实验三：基于 UDP 服务设计可靠传输协议并编程实现

- 姓名 : 马永田
- 年级： 2020 级
- 专业 : 计算机科学与技术
- 指导教师 : 张建忠 & 徐敬东

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。**

- 数据报套接字：UDP；
- 建立连接：实现类似TCP的握手、挥手功能；
- 差错检测：计算校验和；
- 确认重传：rdt2.0、rdt2.1、rdt2.2、rdt3.0等，亦可自行设计协议；
- 单向传输：发送端、接收端；
- 有必要日志输出。

协议设计

报文格式

协议中设计的报文格式如下代码所示：

```
#pragma pack(1) //按字节对齐
struct Packet   //报文格式
{
//SYN=0x01  ACk=0x02  FIN=0x04  PACKET=0x08  StartFile=0x10  EndFile=0x20  File=0x40
    DWORD SendIP;       //发送端IP
    DWORD RecvIP;       //接收端IP
    u_short Port;       //端口
    u_short Protocol;   //协议类型
    int Flags = 0;      //标志位
    int Seq;            //消息发送序号
    int Ack;            //确认序号
    int index;          //文件分片传输 第几片
    int length;         //Data段长度
    u_short checksum;   //校验和
    char Data[BUF_SIZE];//报文数据段
};
#pragma pack()//恢复4Byte编址格式
```

停等机制实现

停等协议是发送双方传输数据的一种协议方式，在停等协议下只有正确收到对方的ACK确认后才会继续发送新的数据包。 实验中设置套接字为非阻塞模式，在客户端中(与服务端的停等接收差别不大)停等机制使用如下函数实现：

```
//收发包
void sendPacket(Packet* sendP) {
    setCheckSum(sendP);//设置校验和
    if (sendto(sockClient, (char*)sendP, sizeof(Packet), 0, (struct sockaddr*)&sockAddr, sizeof(sockaddr)) != SOCKET_ERROR) {
    }
}
bool recvPacket(Packet* recvP) {
    memset(recvP, 0, sizeof(sizeof(&recvP)));
    int re = recvfrom(sockClient, (char*)recvP, sizeof(Packet), 0, (struct sockaddr*)&sockAddr, &addr_len);
    if (re != -1 && checkCheckSum(recvP)) {//检查校验和
        return true;
    }
    return false;
}
```

```
//停等发送函数
bool stopWaitSend(Packet* sendP, Packet* recvP) {
    sendPacket(sendP);
    if (checkFIN(sendP) && state != FIN_WAIT_1) {
        state = FIN_WAIT_1;
    }
    clockStart = clock();    //开始计时
    int retransNum = 0;      //重发次数
    while (1){
        if (recvPacket(recvP)) {
            if (checkACK(recvP) && recvP->Ack == (sendP->Seq + 1)){
                return 1;//收到对sendP的确认报文
            }
        }
        clockEnd = clock();
        if (retransNum == RETRANSMISSION_TIMES) {    //到达最大重发次数
            return 0;
        }
        if ((clockEnd - clockStart)>= WAIT_TIME){    //超时重发
            retransNum++;
            sendPacket(sendP);
            clockStart = clock();
        }
    }
    return 0;//发送失败
}
```

