端重传分组的功能。
- rdt2.1：在2.0的基础上增加分组序列，需要校验ACK/NAK分组是否损坏。
- rdt2.2：使用带序号的ACK代替NAK。
- rdt3.0：在之前校验和、序列号、ACK、重传机制的基础上实现超时重传。

本次实验在rdt3.0的基础上进行协议的设计。

数据包格式

设计的数据报头部如下所示：



- 32位FLAGS位（后五个比特位分别为：TAIL（最后一个文件标志位）、HEAD（头文件标志位）、ACK（确认位）、SYN（同步位）、FIN（结束位））
- 32位发送端IP地址（没用到）
- 32位接收端IP地址（没用到）
- 16位发送端端口号，16位接收端端口号（没用到）
- 32位发送序列号（实际只用了1位）
- 32位确认序列号（实际只用了1位）
- 32位文件长度
- 16位校验和

数据包结构的代码实现如下：

```
#define ACK 4
#define SYN 2
#define SYN_ACK 6
#define FIN 1
#define FIN_ACK 5
#define HEAD 8
#define TAIL 16
#define MES_SIZE 1054

//设计报文头部格式
//以下内容按1byte对齐，如果没有这条指令会以4Byte对齐
class message {
public:
#pragma pack(1)
    uint32_t FLAGS;     //标志位
    uint32_t sendIP;    //发送端IP
    uint32_t recvIP;    //接收端IP
    uint16_t sendPort;  //发送端口
```

```
    uint16_t recvPort;  //接收端口
    uint32_t messeq;    //消息序号
    uint32_t ackseq;    //确认序号
    uint32_t filelen;   //文件长度
    //在第一个数据包时表示整个文件的长度，后面表示单个数据包中数据的长度
    uint16_t checksum;  //校验和
    char data[1024];    //文件数据
#pragma pack()
    message() :FLAGS(0), sendIP(0), recvIP(0), sendPort(0), recvPort(0), messeq(0), ackseq(0), filelen(0), checksum(0) {
        memset(data, 0, 1024);
    }
    message(uint32_t FLAGS, uint32_t sendIP, uint32_t recvIP, u_short sendPort, u_short recvPort, uint32_t messeq, uint32_t ackseq, uint32_t filelen, uint32_t checksum);
    void setHEAD(int messeq, int fileSize, char* fileName);
    void fillData(int messeq, int size, char* data);    //填充数据
    void setchecksum(uint16_t* mes);
    void clearMes();
};
```

传输文件的第一包发送文件的名称和大小，不包含文件的实际内容。`setHEAD()`函数用来设置发送文件的第一个包的信息：

```
void message::setHEAD(int messeq, int fileSize, char* fileName) {
    //设置HEAD位为1
    this->FLAGS = HEAD; //FLAGS->00001000
    this->filelen = fileSize;
    this->messeq = messeq;
    memcpy(this->data, fileName, strlen(fileName) + 1);
}
```

`fillData()`函数用来填充`message`的具体数据：

```
void message::fillData(int messeq, int size, char* data) {
    //将文件数据填入数据包data变量
    this->messeq = messeq;
    this->filelen = size;
    memcpy(this->data, data, size);
}
```

差错检验

计算校验和、设置校验和位和检验文件包是否损坏的函数如下所示：

```
//计算校验和
uint16_t calChecksum(uint16_t* mes) {
    int size = MES_SIZE;    //得到数据的大小(单位为字节)
    int count = size / 2 - 1;   //每次循环计算2字节
    uint16_t* buf = (uint16_t*)malloc(size);    //用于遍历数据的缓冲区
    memset(buf, 0, size);
    memcpy(buf, mes, size); //复制
    uint32_t sum = 0;
    while (count--) {
```

```
        sum += *buf; //累加
        buf++;
        //如果sum的高16位不为0，则
        if (sum & 0xFFFF0000) {
            //存储高16位；
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}

//设置校验和
void message::setchecksum(uint16_t* mes) {
    this->checksum = calChecksum(mes); //按位取反，方便计算
}

//判断文件是否损坏
bool isCorrupt(message* mes) {
    if (calChecksum((uint16_t*)mes)==0) {
        return false;
    }
    return true;
}
```

建立连接

仿照TCP，三次握手建立连接。TCP三次握手过程如下图所示：

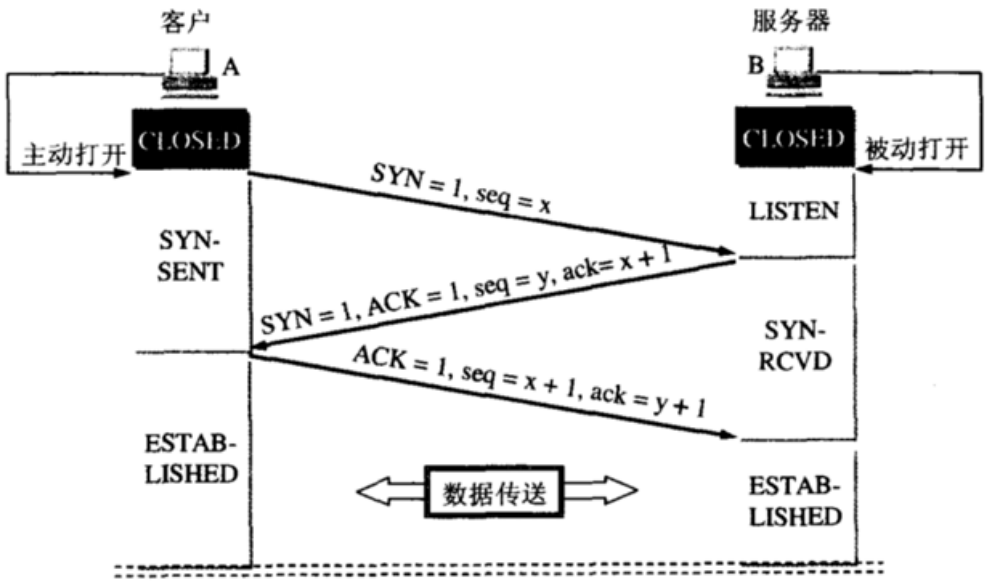


图 5-31　用三次握手建立 TCP 连接

客户端（发送端）

客户端有2个状态：

1. SYN-SENT状态：向服务器端发送SYN包，发送连接请求。
2. ESTAB-LISHED状态：在超时时间内接收服务器端的SYN_ACK包，并发回一个ACK包，建立连接。

编写代码时，用一个整型变量 `state` 表示客户端的状态，在建立连接前不断使用while循环判断状态。其中0对应SYN-SENT状态，1对应ESTAB-LISHED状态。

