

实验3-2

学号：2112066 姓名：于成俊

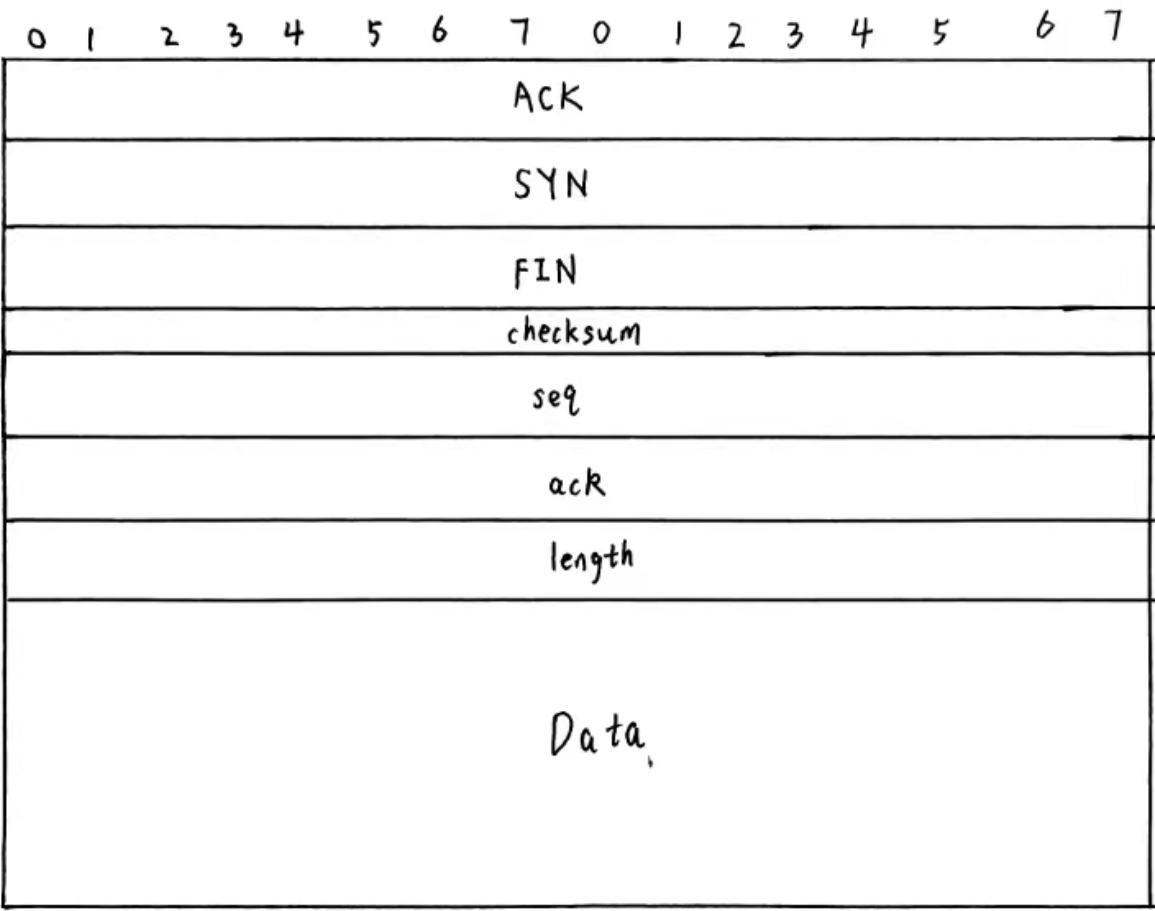
一、实验题目：

在实验3-1的基础上，将停等机制改成基于滑动窗口的流量控制机制，发送窗口大于1，接收窗口为1，支持累积确认，完成给定测试文件的传输。

二、协议设计

1.报文格式：

所设计的报文格式如下图所示：

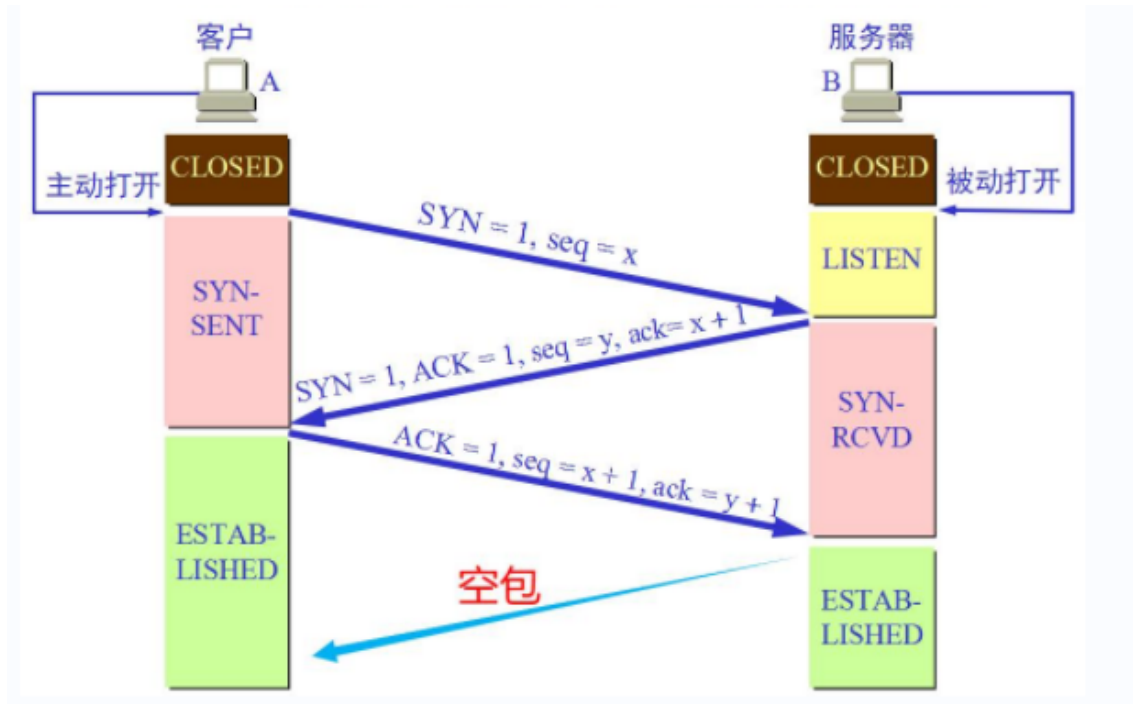


- 报文总长度为82128位，即10266字节。
- 前32位为ACK，用于确认序号有效。
- 33—64位为SYN，用于发起一个连接。
- 65—96位为FIN，用于释放一个连接；以及在数据传输过程中，用于告知数据已传输完毕。
- 97—112位为checksum，即校验和，用于差错检验。
- 113—144位为seq，为传输数据包的序列号。在建立连接和断开连接时，为随机取值；在数据传输时，只为0和1。

- 145—176位为ack，为确认序列号，只在建立连接和断开连接时使用，在数据传输时不涉及，取值为seq+1。
- 177—208位为length，为数据长度。
- 209—82128位为数据。

2.四次握手建立连接

- 参考于TCP的三次握手，所设计的流程图如下：