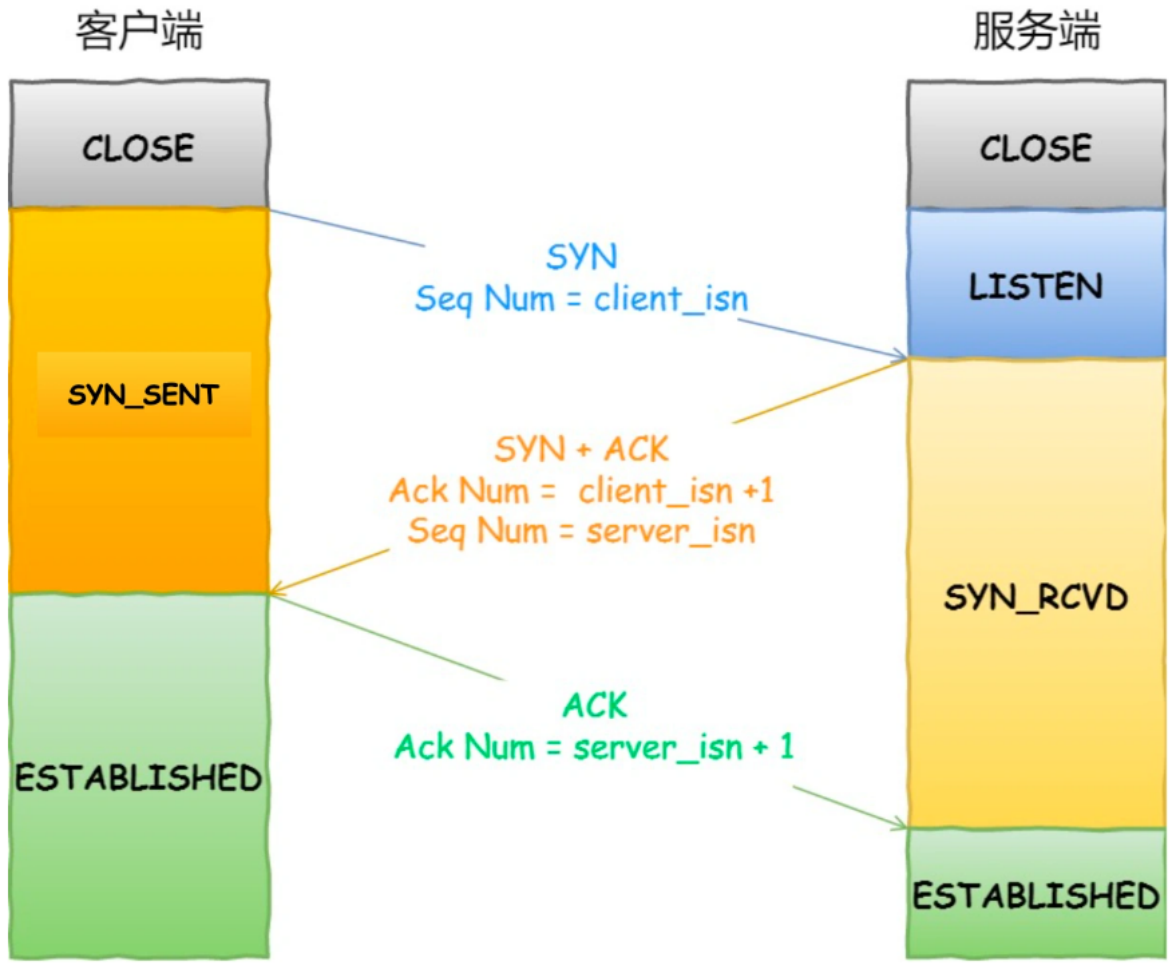
其工作流程如下：

1. 发送方发送前会为该数据帧设置消息序号Seq，并将当前数据帧暂时保留；
2. 发送方发送数据帧后，启动计时器；

3. 当接收方检测到一个含有差错的数据帧时，舍弃该帧；
4. 当接收方收到无差错的数据帧后，向发送方返回一个确认序号Ack=Seq+1的ACK确认帧；
5. 若发送方在规定时间内收到想要的ACK确认帧，开始下一帧的发送；
6. 若发送方在规定时间内未收到ACK确认帧，则重新发送暂留的数据帧；
7. 当重发次数超过规定次数后，发送失败放弃发送。

建立连接

建立连接机制参考了TCP中的三次握手过程，其流程如图所示：



服务器端：在初始化完成后服务端会处于**LISTEN**状态，不断监听数据帧的到来，当接收到数据帧后检查其类型，若收到**SYN同步**请求，则执行**buildConnection**函数，进入**SYN_RCVD**状态。在该函数中，服务端会向客户端回复一个**SYN请求同步与ACK确认**帧，其Ack值为收到的SYN包的Seq+1，而Seq为服务端发送序号sendSeq。当服务端收到该帧的确认ACK数据帧(Ack=sendSeq+1)时说明服务端与客户端的收发能力均无问题，**连接成功建立**，服务端进入**ESTABLISHED**状态。

```
//服务端
bool buildConnection(Packet synPacket) {
    state = SYN_RCVD;
    Packet sendACK, recvACK;
    setSYN(&sendACK);        //发送请求同步的SYN信息帧
    setACK(&sendACK, &synPacket);    //回复对于消息synPacket的ACK
    sendACK.Seq = sendSeq++;     //设置消息sendACK的发送序号.

    if(stopWaitSend(&sendSYN,&recvACK)){//为避免SYN攻击 使用停等发送 规定时间无响应则放弃
```

```
        state = ESTABLISHED;
        return 1;
    }
    state = CLOSE;
    return 0;
}
```