实现代码如下所示：

```
void connect() {
    int state = 0;
    bool flag = true;
    message* recvMes = new message();
    while (flag) {
        switch (state) {
        case 0: //向服务器端发送SYN位
            printTime();
            cout << "开始第一次握手！向服务器端发送SYN包！" << endl;
            sendFLAGS(SYN);
            state = 1;
            break;
        case 1: //在超时时间内收到SYN_ACK包，并发回ACK
            //判断是否超时
            setTO();
            select(clisock + 1, &readfds, NULL, NULL, &tv);
            if (FD_ISSET(clisock, &readfds)) {
                //如果在超时时间内收到数据
                if (recvfrom(clisock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(seraddr), &seraddr_len) > 0) {
                    printTime();
                    if (recvMes->FLAGS == SYN_ACK) {
                        cout << "客户端发来SYN_ACK包！发回ACK数据包！" << endl;
                        sendFLAGS(ACK);
                        cout << "与服务器端建立连接！" << endl;
                        cout << "----------------------" << endl;
                        flag = false; //完成连接
                    }
                    else {
                        cout << "服务器端发送非SYN_ACK包！" << endl;
                    }
                }
            }
            else { //如果发生超时
                printTime();
                cout << "超时未收到服务器端SYN_ACK！重发SYN包！" << endl;
                sendFLAGS(SYN);
            }
            break;
        }
    }
    delete recvMes;
}
```

服务器端（接收端）

服务器端有3个状态：

1. LISTEN状态：等待客户端发起请求。
2. SYN-RCVD状态：收到SYN后，等待确认状态。
3. ESTAB-LISHED状态：收到客户端的ACK包后，建立连接

编写代码时，用一个整型变量`state`表示服务器端的状态，在建立连接前不断使用while循环判断状态。其中0对应LISTEN状态，1对应SYN-RCVD状态。

实现代码如下所示：

```
//三次握手建立连接
void connect() {
    int state = 0; //标识目前握手的状态
    bool flag = true;  //表明是否完成握手
    int res = 0;
    message* recvMes = new message;
    u_long mode = 1;
    int err = 0;
    while (flag) {
        switch (state) {
        case 0: //等待客户端发送SYN数据包状态
            // 设置阻塞模式
            setTO();
            err=select(sersock + 1, &readfds, NULL, NULL, NULL); //timeout设置为NULL表示等待无限长时间
            if (FD_ISSET(sersock, &readfds)) { //如果收到数据
                printTime();
                res = recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(cliaddr), &cliaddr_len);
                if (res > 0) {  //接收到数据
                    if (recvMes->FLAGS == SYN) {
                        cout << "收到来自客户端的连接请求！第一次握手成功！" << endl;
                        //服务器端发送SYN_ACK包，第二次握手
                        sendFLAGS(SYN_ACK);
                        state = 1; //转状态1
                    }
                    else {
                        cout << "第一次握手数据包不匹配！" << endl;
                    }
                }
                else {
                    cout << "接收数据不正确：" << WSAGetLastError() << endl;
                }
            }
            else {
                printTime();
                cout << "接收数据报错误：" << WSAGetLastError() << endl;
            }
            break;
        case 1: //接收客户端的ACK=1数据包
            //select函数确定一个或多个套接字的状态，并在必要时等待执行同步I/O。
            //判断是否超时
            setTO();
            select(sersock + 1, &readfds, NULL, NULL, &tv);
            if (FD_ISSET(sersock, &readfds)) {
                //如果在超时时间内收到数据
                if (recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(cliaddr), &cliaddr_len) > 0) {
                    printTime();
```

```
                    if (recvMes->FLAGS == ACK) {
                        cout << "客户端发来ACK包！第三次握手成功！" << endl;
                        cout << "--------------------------------" << endl;
                        flag = false;
                    }
                    else {
                        cout << "客户端发送非ACK包！" << endl;
                    }
                }
            }
            else { //如果发生超时
                printTime();
                cout << "超时未收到客户端ACK！重发SYN_ACK包！" << endl;
                sendFLAGS(SYN_ACK);
            }
            break;
        }
    }
    delete recvMes;
}
```

断开连接

仿照TCP，四次挥手断开连接。四次挥手过程如下图所示：

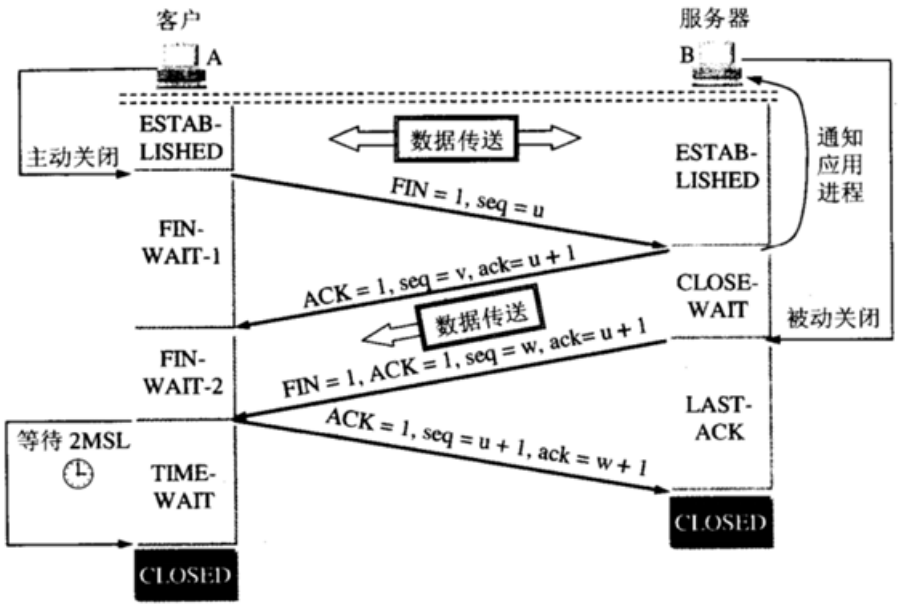


图 5-32　TCP 连接释放的过程

客户端（发送端）

客户端有5个状态：

1. ESTAB-LISHED状态：发起结束请求，向服务器端发送一个FIN包。
2. FIN-WAIT-1状态：等待服务器端的ACK包。
3. FIN-WSIT-2状态：等待服务器端的FIN_ACK包。
4. TIME-WAIT状态：发送一个ACK包，等待两秒后关闭连接。

编写代码时，用一个整型变量`state`表示客户端的状态，在断开连接前不断使用while循环判断状态。其中0对应ESTAB-LISHED状态，1对应FIN-WAIT-1状态，2对应FIN-WAIT-2状态，3对应TIME-WAIT状态。