- 首先客户端向服务器端发送一个报文，其 SYN 标志位置 1，标志请求建立连接；并从1~100中随机选取一个整数x赋值给seq。
- 服务器收到请求后，向客户端回复一个报文，SYN 和 ACK 标志位置 1，标志允许建立连接；并从1~100中随机选取一个整数y赋值给seq。
- 客户端收到服务器反馈后，向服务器发送一个报文，ACK 置 1，标志将要开始传输；且将seq设为x+1，将ack设为y+1；
- 最后服务器再发送个空包（任意一个包也行）给客户端，这样，客户端收到一个空包就知道服务器端已经收到了第三次的报文。这就是第四次握手。

3.可靠数据传输

发送端和接收端均参考GBN协议，具体如下：

- 允许客户端（发送端）连续发出N个未得到确认的分组。其中，**N为发送窗口的大小**。
- 与停等协议相比，增加了序列号的范围。seq的取值为1到MaxSeq-1。其中，MaxSeq表示序号空间的大小。注意：**序号空间应大于窗口大小**，即 $MaxSeq > N$ 。
- 服务器（接收端）采取**累计确认**的方式，只确认连续正确接收分组的最大序列号。
- 服务器（接收端）需要保存希望接收的分组序号。

具体的交互过程如下：

客户端：

- 开始时，客户端连续发出多个分组直至发送窗口满为止。在此过程中，客户端需要计算每个分组的**校验和**，并将校验和连同数据一起发送。然后，客户端开启发送的第一个分组的超时定时器。