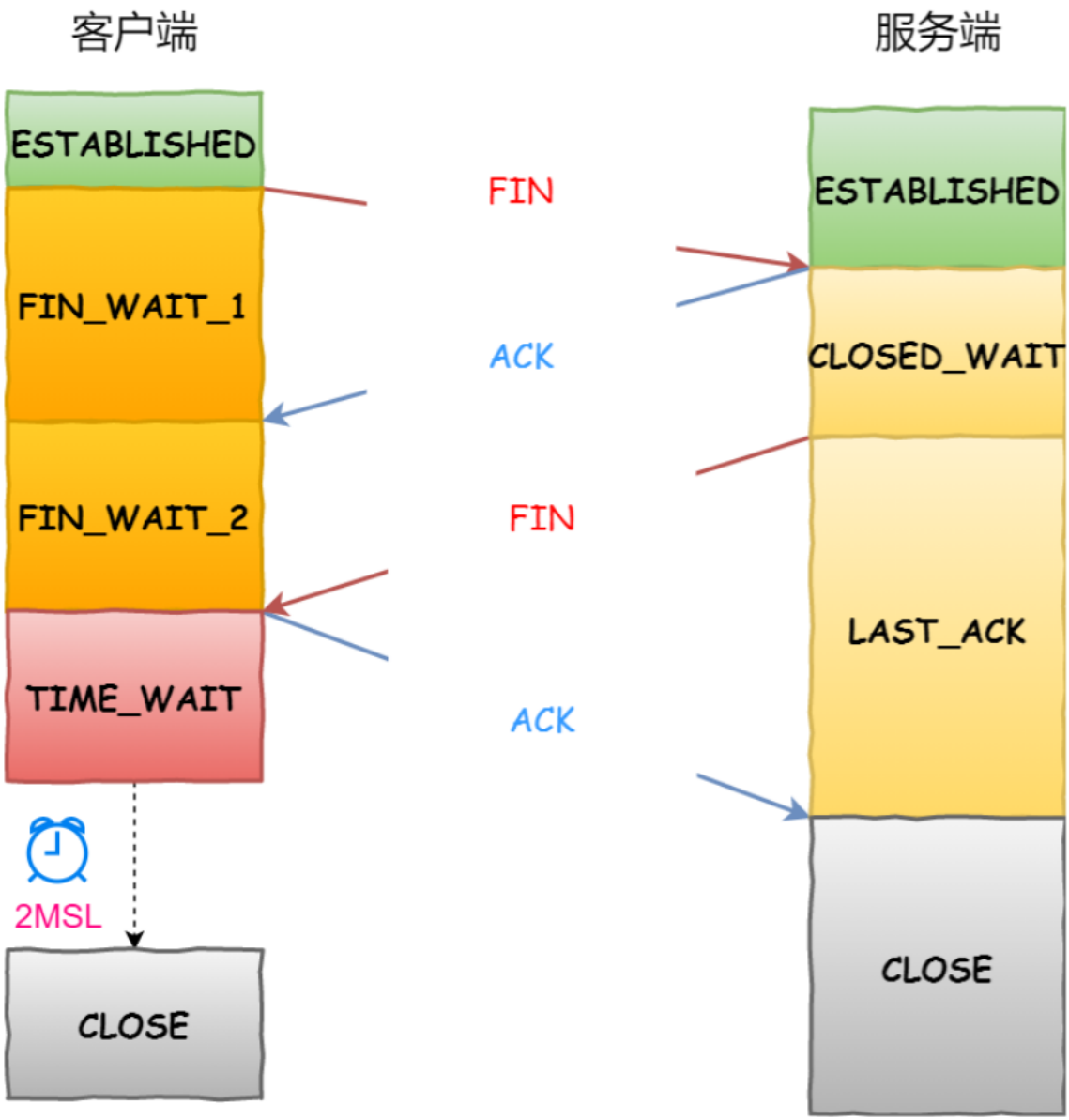
客户端：在初始化完成后用户可以通过输入控制信号进行选择是否建立连接，若请求建立连接，输入服务器的IP地址，之后客户端就会执行**buildConnection**函数，使用**停等发送**功能向服务端发送一个**SYN**请求同步建立连接，进入**SYN_SENT**状态检查收到的ACK是否也为**SYN同步**请求，若是则回复一个ACK，其Ack值为收到的SYN的Seq值加一，之后**连接成功建立**，进入ESTABLISHSED状态。

```
bool buildConnection() {
    Packet sendSYN, recvACK, sendACK;
    sendSYN.Seq = sendSeq++;
    setSYN(&sendSYN);
    state = SYN_SENT;

    if(stopWaitSend(&sendSYN,&recvACK)){
        if (checkSYN(&recvACK)) {
            sendACK.Seq = sendSeq++;
            setACK(&sendACK, &recvACK);
            sendPacket(&sendACK);
            state = ESTABLISHED;
            return 1;
        }
    }
    state = CLOSE;
    return 0;
}
```