代码实现如下：

```
//退出连接
void disconnect() {
    int state = 0;
    bool flag = true;

    message* recvMes = new message();

    while (flag) {
        switch (state) {
        case 0: //第一次挥手，发送FIN包
            printTime();
            cout << "结束连接，发送FIN包！" << endl;
            sendFLAGS(FIN);
            state = 1;
            break;
        case 1:
            //判断是否超时
            setTO();
            select(clisock + 1, &readfds, NULL, NULL, &tv);
            if (FD_ISSET(clisock, &readfds)) {
                //如果在超时时间内收到数据
                printTime();
                if (recvfrom(clisock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(seraddr), &seraddr_len) > 0) {
                    if (recvMes->FLAGS == ACK) {
                        cout << "服务器端发来ACK包！第二次挥手成功！" << endl;
                        state = 2;
                    }
                    else {
                        cout << "客户端发送非ACK包！" << endl;
                    }
                }
            }
            else { //如果发生超时
                printTime();
                cout << "超时未收到客户端ACK！重发FIN包！" << endl;
                sendFLAGS(FIN);
            }
            break;
        case 2:
            res = recvfrom(clisock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(seraddr), &seraddr_len);
            if (res >= 0) {
                if (recvMes->FLAGS == FIN_ACK) {
                    printTime();
                    cout << "收到了来自服务器端的FIN_ACK包，第三次挥手成功！" <<
endl;
                    state = 3;
                }
                else {
                    printTime();
                    cout << "服务器端发送非FIN_ACK包！" << endl;
                }
            }
            break;
```

```
        case 3: //向服务器端发送ACK包
            printTime();
            cout << "向服务器端发送ACK包，结束连接！" << endl;
            sendFLAGS(ACK);
            flag = false;
            break;
        }
    }
    delete recvMes;
}
```

服务器端（接收端）

服务器端有4个状态：

1. ESTAB-LISHED状态：接收客户端的FIN请求，并发回一个ACK包。
2. CLOSE-WAIT状态：进入关闭等待状态，通知应用进程关闭。
3. LAST-ACK状态：发送一个FIN_ACK包给客户端，等待客户端在超时时间内响应。
4. CLOSED状态：服务器端关闭。

编写代码时，用一个整型变量`state`表示服务器端的状态，在断开连接前不断使用while循环判断状态。其中0对应ESTAB-LISHED状态，1对应CLOSE-WAIT状态，2对应LAST-ACK状态。

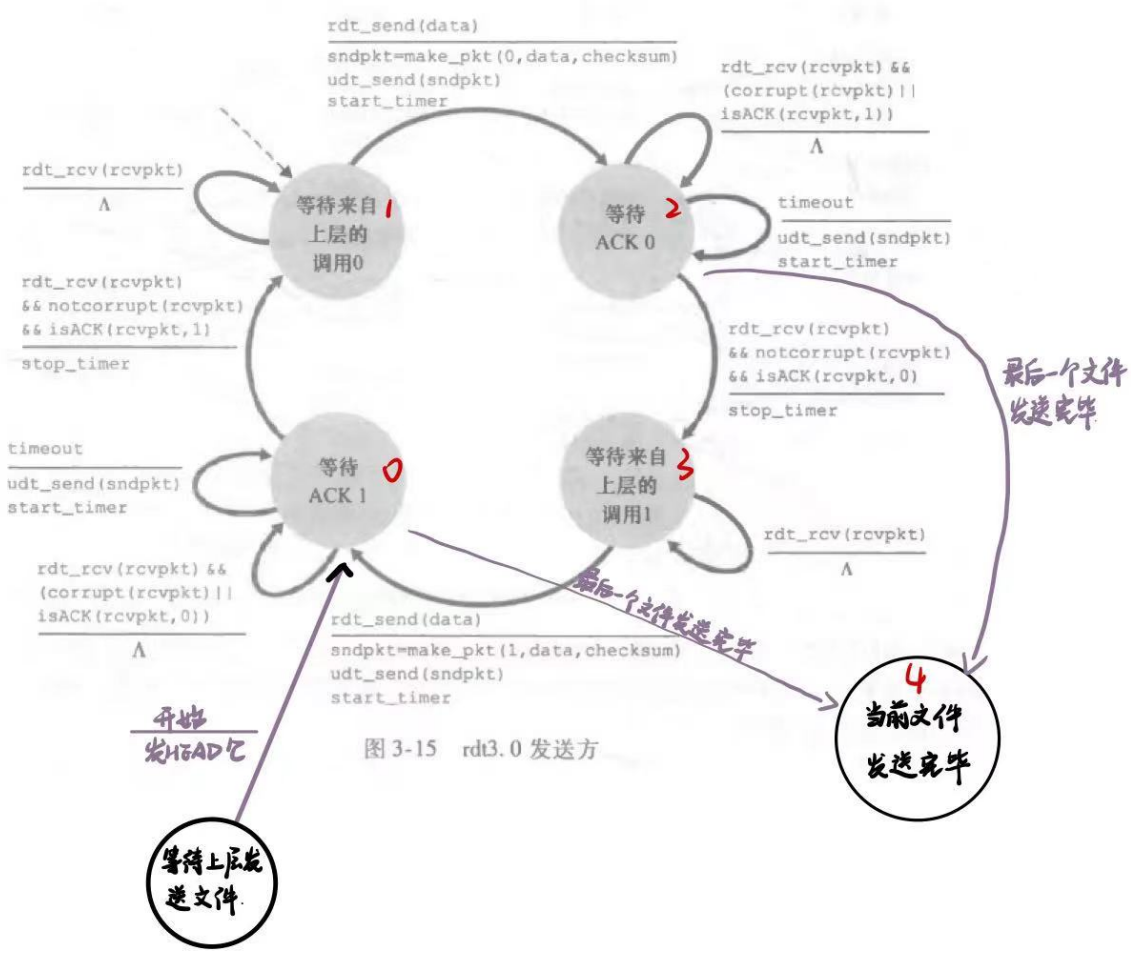
代码实现如下：

```
//断开连接
void disconnect() {
    int state = 0; //标识目前挥手状态
    bool flag = true; //为true时没有挥手结束
    int res = 0;
    message* recvMes = new message();

    while (flag) {
        switch (state) {
        case 0:
            printTime();
            cout << "客户端发起结束连接请求！" << endl;
            //向客户端发送ACK包
            sendFLAGS(ACK);
            state = 1;
            break;
        case 1:
            printTime();
            cout << "开始第三次挥手！发送FINACK包！" << endl;
            sendFLAGS(FIN_ACK);
            state = 2;
            break;
        case 2:
            //判断是否超时
            setTO();
            select(sersock + 1, &readfds, NULL, NULL, &tv);
            if (FD_ISSET(sersock, &readfds)) {
                //如果在超时时间内收到数据
                printTime();
                if (recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(cliaddr), &cliaddr_len) > 0) {
```

```
31                      if (recvMes->FLAGS == ACK) {
32                          cout << "客户端发来ACK包！第四次挥手成功！" << endl;
33                          cout << "--------------------------------" << endl;
34                          flag = false;
35                      }
36                      else {
37                          cout << "客户端发送非ACK包！" << endl;
38                      }
39                  }
40              }
41              else { //如果发生超时
42                  printTime();
43                  cout << "超时未收到客户端ACK！重发FIN_ACK包！" << endl;
44                  sendFLAGS(FIN_ACK);
45              }
46              break;
47          }
48      }
49      delete recvMes;
50  }
```