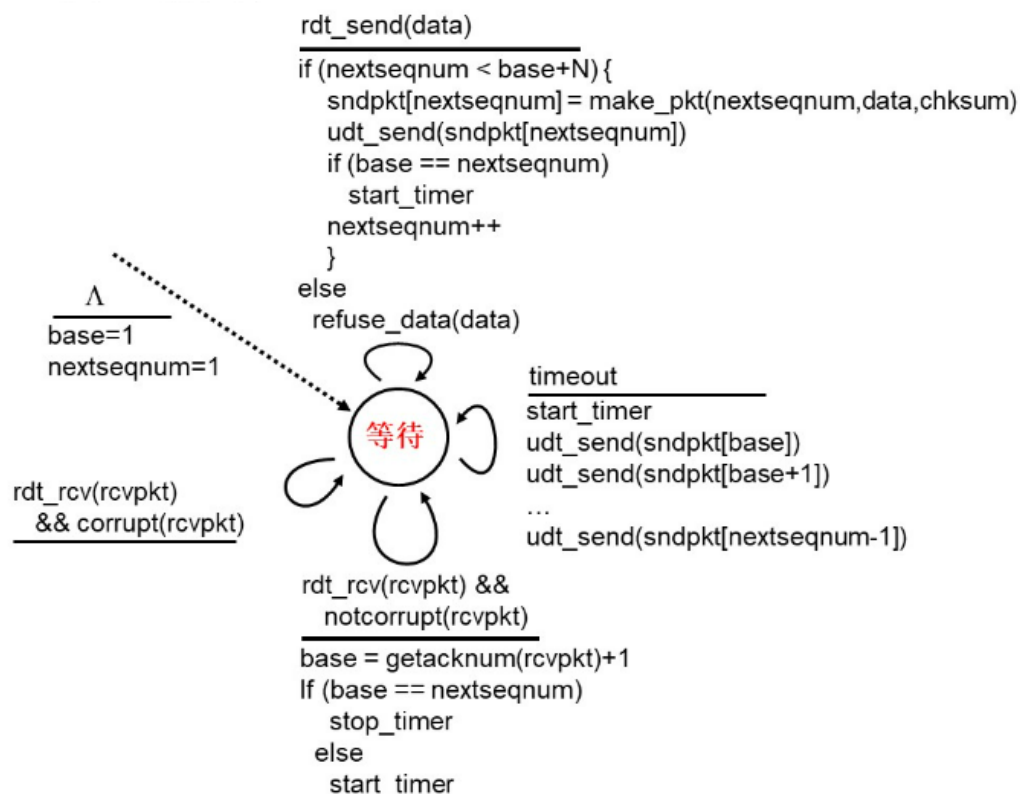
- 如果接收到第一个分组的确认号，则将第一个分组从发送窗口移除，再发送一个新的分组；
- 如果没有接收到第一个分组的确认号而是接收到第一个分组后面的分组的确认号，则将该确认的分组及其之前的分组从发送窗口移除，然后再连续发送多个分组直至发送窗口满为止。
- 如果没有接收到第一个分组的确认号，也没有接收到第一个分组后面的分组的确认号。则当超时定时器报警后，将发送窗口的分组重新发送，循环以上过程。
- 当数据发送完成后，向服务器发送FIN=-1的数据报，告知数据已发送完。

服务器：

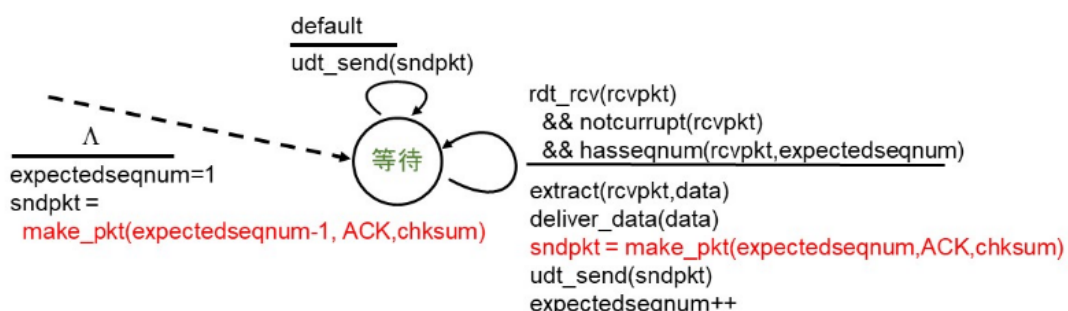
- 接收端等待来自客户端发送的分组，接收到后计算校验和并检查序列号。
- 若校验和不为0，说明分组出现错误，则给客户端发送ACK=seq的分组(该seq为最新已正确接收到的分组的seq)。表示分组出现错误，要求重传。
- 若校验和为0且分组的序列号是期望的序列号，则给客户端发送ACK=seq的分组(该seq为刚接收到的分组的seq)。表示已正确接收分组。
- 若校验和为0而分组的序列号不是期望的序列号，说明出现乱序或者收到重复分组的情况，则直接丢弃并给客户端发送ACK=seq的分组(该seq为最新已正确接收到的分组的seq)。告知客户端想接受的分组的序号。
- 若收到FIN=-1且校验和为0的分组，则表示发送数据接收，不再等待接收该文件数据。并随便发送个报文，告知对方我已知道。