断开连接

断开连接机制也是参考了TCP协议中的四次挥手：其流程如下图所示：



服务端：服务端会不断监听数据帧的到来，当接收到数据帧后检查其类型，若收到**FIN**，则执行**breakConnection**函数，进入**CLOST_WAIT**状态。在**breakConnection**函数中，服务端会向客户端回复一个**ACK**确认收到的FIN，之后服务端会使用**停等发送功能**向客户端发送一个**FIN**数据帧，在发送FIN后进入**LAST_ACK**状态，**等待最后一个ACK确认**，收到相应的ACK确认后**连接断开**，进入**CLOSE**状态。

```
bool breakConnection(Packet finPacket) {
    state = CLOSE_WAIT;
    Packet sendACK, sendFIN, recvACK;
    sendACK.Seq = sendSeq++;
    setACK(&sendACK, &finPacket);
    sendPacket(&sendACK);

    setFIN(&sendFIN);//FIN=1
    sendFIN.Seq = sendSeq++;
    if (stopWaitSend(&sendFIN, &recvACK)) {
        state = CLOSE;
```

```
        nowTime = getTime();
        cout << nowTime << " [ BREAK ] " << "The connection is down! Enter the " << state << " state!" << endl;
        return 1;
    }
    return 0;
}
```

客户端：客户端在**ESTABLISHED**状态下可以通过输入控制信号选择**断开连接**，执行**breakConnection**函数，在该函数中客户端会先使用**停等发送功能**发送一个FIN请求，进入**FIN_WAIT_1**状态，当收到**ACK确认**报文后进入**FIN_WAIT_2**状态，之后使用**停等接收功能**，等待一个**FIN**数据包并回复**ACK**确认，表示可以断开连接，此时客户端进入**TIME_WAIT**状态，等待**2MSL**后**成功断开连接**，进入**CLOSE**状态。

```

bool breakConnection() {
    Packet sendFIN, recvACK, recvFIN, sendACK;
    sendFIN.Seq = sendSeq++;
    setFIN(&sendFIN);
    if (stopWaitSend(&sendFIN, &recvACK)) {
        state = FIN_WAIT_2;
    }
    else {
        state = ESTABLISHED;
        return 0;
    }
    if (stopWaitRecv(&recvFIN, &sendACK)) {
        state = TIME_WAIT;
    }
    else {
        state = ESTABLISHED;
        return 0;
    }
    _sleep(2*MSL);
    state = CLOSE;
    return 1;
}
```