客户端发送文件

发送端的状态机如下图所示：



客户端每次发送一个文件都会重新调用`sendFile()`函数，在进入循环判断状态前，客户端先发送一个文件头包（发送序号为1）表示开始传输新的文件，然后进入循环判断状态（`state`）发送文件。状态数及操作如下：

0. 等待ACK1，如果超时或收到包错误就重传上一个包，收到后进入状态1

1. 上层调用，发送下一文件包，发送序号为1，收到后进入状态2
2. 等待ACK0，如果超时或收到包错误就重传上一个包，收到后进入状态3
3. 上层调用，发送下一文件包，发送序号为0，收到后进入状态0

进行以上循环直到发送至最后一个文件，在上述每个状态的操作后判断是否发送完毕，如果发送完毕进入状态4，表示当前文件发送完毕。

代码实现如下：

```
//发送文件
void sendFile() {
    int allsendnum = 0; //所有发送的包数
    int lostnum = 0;//丢包数
    //先发送一个记录文件名的数据包，并设置HEAD标志位为1，表示开始文件传输
    message* sendMes = new message;
    message* recvMes = new message;
    printTime();
    clock_t start = clock();
    cout << "发送文件头数据包....." << endl;
    char* fileNamec = new char[128];
    strcpy(fileNamec, fileName.c_str());
    sendMes->setHEAD(1, fileSize, fileNamec);
    sendMes->setchecksum((uint16_t*)sendMes);
    printSendMessage(sendMes);
    sendto(clisock, (char*)sendMes, BUFFER_SIZE, 0, (SOCKADDR*)&seraddr, seraddr_len);
    int sendnum = 0; //已发送文件包数量
    int state = 0;
    allsendnum++;
    bool flag = true;
    while (flag) {
        switch (state) {
        case 0:
            //判断是否超时
            setTO();
            select(clisock + 1, &readfds, NULL, NULL, &tv);
            if (FD_ISSET(clisock, &readfds)) {
                //如果在超时时间内收到数据
                printTime();
                if (recvfrom(clisock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(seraddr), &seraddr_len) > 0) {
                    if (isACK(recvMes, 1)&&!isCorrupt(recvMes)) {
                        cout << "收到ACK_1" << endl;
                        if (sendnum == packetNum) {
                            state = 4;
                        }
                        else {
                            state = 1;
                        }
                    }
                    else {
                        cout << "接收的包已损坏或不是ACK_1包！" << endl;
                    }
                }
                else {
```

```
                    cout << "接收ACK包错误：" << WSAGetLastError() << endl;
                }
            }
            else { //如果发生超时
                printTime();
                cout << "超时未收到ACK！重发数据包！" << endl;
                allsendnum++;
                lostnum++;
                sendto(clisock, (char*)sendMes, BUFFER_SIZE, 0,
(SOCKADDR*)&seraddr, seraddr_len);
            }
            break;
        case 1:
            //发送下一数据包
            printTime();
            cout << "正在发送第" << sendnum << "个数据包！" << endl;
            sendMes->clearMes();
            //如果是最后一个包
            if (sendnum == packetNum-1) {
                sendMes->FLAGS=TAIL;
            }
            sendMes->messeq = 0;
            sendMes->fillData(0, (fileSize - sendnum * 1024) > 1024 ? 1024
: fileSize - sendnum * 1024, fileBuffer + sendnum * 1024);
            sendMes->setchecksum((uint16_t*)sendMes);
            cout << "test校验和为：" << sendMes->checksum << endl << endl;
            res = sendto(clisock, (char*)sendMes, BUFFER_SIZE, 0,
(SOCKADDR*)&seraddr, seraddr_len);
            if (res > 0) {
                allsendnum++;
                cout << "数据包发送成功！" << endl;
                printSendMessage(sendMes);
                sendnum++;
                state = 2;
            }
            else {
                cout << "数据包发送失败！" << endl;
            }
            break;
        case 2:
            //判断是否超时
            setTO();
            select(clisock + 1, &readfds, NULL, NULL, &tv);
            if (FD_ISSET(clisock, &readfds)) {
                //如果在超时时间内收到数据
                printTime();
                if (recvfrom(clisock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(seraddr), &seraddr_len) > 0) {
                    if (isACK(recvMes, 0)&&!isCorrupt(recvMes)) {
                        cout << "收到ACK_0" << endl;
                        if (sendnum == packetNum) {
                            state = 4;
                        }
                        else {
                            state = 3;
```

```
                        }
                    }
                    else {
                        cout << "接收的包已损坏或不是ACK_0包！" << endl;
                    }
                }
                else {
                    cout << "接收ACK包错误：" << WSAGetLastError() << endl;
                }
            }
            else { //如果发生超时
                printTime();
                lostnum++;
                allsendnum++;
                cout << "超时未收到ACK！重发数据包！" << endl;
                sendto(clisock, (char*)sendMes, BUFFER_SIZE, 0,
(SOCKADDR*)&seraddr, seraddr_len);
            }
            break;
        case 3:
            //发送下一数据包
            printTime();
            cout << "正在发送第" << sendnum << "个数据包！" << endl;
            sendMes->clearMes();
            //如果是最后一个包
            if (sendnum == packetNum - 1) {
                sendMes->FLAGS = TAIL;
            }
            sendMes->fillData(1, (fileSize - sendnum * 1024) > 1024 ? 1024
: fileSize - sendnum * 1024, fileBuffer + sendnum * 1024);
            sendMes->setchecksum((uint16_t*)sendMes);
            cout << "test校验和为：" << sendMes->checksum << endl << endl;
            res = sendto(clisock, (char*)sendMes, BUFFER_SIZE, 0,
(SOCKADDR*)&seraddr, seraddr_len);
            if (res > 0) {
                cout << "数据包发送成功！" << endl;
                allsendnum++;
                printSendMessage(sendMes);
                sendnum++;
                state = 0;
            }
            else {
                cout << "数据包发送失败！" << endl;
            }
            break;
        case 4: //当前文件已经发完了
            flag = false;
            printTime();
            cout << "当前文件发送完毕。" << endl;
            clock_t end = clock();
            cout << "总传输时间为：" << (end - start)/CLOCKS_PER_SEC << "秒"
<< endl;
            cout << "吞吐率为：" << (float)fileSize / ((end - start)*1000 /
CLOCKS_PER_SEC) << "比特/毫秒" << endl;
```

```
145             cout << "错误（丢包+超时）率为：" << (float)lostnum / allsendnum
    << endl;
146             break;
147         }
148     }
149     delete sendMes;
150     delete recvMes;
151     delete []fileNamec;
152 }
```