状态机：

- 发送端的有限状态机如下图所示：

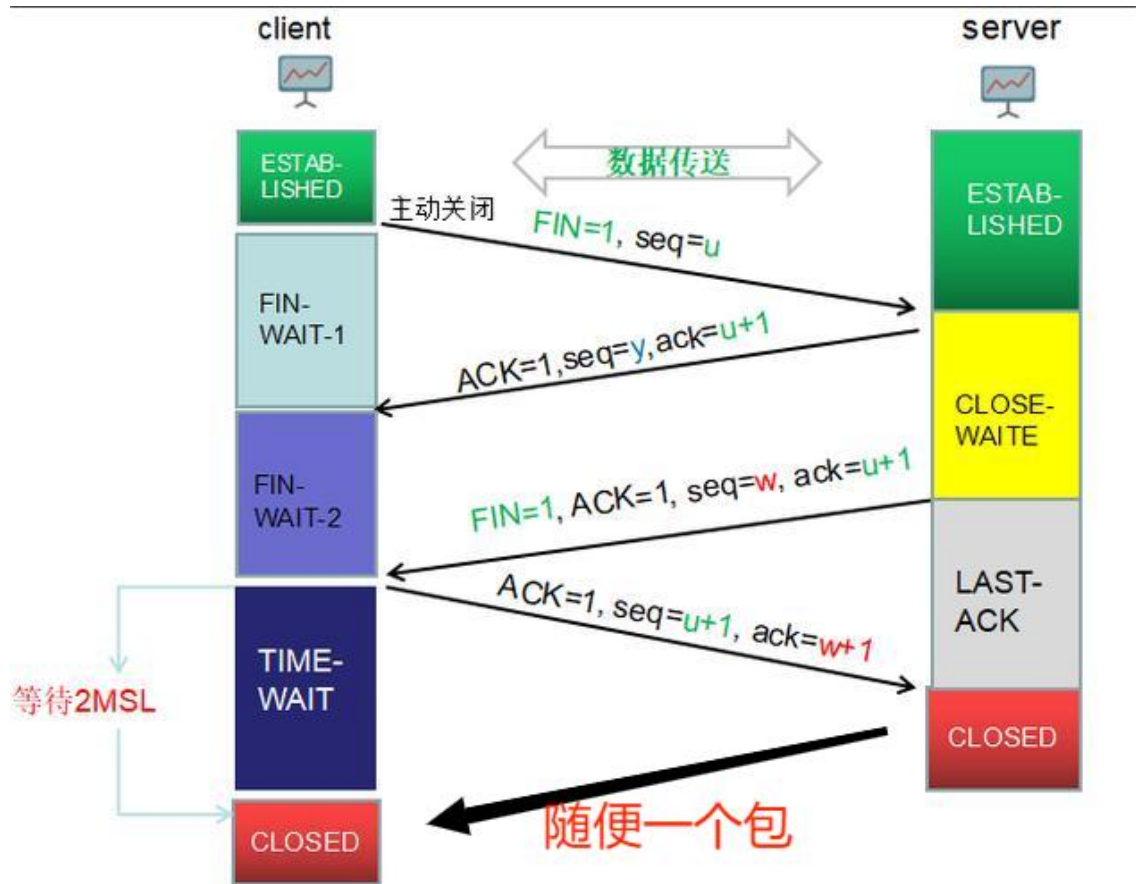


- 接收端的有限状态机如下图所示：



4.五次挥手断开连接

- 参考TCP的四次挥手，所设计的流程图如下：



- 客户端向服务器端发送一个报文，将 FIN 标志位置 1，并从1~100中随机选取一个整数u赋值给 seq，标识请求断开连接。
- 服务器端收到断开请求后，回应一个报文，将 ACK 标志位置 1，并从1~100中随机选取一个整数y赋值给seq，且令ack=u+1，标识接到断开请求。
- 服务器端向客户端发送一个报文，将ACK和FIN 标志位置 1，并从1~100中随机选取一个整数w赋值给seq，且令ack=u+1，标识请求断开连接。
- 客户端收到断开请求后，回应一个报文，将 ACK 标志位置 1，令seq=u+1，ack=w+1。
- 最后，服务器随便发送一个包，我已关闭。客户端接收到后，关闭连接。

三、设计实现

1.报文格式：

```
const int BUFFER_SIZE = 10240;
//数据报
struct Datagram {
    //标志位
    int ACK, SYN, FIN;
    //校验和
    unsigned short int checksum; //unsigned short int是16位
    int seq, ack; //序列号和确认号
    int length; //数据长度
    char data[BUFFER_SIZE]; //数据
};
```

2.差错检验机制实现（计算校验和）：

- 发送方生成校验和：
 - 将接受的数据报分成若干个16位的位串，每个位串看成一个二进制数。
 - 将校验和域段清零，该字段也参与校验和运算。
 - 对这些16位的二进制数进行反码求和。
 - 将累加的结果再取反，得到校验和，放入校验和域段

```
//发送时计算校验和
void send_calculate_checksum(Datagram& datagram) {
    unsigned short int* buff = (unsigned short int*) & datagram;
    int num = sizeof(Datagram) / sizeof(unsigned short int); //有多少16位
    datagram.checksum = 0; //将校验和域段清0
    unsigned long int checksum = 0; //unsigned long int占四个字节
    while (num--)
    {
        checksum += *buff;
        buff++; //指向下一个16位