差错检测

使用**校验和**检测是否出现差错，其计算与校验的过程如下所示：

```
//设置校验和
void setCheckSum(Packet* t){
    int sum = 0;
    u_char* packet = (u_char*)t;
    for (int i = 0; i < 16; i++){//对报文的前32字节 即256位进行校验计算
        sum += packet[2 * i] << 8 + packet[2 * i + 1];
        while (sum > 0xFFFF) {//溢出后将高十六位加到低十六位上
            int sumh = sum >> 16;
            int suml = sum & 0xFFFF;
            sum = sumh + suml;
        }
    }
```

```
    t->checksum = ~(u_short)sum;//按位取反
}
```

```
//校验
bool checkCheckSum(Packet* t){
    int sum = 0;
    u_char* packet = (u_char*)t;
    for (int i = 0; i < 16; i++) {//对报文的前32字节 即256位进行校验计算
        sum += packet[2 * i] << 8 + packet[2 * i + 1];
        while (sum > 0xFFFF) {//溢出后将高十六位加到低十六位上
            int sumh = sum >> 16;
            int suml = sum & 0xFFFF;
            sum = sumh + suml;
        }
    }
    //校验和与报文中该字段相加，等于0xFFFF则校验成功
    if (t->checksum + (u_short)sum == 0xFFFF) {
        return true;
    }
    return false;
}
```

确认重传

该部分的实现主要在**停等机制**中提到的代码里，在发送数据后会打开**计时器**并进入循环中不断接收数据，对于收到的数据报文会先进行**校验和的计算与检验**，确定数据帧没有问题后，检查是否为**ACK确认**以及其确认序号**Ack的值**是否为发出数据的序号**Seq值加一**，均无问题后表示数据发送成功；而若在**规定时间内**未收到正确的ACK确认，则进行重发，当**重发次数超过最大重传次数**后，**发送失败**放弃发送。

```
//发包函数
bool recvPacket(Packet* recvP) {
    memset(recvP, 0, sizeof(sizeof(&recvP)));
    int re = recvfrom(sockClient, (char*)recvP, sizeof(Packet), 0, (struct
sockaddr*)&sockAddr, &addr_len);
    if (re != -1 && checkCheckSum(recvP)) {//检查校验和
        return true;
    }
    return false;
}
//停等发送函数
bool stopWaitSend(Packet* sendP, Packet* recvP) {
    sendPacket(sendP);
    if (checkFIN(sendP) && state != FIN_WAIT_1) {
        state = FIN_WAIT_1;
    }
    clockStart = clock();    //开始计时
    int retransNum = 0;      //重发次数
    while (1){
        if (recvPacket(recvP)) {
            if (checkACK(recvP) && recvP->Ack == (sendP->Seq + 1)){
                return 1;//收到对sendP的确认报文
            }
        }
        clockEnd = clock();
```

```
        if (retransNum == RETRANSMISSION_TIMES) {    //到达最大重发次数
            return 0;
        }
        if ((clockEnd - clockStart)>= WAIT_TIME){    //超时重发
            retransNum++;
            sendPacket(sendP);
            clockStart = clock();
        }
    }
    return 0;//发送失败
}
```

文件传输

发送端可以通过输入控制信号决定是否要发送文件，输入文件名后执行**sendFile**函数，首先发送端会将文件读入**缓冲区**并根据设置好的**最大数据长度**对文件进行分段，并记录长度；之后会使用**停等发送功能**发送一个设置**SF标志位**的数据包，其中包含文件名，文件分片数目等信息，表示开始发送文件；之后将**分段的文件按序打包**，设置**FILE标志**，其中文件分片序号、分片数据长度等信息，使用**停等发送功能**依次发送，当发送到最后一个包时设置其标志位为**EF**表示文件**发送结束**，计算吞吐率。

```
int sendFile(char* fileName) {
    if (state != ESTABLISHED) {//连接建立后才可发送
        return 0;
    }
    if (!readFile(fileName)) {
        return 0;
    }
    clock_t timeStart = clock();//timer
    int nameLength = strlen(fileName);
    Packet sendFileName, recvACK;;
    setStart(&sendFileName);
    sendFileName.Seq = sendSeq++;
    sendFileName.index = PacketNum;
    sendFileName.length = nameLength;
    for (int i = 0; i < nameLength; i++) {
        sendFileName.Data[i] = fileName[i];
    }
    if (!stopWaitSend(&sendFileName, &recvACK)) {
        return 0;
    }
    for (int i = 0; i <= PacketNum; i++) {
        Packet sendFile;
        sendFile.Seq = sendSeq++;
        sendFile.index = i;
        setFile(&sendFile);
        if (i == PacketNum) {
            setEnd(&sendFile);
            sendFile.length = data_p;
        }
        else {
            sendFile.length = BUF_SIZE;
        }
        for (int j = 0; j < sendFile.length; j++) {
            sendFile.Data[j] = dataToSend[i][j];
```

```
        }
        if (!stopWaitSend(&sendFile, &recvACK)) {
            return 0;
        }
    }
    //吞吐率计算
    clock_t timeEnd = clock();
    double totalTime = (double)(timeEnd - timeStart) / CLOCKS_PER_SEC;
    double RPS = (double)(PacketNum + 2) * sizeof(Packet) * 8 / totalTime /
1024 / 1024;
    return 1;
}
```