超时重传

本次实验中使用`select()`函数判断接收ACK时是否超时。

`select()`函数返回已就绪并包含在`fd_set`结构中的套接字句柄总数；如果时间限制过期，则返回零；如果发生错误，则返回SOCKET_ERROR。

每次调用`select()`参数前设置参数调用的`setTO()`函数如下所示：

```
1  //设置超时时间
2  struct timeval tv;  //用于设置select函数的超时时间
3  fd_set readfds;     //文件描述符集合，用于监视文件描述符的状态
4
5  void setTO() {
6      FD_ZERO(&readfds);  //初始化设置为空
7      FD_SET(clisock, &readfds);  //设置套接字
8      tv.tv_sec = 0;  //设置超时时间为1s
9      tv.tv_usec = waittime;
10 }
```

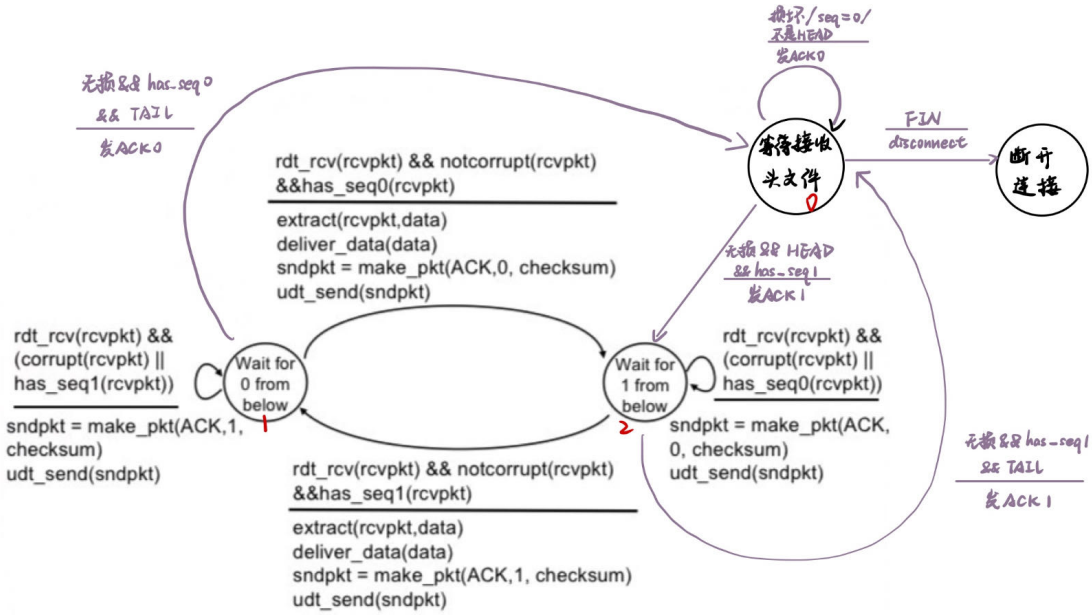
其中`waittime`参数将在每次程序运行时设定，设定过程如下：

```
1 cout << "请设置最大超时等待时间（毫秒）：" << endl;
2 cin >> waittime;
```

判断是否超时的具体实现在上述客户端发送文件的`sendFile()`函数中体现。

服务器端接收文件

接收端的状态机如下图所示：



服务器端完成连接后调用`receiveFile()`函数，进入循环使用`state`判断服务器端状态。状态数及操作如下：

0. 等待接收头文件或接收结束连接指令，如果不是头文件或序号错误，发送ACK0；如果收到头文件，则进入状态1循环；如果收到FIN请求，调用`disconnect()`函数断开连接。
1. 等待0序号文件，如果包已损坏或序号错误发ACK1；若收到正确的包发送ACK0，进入状态2；如果收到的包是当前文件的最后一个包，保存当前文件，返回状态0
2. 等待1序号文件，如果包已损坏或序号错误发ACK0；若收到正确的包发送ACK1，进入状态1；如果收到的包是当前文件的最后一个包，保存当前文件，返回状态0