        //若超出16位，即有进位
        if (checksum & 0xffff0000)
        {
            //将超出16位的部分置0
            checksum &= 0xffff;
            //进位+1
            checksum++;
        }
    }
    //取反
    datagram.checksum = ~(checksum & 0xffff);
}
```

- 接收方生成校验和：

除了不用将校验和域段清零外，其他步骤一样。若结果为0，则没错误。

3.超时重传机制实现（与3-1不同）：

在实验3-1中，因为是停等协议，所以我使用了 `setsockopt()` 函数来实现超时重传。而本次实验，是GBN协议，发送端可以连续发送未确认的分组，`setsockopt()` 函数当 `recvfrom` 函数在规定时间内收到信息就不会超时，**但这次实验需要接受到在发送窗口里的分组序号才不会超时**，`setsockopt()` 函数无法满足这个需求。所以我采用了将 `recvfrom()` 函数设置为非阻塞状态来实现。即使用 `ioctlsocket()` 函数

`ioctlsocket()` 函数原型如下：

```
int ioctlsocket( SOCKET s, long cmd, u_long FAR *argp );
```

各参数定义：

- s：一个标识套接口的描述字。
- cmd：对套接口s的操作命令。
- argp：指向cmd命令所带参数的指针。

使用方法如下：

```
// 设置Socket为非阻塞模式
bool setNonBlocking(SOCKET socket) {
    u_long mode = 1; // 1表示非阻塞，0表示阻塞
    if (ioctlsocket(socket, FIONBIO, &mode) == SOCKET_ERROR) {
        cerr << "设置非阻塞模式失败！ " << endl;
        return false;
    }
    return true;
}
```

在某些情况下，还需设置为阻塞状态。

```
// 设置Socket为阻塞模式
bool setBlocking(SOCKET socket) {
    u_long mode = 0; // 0表示阻塞，1表示非阻塞
    if (ioctlsocket(socket, FIONBIO, &mode) == SOCKET_ERROR) {
        cerr << "设置阻塞模式失败！ " << endl;
        return false;
    }
    return true;
}
```