接收端在接收到**SF**数据包后会执行**recvFile**函数，先将SF数据包进行解析，得到文件名、文件分片数量等信息，之后进入**停等接收状态**，依次接收后续的文件分片内容，直到收到最后一片时检查**EF位**，文件接收结束，将文件导出。

```
int recvFile(Packet recvFileName){
    Packet sendACK;
    sendACK.Seq = sendSeq++;
    setACK(&sendACK, &recvFileName);
    sendPacket(&sendACK);
    //获取文件名
    int PacketNum = recvFileName.index;
    int nameLength = recvFileName.length;
    char* fileName = new char[nameLength + 1];
    memset(fileName, 0, nameLength + 1);
    for (int i = 0; i < nameLength; i++) {
        fileName[i] = recvFileName.Data[i];
    }
    fileName[nameLength] = '\0';

    int lastLength = 0;
    for (int i = 0; i <= PacketNum; i++) {
        Packet recvFile, sendACK;
        memset(recvData[i], 0, BUF_SIZE);
        if (stopWaitRecv(&recvFile, &sendACK)) {
            if (recvFile.index != i) {
                return 0;
            }
            for (int j = 0; j < recvFile.length; j++) {
                recvData[i][j] = recvFile.Data[j];
            }
        }
        else {
            return 0;
        }
        if (i == PacketNum) {
            lastLength = recvFile.length;
            if (!checkEnd(&recvFile)) {
                return 0;
            }
        }
    }
    outFile(fileName, PacketNum, BUF_SIZE, lastLength);
```

```
    return 1;
}
```

日志输出

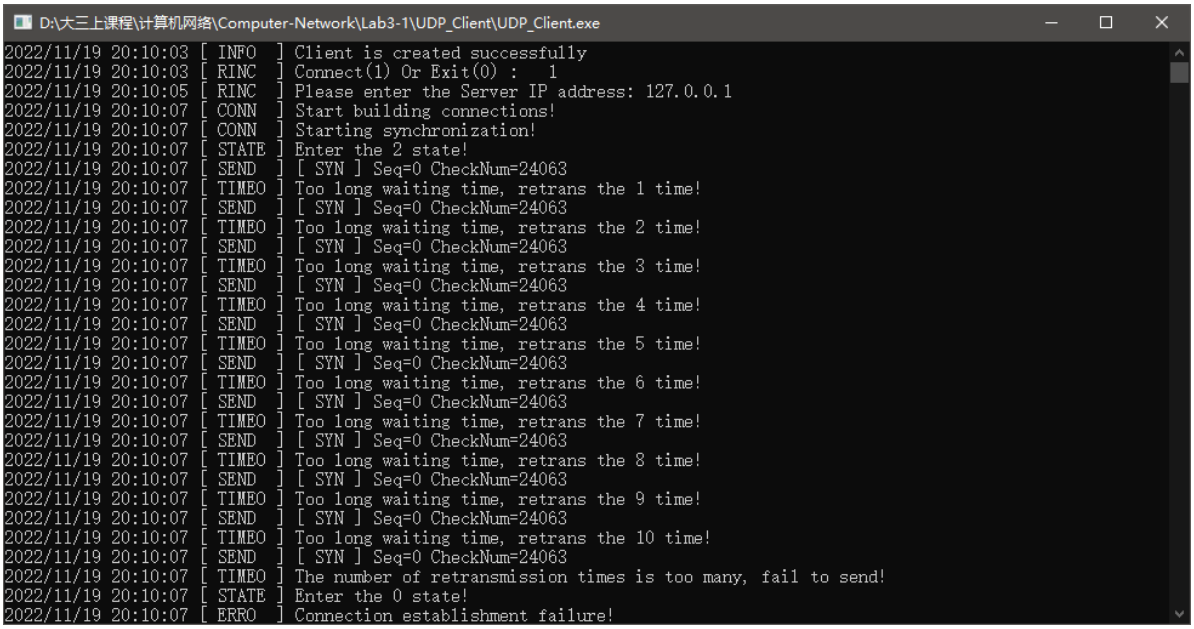
对于客户端与服务端运行时的不同状态，均会在相应的地方进行日志的输出打印，此外对于报文的内容也会进行打印，该部分主要由如下代码实现：

```
string PacketLog(Packet* toLog) {
    string log = "[ ";
    if (checkSYN(toLog)) {
        log += "SYN ";
    }
    if (checkACK(toLog)) {
        log += "ACK ";
    }
    if (checkFIN(toLog)) {
        log += "FIN ";
    }
    if (checkStart(toLog)) {
        log += "SF ";
    }
    if (checkFile(toLog)) {
        log += "FILE ";
    }
    if (checkEnd(toLog)) {
        log += "EF ";
    }
    log += "] ";
    log += "Seq=";
    log += to_string(toLog->Seq);
    if (checkACK(toLog)) {
        log += " Ack=";
        log += to_string(toLog->Ack);
    }
    if (checkFile(toLog)) {
        log += " Index=";
        log += to_string(toLog->index);
    }
    log += " CheckNum=";
    log += to_string(toLog->checksum);
    return log;
}
```