在收到FIN请求前，上述循环将一直运行。代码实现如下：

```
//接收文件
void receiveFile() {
    int state = 0;  //状态
    bool flag = true; //为true时保持连接
    int res = 0;    //接收返回
    int recvnum = 0;

    message* recvMes = new message();

    while (flag) {  //循环接收文件数据包
        switch (state) {
        case 0: //等待数据包头文件状态
            res = recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0,
(SOCKADDR*)&(cliaddr), &cliaddr_len);
            if (res > 0) {
                if (randomNum(randomGen) <= LOSS_RATE) {//主动丢包
                    printTime();
                    cout << "主动丢包" << endl;
                }
                else {
                    if (randomNum(randomGen) <= DELAY_RATE) {//延时
                        int time = DELAY_BASE * randomNum(randomGen);
                        Sleep(time);
```

```
                        printTime();
                        cout << "延时" << time << "毫秒" << endl;
                    }
                    printTime();
                    printRecvMessage(recvMes);
                    if (recvMes->FLAGS == FIN) {
                        flag = false;
                        disconnect();
                    }
                    else if (isCorrupt(recvMes) || has_seq0(recvMes)) {
                        cout << "数据包接收失败，请求重传" << endl;
                        //发送ACKseq=0的包
                        sendACK(0);
                    }
                    else if (!isCorrupt(recvMes) && has_seq1(recvMes)) {
                        if (recvMes->FLAGS == HEAD) {
                            recvnum = 0;
                            cout << "接收数据包头成功！" << endl;
                            //初始化接收文件缓冲区
                            recvSize = 0;
                            fileSize = recvMes->filelen;
                            fileBuffer = new char[fileSize];
                            fileName = new char[128];
                            memcpy(fileName, recvMes->data, strlen(recvMes->data) + 1);
                            //计算文件包数量
                            packetnum = fileSize % 1024 ? fileSize / 1024 + 1 : fileSize / 1024;
                            cout << "开始接收来自发送端的文件，文件名为：" << fileName << endl;
                            cout << "文件大小为：" << fileSize << "比特，总共需要接收" << packetnum << "个数据包" << endl;
                            cout << "等待发送文件数据包……" << endl;
                            //发送ACK_1数据包
                            sendACK(1);
                            state = 1;
                        }
                        else {
                            cout << "收到的数据包不是文件头，等待发送端重传……" << endl;
                        }
                    }
                }
            }
            break;
        case 1:
            res = recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(cliaddr), &cliaddr_len);
            if (res > 0) {
                if (randomNum(randomGen) <= LOSS_RATE) {//主动丢包
                    printTime();
                    cout << "主动丢包" << endl;
                }
                else {
                    if (randomNum(randomGen) <= DELAY_RATE) {//延时
```

```
                        int time = DELAY_BASE * randomNum(randomGen);
                        Sleep(time);
                        printTime();
                        cout << "延时" << time << "毫秒" << endl;
                    }
                    printTime();
                    printRecvMessage(recvMes);
                    if (isCorrupt(recvMes) || has_seq1(recvMes)) {
                        //发送ACKseq=1的包，申请重发
                        cout << "数据包损坏或顺序有误，申请重发！" << endl;
                        sendACK(1);
                    }
                    else if (!isCorrupt(recvMes) && has_seq0(recvMes)) {
                        //收到了目标的包
                        recvnum++;
                        memcpy(fileBuffer + recvSize, recvMes->data, recvMes->filelen);
                        recvSize += recvMes->filelen;
                        sendACK(0);
                        if (recvMes->FLAGS == TAIL) {
                            //如果是最后一个包，进入状态0，也就是等待头文件或者断开连接
                            state = 0;
                            cout << "当前文件接收完毕！" << endl;
                            cout << "--------------------" << endl;
                            //保存文件
                            saveFile();
                        }
                        else {
                            cout << "第" << recvnum << "个文件包接收成功！" << endl;
                            state = 2;
                        }
                    }
                }
            }
            break;
        case 2:
            res = recvfrom(sersock, (char*)recvMes, BUFFER_SIZE, 0, (SOCKADDR*)&(cliaddr), &cliaddr_len);
            if (res > 0) {
                if (randomNum(randomGen) <= LOSS_RATE) {//主动丢包
                    printTime();
                    cout << "主动丢包" << endl;
                }
                else {
                    if (randomNum(randomGen) <= DELAY_RATE) {//延时
                        int time = DELAY_BASE * randomNum(randomGen);
                        Sleep(time);
                        printTime();
                        cout << "延时" << time << "毫秒" << endl;
                    }
                    printTime();
                    printRecvMessage(recvMes);
                    if (isCorrupt(recvMes) || has_seq0(recvMes)) {
```

```
                        //发送ACKseq=0的包，申请重发
                        cout << "数据包损坏或顺序有误，申请重发！" << endl;
                        sendACK(0);
                    }
                    else if (!isCorrupt(recvMes) && has_seq1(recvMes)) {
                        //收到了目标的包
                        recvnum++;
                        memcpy(fileBuffer + recvSize, recvMes->data,
recvMes->filelen);
                        recvSize += recvMes->filelen;
                        sendACK(1);
                        if (recvMes->FLAGS == TAIL) {
                            //如果是最后一个包，进入状态0，也就是等待头文件或者断开
连接
                            state = 0;
                            cout << "当前文件接收完毕！" << endl;
                            cout << "--------------------" << endl;
                            //保存文件
                            saveFile();
                        }
                        else {
                            cout << "第" << recvnum << "个文件包接收成功！" <<
endl;
                            state = 1;
                        }
                    }
                }
            }
            break;
        }
    }
    delete recvMes;
}
```

丢包及延时

本次实验中在服务器端（接收端）使用随机数设置了主动丢包和延时。在程序开始时设置了0~1的随机数，并在每次运行程序时设定丢包率和延时率，在上述状态1、2时判断随机数是否小于设定数，如果是，则采用跳过当前步骤主动丢包或使用`Sleep()`函数强制延时。

设置随机数代码如下：

```
default_random_engine randomGen;
uniform_real_distribution<float> randomNum(0.0, 1.0);  // 通过随机数设置丢包
```

设定丢包率及延时率过程如下：

```
cout << "请设置丢包率（0~1）（设为<0时不会发生丢包）：" << endl;
cin >> LOSS_RATE;
cout << "请设置延时率（0~1）（设为<0时不会发生延时）：" << endl;
cin >> DELAY_RATE;
cout << "请设置最大延时时间（毫秒）：" << endl;
cin >> DELAY_BASE;
```

具体实现过程在前面的服务器端`receiveFile()`函数中体现。

过程实现

创建套接字

在本次实验中，服务器端需要绑定，而客户端不需要。

对于服务器端而言，如果不进行绑定，操作系统会随机生成一个端口号给服务器端，如果有其他程序同时准备使用这个端口号，那么服务器就会无法启动。并且，如果服务器端不进行端口号的绑定，客户端无法知道服务器端的端口号，就算知道也需要每次重新获取。并且，服务器启动后一般不会停止，绑定服务器的端口号更有利于管理。

而对于客户端而言，服务器端不需要主动向客户端发出连接，客户端向服务器端发出连接请求的时候会将自己的IP地址和端口号一起发送过去，就算不进行绑定服务器端也可以准确找到客户端。并且，客户端不是一直运行的，每次由系统随机分配就可以。

日志输出

用于输出发送的数据包信息、接收的数据包信息、打印系统时间的函数如下所示：

```
//打印数据包消息
void printMessage(message* Mes) {
    cout << "数据包大小=" << Mes->filelen << "比特，标志位=" << Mes->FLAGS << endl;
    cout << "发送序号=" << Mes->messeq << "确认序号=" << Mes->ackseq << endl;
    cout << "校验和=" << Mes->checksum << endl;
}

//打印发送数据包消息
void printSendMessage(message* Mes) {
    cout << "【发送包信息】" << endl;
    printMessage(Mes);
}

//打印接收数据包消息
void printRecvMessage(message* Mes) {
    cout << "【接收包消息】" << endl;
    printMessage(Mes);
}

//打印系统时间
void printTime() {
    cout << endl;
    stringstream ss;    //声明一个stringstream对象ss，用于将数据转换为字符串
    SYSTEMTIME sysTime = { 0 }; //结构体SYSTEMTIME，用于保存系统时间的各个组成部分
    ss.clear();
    ss.str(""); //清空stringstream
    //获取当前系统时间，并存储在sysTime结构体中
    GetSystemTime(&sysTime);
    //将时间写入stringstream对象ss
    ss << "[" << sysTime.wYear << "/" << sysTime.wMonth << "/" << sysTime.wDay
        << " " << sysTime.wHour + 8 << ":" << sysTime.wMinute << ":"
        << sysTime.wSecond << ":" << sysTime.wMilliseconds << "]";
```

```
    cout << "时间: " << ss.str() << endl;
}
```