本次实验的**超时重传功能**都是基于下面的方式实现的。（超时时间设置为500毫秒）

```
//时间限制500毫秒
const int TIMEOUT = 500;
while (true) {
    /*
    ... .. 发送数据 ... ..
    */
    //定时器启动
    clock_t start = clock();
    bool istimeout = true;
    while (clock() - start <= TIMEOUT) {
        // 接收数据
```

```

        if (recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
            continue;
        }
        istimeout = false;
        break;
    }
    if (!istimeout) {
        break;
    }
}
}

```

4.滑动窗口实现:

- 设置序号空间大小和发送窗口大小:

```

// 发送窗口大小
const int windowSize = 33;
// 传输数据时的序号空间大小
const int MaxSeq = 100;

```

- 通过队列queue来实现发送窗口

```

//发送窗口
queue<Datagram>SendDataqueue;

```

- 使用队列来表示窗口有以下几个原因:
 - 队列是一种**先进先出**的数据结构, 与发送窗口的滑动方式完美契合, 使用非常便利。如下:

```

//当发送新的分组时, 只需用push函数
SendDataqueue.push(SendData);
//当丢弃已确认的分组时, 只需用pop函数
SendDataqueue.pop();

```

- 当想获得位于发送窗口下沿和上沿的分组时, 只需如下操作:

```

//位于下沿的分组序号
SendDataqueue.front().seq
//位于上沿的分组序号
SendDataqueue.back().seq

```

- 发送数据时:

```

//发送数据
while (SendDataqueue.size() < windowSize) {
    if (file.eof()) {
        break;
    }
    memset(&SendData, 0, sizeof(SendData));
    SendData.seq = sequenceNumber++;
    sequenceNumber = sequenceNumber % MaxSeq ;
    //每次从文件中读取最多BUFFER_SIZE 字节的数据

```

```

file.read(SendData.data, BUFFER_SIZE);
int bytesToSend = static_cast<int>(file.gcount());
totalBytes += bytesToSend;
//设置数据长度
SendData.length = bytesToSend;
//计算校验和
send_calculate_checksum(SendData);
//加入到窗口
SendDataqueue.push(SendData);
//打印发送的数据报
cout << "发送数据报: ";
printDatagram(SendData);
if (sendto(ClientSocket, (char*)&SendData, sizeof(SendData), 0,
(SOCKADDR*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
    cerr << "发送错误!" << endl;
}
cout << "发送窗口大小为: " << SendDataqueue.size() << endl;
cout << "发送窗口的下沿为:" << SendDataqueue.front().seq << endl;
cout << "发送窗口的上沿为:" << SendDataqueue.back().seq << endl;
}

```

- 接收确认号，将已确认的分组移除发送窗口：（需要判断确认号是否在发送窗口中，有三种情况）

①情况一：

确认号为1				
1	2	3	4	5
确认号为5				
5	0	1	2	3

②情况二：

确认号为10				
9	10	11	0	1
确认号为1				
10	11	0	1	2

③情况三：

确认号为3				
1	2	3	4	5

代码实现如下:

```
//情况一
if (ReceiveData.checksum == 0 && ReceiveData.ACK == SendDataqueue.front().seq) {
    SendDataqueue.pop();
    istimeout = false;
    break;
}
//情况二
if (ReceiveData.checksum == 0 && SendDataqueue.front().seq >
SendDataqueue.back().seq && ReceiveData.ACK >=0 ) {
    if (ReceiveData.ACK > SendDataqueue.front().seq || ReceiveData.ACK <=
SendDataqueue.back().seq) {
        while (ReceiveData.ACK == SendDataqueue.front().seq) {
            SendDataqueue.pop();
        }
        SendDataqueue.pop();
        istimeout = false;
        break;
    }
}
//情况三
if (ReceiveData.checksum == 0 && SendDataqueue.front().seq <
SendDataqueue.back().seq) {
    if (ReceiveData.ACK > SendDataqueue.front().seq && ReceiveData.ACK <=
SendDataqueue.back().seq) {
        while (ReceiveData.ACK == SendDataqueue.front().seq) {
            SendDataqueue.pop();
        }
        SendDataqueue.pop();
        istimeout = false;
        break;
    }
}
```

5.四次握手建立连接:

- 握手过程中的差错检验是靠校验和实现的, 若有错误, 则等待对方重发。
- 握手过程还需检验ack、ACK、SYN等标志位是否等于预期值, 若不等于, 则等待对方重发。
- 超时重传机制。
 - 规定发出握手后, 时间不能超过1800秒。若超过, 则从头再来。

```
const int maxtime = 1800; //1800秒
```