程序执行演示

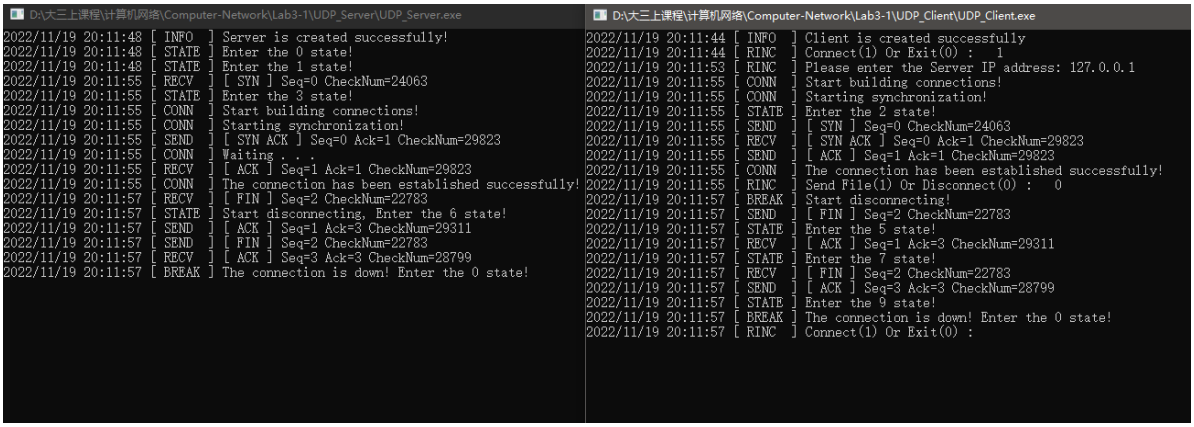
超时重传

在客户端对未启动的服务端发起连接，检验超时重传功能：



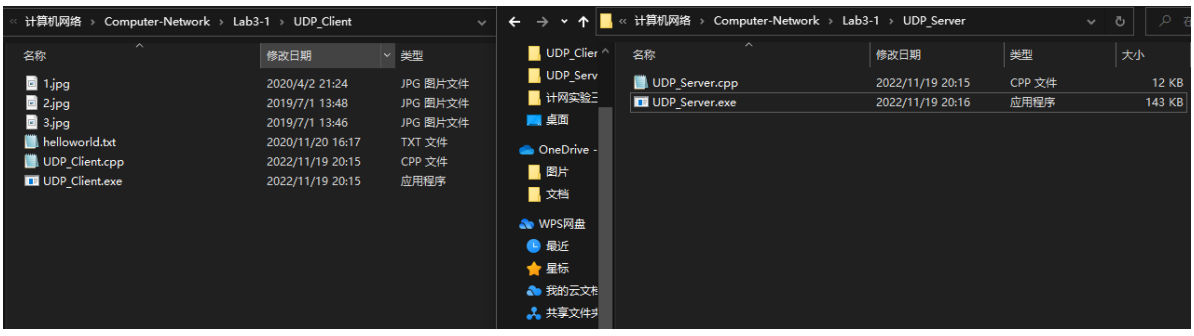
建立连接与断开连接

测试请求建立连接与断开连接，如下可见在三次握手后成功建立连接，四次挥手后成功断开连接。

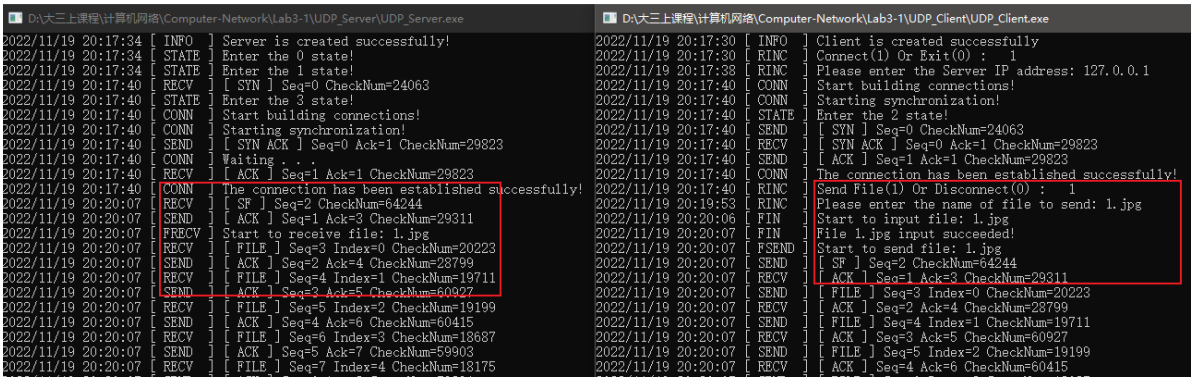