客户端（发送端）

读取文件

客户端以二进制形式读取文件的函数如下所示：

```
int fileSize; //文件大小
string fileName; //文件名字
string filePath; //文件路径
char* fileBuffer; //文件缓冲区
int packetNum; //总文件数

//读取文件
bool readFile() {
    //cout << "请输入要传输的文件路径（绝对路径使用/）: " << endl;
    //cin.ignore();
    //getline(cin, filePath);
    cout << "请输入要传输的文件名（包括后缀）: " << endl;
    cin.ignore();
    getline(cin, fileName);
    filePath = "C:/Users/10141/Documents/Fi/2023Fall/7 ComputerNetwork/labs/3/实验3测试文件和路由器程序/测试文件/" + fileName;
    // fileName = "test.txt";
    ifstream f(filePath, ifstream::in | ios::binary);   //以二进制方式打开
    if (!f.is_open()) {
        cout << "文件无法打开! " << endl;
        return false;
    }
    f.seekg(0, std::ios_base::end);//以文件流指针定位到流的末尾
    fileSize = f.tellg();
    packetNum = fileSize % 1024 ? fileSize / 1024 + 1 : fileSize / 1024;
    cout << "文件大小为: " << fileSize << "比特! 总共要发送" << packetNum << "个数据包" << endl;
    f.seekg(0, std::ios_base::beg); //将文件流指针重新定位到流的开始
    fileBuffer = new char[fileSize];
    if (fileBuffer == NULL) {
        cout << "内存分配失败! " << endl;
        f.close();
        return false;
    }
    f.read(fileBuffer, fileSize);
    f.close();
    return true;
}
```

main函数

客户端总实现如下：

```
int main() {
    //初始化socket环境
    WSADATA wsaData;
```

```
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cout << "socket初始化失败" << endl;
        return 0;
    }
    //创建socket
    clisock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    //ipv4地址，使用UDP数据报套接字，不指定协议
    if (clisock == INVALID_SOCKET) {
        cout << "socket创建失败" << endl;
        return -1;
    }
    //初始化地址
    //默认IP
    //定义套接字的IP和端口
    memset(&seraddr, 0, sizeof(sockaddr_in)); //清空addr内存部分的值
    seraddr.sin_family = AF_INET;
    //使用htons将u_short将主句转换为IP网络字节顺序（大端）
    seraddr.sin_port = htons(8000);
    //使用inet_addr函数将IPv4地址的字符串转换为s_addr结构
    seraddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    cout << "请设置最大超时等待时间（毫秒）: " << endl;
    cin >> waittime;

    cout << "客户端初始化完毕" << endl;

    connect();
    while (true) {
        cout << "是否要发送文件？是：0/退出：1" << endl;
        int order = -1;
        cin >> order;
        if (order == 1) {
            break;
        }
        else if (order == 0) {
            if (readFile()) {
                sendFile();
            }
        }
        else {
            cout << "指令错误！" << endl;
        }
    }
    //断开连接
    disconnect();

    res = closesocket(clisock);
    if (res == SOCKET_ERROR) {
        cout<< "关闭套接字错误：" << WSAGetLastError() << endl;
        return 1;
    }

    printTime();
    cout << "程序退出……" << endl;
    WSACleanup();
```

```
59
60      delete fileBuffer;
61
62      return 0;
63  }
```

服务器端（接收端）

保存文件

实现以二进制形式保存文件的函数如下所示：

```
1   int fileSize = 0; //总文件长度
2   char* fileName;
3   char* fileBuffer;
4   int packetnum;  //某个文件需要的数据包数量
5   unsigned int recvSize; //累计收到的文件位置（用于写入缓冲区）
6
7   //保存文件
8   void saveFile() {
9       string filePath = "C:/Users/10141/Documents/Fi/2023Fall/7
    ComputerNetwork/labs/3/3-1/save/";
10      for (int i = 0; fileName[i]; i++) {
11          filePath += fileName[i];
12      }
13      cout << "文件路径为：" << filePath << endl;
14      /*ofstream fout;
15      fout.open(filePath, ios::app);
16      for (int i = 0; i < fileSize; i++) {
17          fout << fileBuffer[i];
18      }*/
19      ofstream fout(filePath, ios::binary | ios::out);
20      fout.write(fileBuffer, fileSize); // 这里还是size,如果使用string.data或
    c_str的话图片不显示，经典深拷贝问题
21      printTime();
22      cout << "当前文件保存成功！" << endl;
23      fout.close();
24  }
```

main函数

服务器端总实现如下所示：

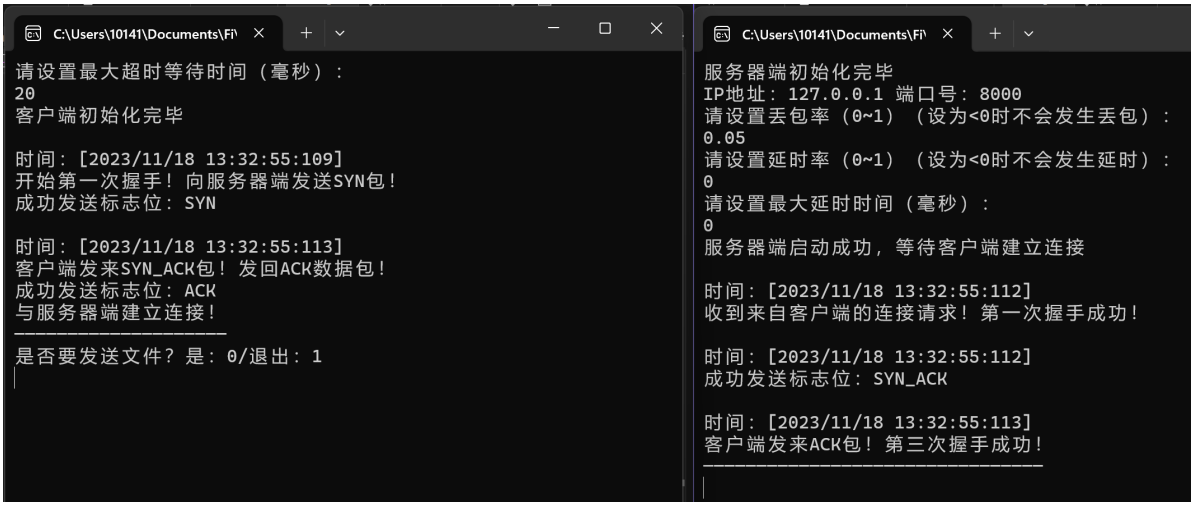
```
1   int main() {
2       //加载socket库
3       WSADATA wsaData;
4       if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
5           cout << "socket初始化失败" << endl;
6           return 0;
7       }
8       //创建socket
9       sersock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
10      //ipv4地址，使用UDP数据报套接字，不指定协议
11      if (sersock == INVALID_SOCKET) {
12          cout << "socket创建失败" << endl;
```

```
        return -1;
    }
    cout << "服务器端初始化完毕" << endl;

    //初始化地址
    //默认IP
    //定义套接字的IP和端口
    memset(&seraddr, 0, sizeof(sockaddr_in)); //清空addr内存部分的值
    seraddr.sin_family = AF_INET;
    //使用htons将u_short将主句转换为IP网络字节顺序（大端）
    string serIP = "127.0.0.1";
    int port = 8000;
    /*cout << "请输入IP地址：" << endl;
    cin >> serIP;
    cout << "请输入端口号：" << endl;
    cin >> port;*/
    char* portc = new char[20];
    memcpy(portc, serIP.c_str(),20);

    seraddr.sin_port = htons(port);
    //使用inet_addr函数将IPv4地址的字符串转换为s_addr结构
    seraddr.sin_addr.s_addr = inet_addr(portc);
    //将套接字绑定到本地IP和端口
    if (bind(sersock, (SOCKADDR*)&seraddr, sizeof(SOCKADDR))==-1) {
        //如果绑定发生错误
        cout << "套接字绑定发生错误：" << WSAGetLastError() << endl;
        return 0;
    }

    cout << "请设置丢包率（0~1）（设为<0时不会发生丢包）：" << endl;
    cin >> LOSS_RATE;
    cout << "请设置延时率（0~1）（设为<0时不会发生延时）：" << endl;
    cin >> DELAY_RATE;
    cout << "请设置最大延时时间（毫秒）：" << endl;
    cin >> DELAY_BASE;

    cout << "服务器端启动成功，等待客户端建立连接" << endl;

    connect();
    receiveFile();

    //关闭socket
    int res = closesocket(sersock);
    if (res == SOCKET_ERROR) {
        cout << "关闭套接字错误：" << WSAGetLastError() << endl;
        return 1;
    }
    printTime();
    cout << "程序退出……" << endl;
    WSACleanup();

    delete fileBuffer;
    delete fileName;
    delete []portc;
```

```
68      return 0;
69  }
```