- 规定请求建立连接后，不能超过3600秒。否则，连接失败。

```
const int MAXTIME = 3600;//3600秒
```

- 相关代码如下：

```
/*-----发送端：主动握手-----*/
clock_t startTime = clock();
//主动发出握手
while (!active_shakehands(clientSocket, routerAddr,
sizeof(routerAddr))) {
    //超时
    if ((clock() - startTime) / CLOCKS_PER_SEC > MAXTIME) {
        cout << "连接不成功！" << endl;
        // 关闭套接字和清理winsock
        closesocket(clientSocket);
        WSACleanup();
        return -1;
    }
}
cout << "连接成功！" << endl;
/*-----接收端：被动握手-----*/
clock_t startTime = clock();
//被动接收握手
while (!passive_shakehands(serverSocket, routerAddr, routerAddrLen))
{
    //超时
    if ((clock() - startTime) / CLOCKS_PER_SEC > MAXTIME) {
        cout << "连接不成功！" << endl;
        // 关闭套接字和清理winsock
        closesocket(serverSocket);
        WSACleanup();
        return -1;
    }
}
cout << "连接成功！" << endl;
```

- 四次握手实现过程如下：（只有关键代码）

- 客户端（发送端）：

```
//主动握手
bool active_shakehands(SOCKET& clientsocket, sockaddr_in& routerAddr,
int routerAddrLen) {
    clock_t total_time = clock();
    while (true) {
        //超时连接不成功！
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        // 将SYN设置为1，表示要建立连接
        SendData.SYN = 1;
        //随机设定seq(1到100)
```

```

        SendData.seq = (rand() % 100) + 1;
        //计算校验和
        send_calculate_Checksum(SendData);
        if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(clientsocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <=
0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            break;
        }
    }
    cout << "第一次握手成功" << endl;
    //计算校验和
    receive_calculate_Checksum(ReceiveData);
    //改为阻塞状态
    if (!setBlocking(clientsocket)) {
        return false;
    }
    while (!(ReceiveData.ACK == 1 && ReceiveData.SYN == 1 &&
ReceiveData.ack == SendData.seq + 1 && ReceiveData.checksum == 0)) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen);
        //计算校验和
        receive_calculate_Checksum(ReceiveData);
    }
    cout << "第二次握手成功!" << endl;
    //改为非阻塞状态
    if (!setNonBlocking(clientsocket)) {
        return false;
    }
    while (true) {
        //超时连接不成功!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
    }

```

```

        SendData.ack = 1;
        SendData.seq = ReceiveData.ack;
        SendData.ack = ReceiveData.seq + 1;
        //计算校验和
        send_calculate_Checksum(SendData);
        if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
        (struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(clientsocket, (char*)&ReceiveData,
            sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <=
            0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            break;
        }
    }
    cout << "第三次握手成功" << endl;
    cout << "第四次握手成功" << endl;
    return true;
}

```

○ 服务端（接收端）：

```

//被动握手
bool passive_shakehands(SOCKET& serversocket, sockaddr_in& routerAddr,
int routerAddrLen) {
    clock_t total_time = clock();
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }
    while (true) {
        //超时连接不成功!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        // 接收数据
        if (recvfrom(serversocket, (char*)&ReceiveData,
        sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) ==
        SOCKET_ERROR) {
            continue;
        }
    }
}

```

```

        //计算校验和
        receive_calculate_Checksum(ReceiveData);
        if (!(ReceiveData.checksum == 0 && ReceiveData.SYN == 1)) {
            continue;
        }
        break;
    }
    cout << "第一次握手成功" << endl;
    //设置为非阻塞状态
    if (!setNonBlocking(serversocket)) {
        return false;
    }
    while (true) {
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        SendData.SYN = 1;
        SendData.ACK = 1;
        SendData.seq = (rand() % 100) + 1;
        SendData.ack = ReceiveData.seq + 1;
        //计算校验和
        send_calculate_Checksum(SendData);
        if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(serversocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <=
0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            cout << "第二次握手成功" << endl;
            break;
        }
    }
    //计算校验和
    receive_calculate_Checksum(ReceiveData);
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }
    while (!(ReceiveData.ACK == 1 && ReceiveData.seq == SendData.ack &&
ReceiveData.ack == SendData.seq + 1 && ReceiveData.checksum == 0)) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {