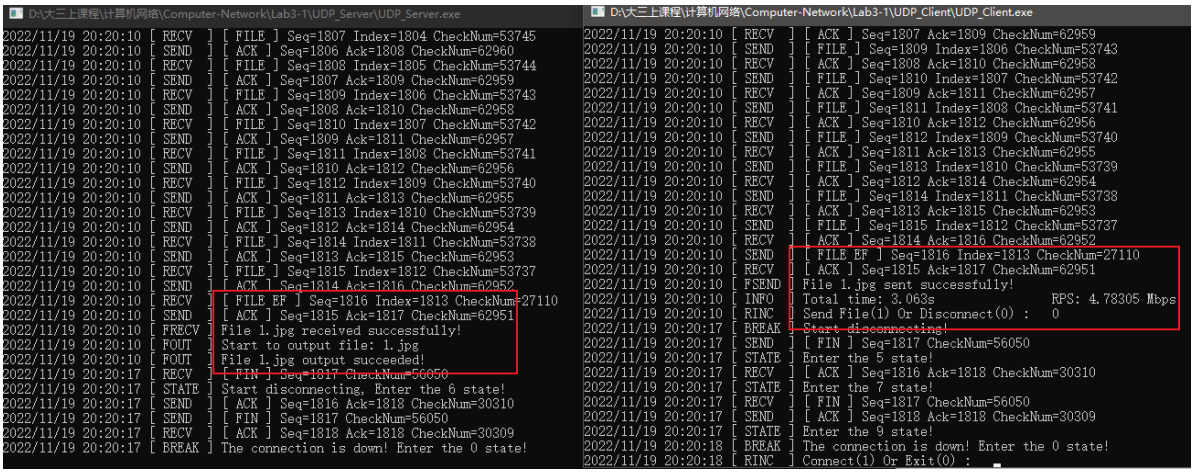
测试文件传输

首先将测试文件移到客户端目录下，并确保服务端目录下无其他文件：



之后打开客户端与服务端开始传输测试文件 1.jpg：





如下图可见对应目录下出现了刚刚传输的图片1.jpg，且能够正常的打开，说明文件成功传输，其中由于I/O占用较长时间，因此吞吐率RPS较低。