运行结果

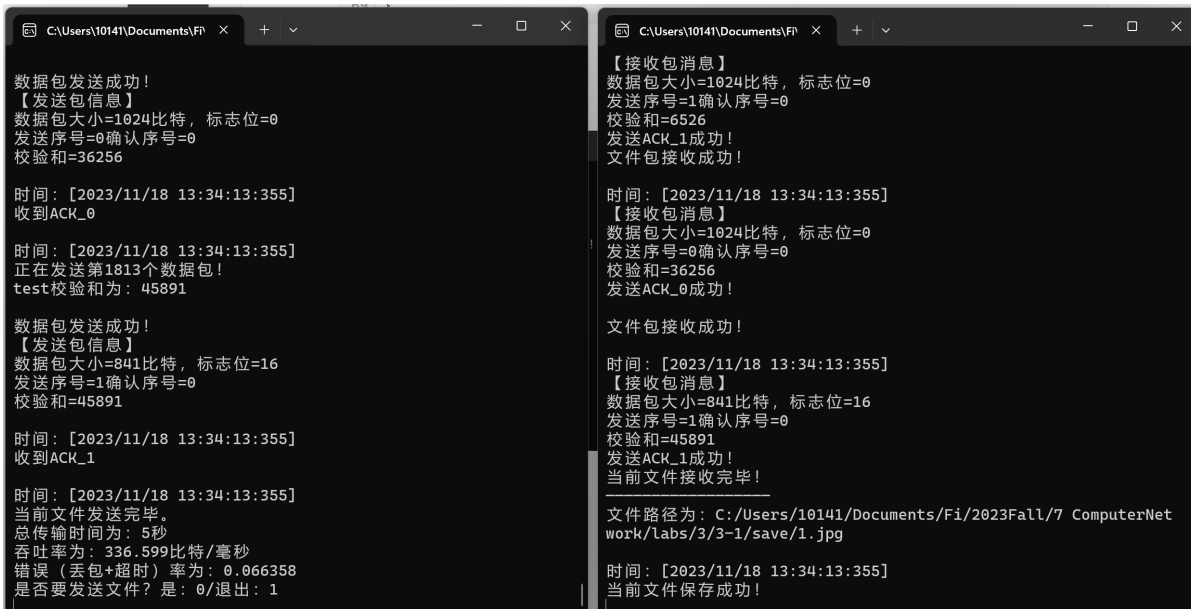
建立连接及设定丢包率



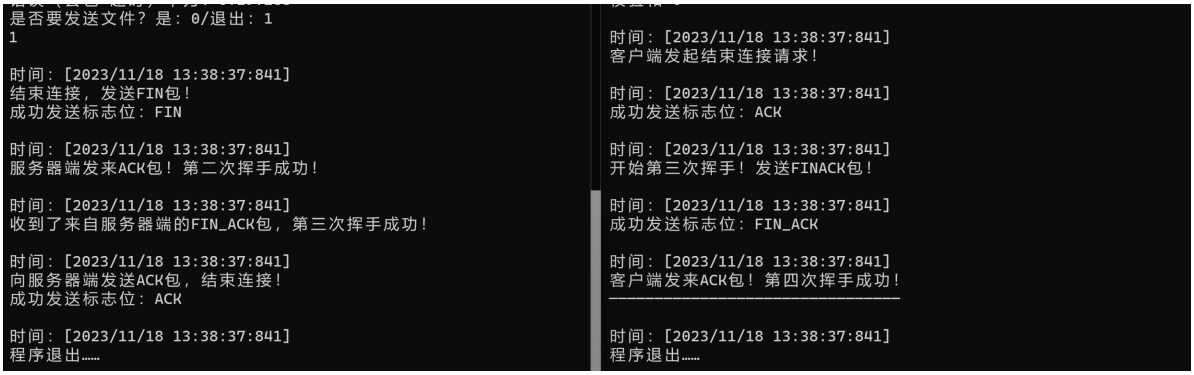
传输文件

丢包显示

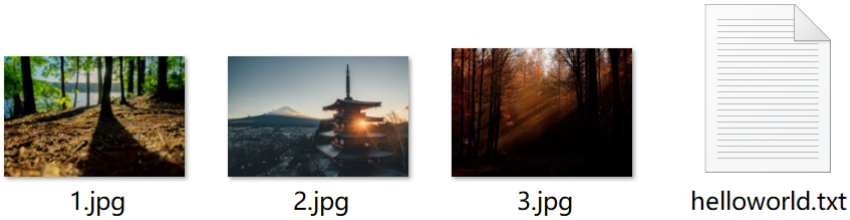
日志及吞吐率打印



退出连接



传输文件结果



实验总结

通过本次实验，基于rdt3.0设计了基于UDP服务的可靠传输协议，实现了差错检测、接收确认、超时重传等功能。为后续的实验奠定了基础。