```

```

        return false;
    }
    //将要接受的数据清空
    memset(&ReceiveData, 0, sizeof(Datagram));
    if (recvfrom(serversocket, (char*)&ReceiveData,
        sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) ==
        SOCKET_ERROR) {
        continue;
    }
    //计算校验和
    receive_calculate_checksum(ReceiveData);
}
cout << "第三次握手成功" << endl;
//将要发送的数据清空
memset(&SendData, 0, sizeof(Datagram));
//发送一个空包
if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0,
    (struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
    cerr << "发送错误!" << endl;
}
cout << "第四次握手成功" << endl;
return true;
}

```

6.可靠数据传输

- 客户端（发送端）（包括相关日志输出、传输时间、吞吐量、发送窗口大小即变化、丢包提示）
完全发送一个文件后，再调用该函数，发送另一个文件。

```

//传输文件
int transform_file(char filepath[], SOCKET& clientSocket, sockaddr_in&
    routerAddr, int& routerAddrLen) {
    cout << "现在开始发送 " << filepath << endl;
    // 以二进制模式打开文件（图片或文本文件）
    ifstream file(filepath, ios::binary);
    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        closesocket(clientSocket);
        WSACleanup();
        return 0;
    }
    //文件总字节
    int totalBytes = 0;
    // 序号
    int sequenceNumber = 0; //从0开始
    //开始传输时间
    clock_t startTime = clock();
    //定时器
    clock_t start;
    //执行到文件末尾（end of file, EOF）
    while (true) {
        //发送数据
        while (SendDataqueue.size() < windowSize) {

```

```

        if (file.eof()) {
            break;
        }
        memset(&SendData, 0, sizeof(SendData));
        SendData.seq = sequenceNumber++;
        sequenceNumber = sequenceNumber % MaxSeq ;
        //每次从文件中读取最多BUFFER_SIZE 字节的数据
        file.read(SendData.data, BUFFER_SIZE);
        int bytesToSend = static_cast<int>(file.gcount());
        totalBytes += bytesToSend;
        //设置数据长度
        SendData.length = bytesToSend;
        //计算校验和
        send_calculate_checksum(SendData);
        //加入到窗口
        SendDataqueue.push(SendData);
        //打印发送的数据报
        cout << "发送数据报: ";
        printDatagram(SendData);
        if (sendto(clientSocket, (char*)&SendData, sizeof(SendData), 0,
(SOCKADDR*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        cout << "发送窗口大小为: " << SendDataqueue.size() << endl;
        cout << "发送窗口的下沿为:" << SendDataqueue.front().seq << endl;
        cout << "发送窗口的上沿为:" << SendDataqueue.back().seq << endl;
    }
    //清空接收数据
    memset(&ReceiveData, 0, sizeof(ReceiveData));
    //等待ACK反馈
    while (true) {
        //定时器启动
        start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(clientSocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
                continue;
            }
            //计算校验和
            receive_calculate_checksum(ReceiveData);
            //打印接收的数据报
            cout << "接收数据报: ";
            printDatagram(ReceiveData);
            if (ReceiveData.checksum == 0 && ReceiveData.ACK ==
SendDataqueue.front().seq) {
                SendDataqueue.pop();
                istimeout = false;
                break;
            }
            if (ReceiveData.checksum == 0 && SendDataqueue.front().seq >
SendDataqueue.back().seq && ReceiveData.ACK >=0 ) {
                if (ReceiveData.ACK > SendDataqueue.front().seq ||
ReceiveData.ACK <= SendDataqueue.back().seq) {
                    while (ReceiveData.ACK == SendDataqueue.front().seq)
                    {
                        SendDataqueue.pop();

```

```

        }
        SendDataqueue.pop();
        istimeout = false;
        break;
    }
}
}
if (ReceiveData.checksum == 0 && SendDataqueue.front().seq <
SendDataqueue.back().seq) {
    if (ReceiveData.ACK > SendDataqueue.front().seq &&
ReceiveData.ACK <= SendDataqueue.back().seq) {
        while (ReceiveData.ACK == SendDataqueue.front().seq)
        {
            SendDataqueue.pop();
        }
        SendDataqueue.pop();
        istimeout = false;
        break;
    }
}
//清空接收数据
memset(&ReceiveData, 0, sizeof(ReceiveData));
}
//超时
if (istimeout) {
    cout << "分组" << SendDataqueue.front().seq << "丢了!!! 重新发
送分组" << SendDataqueue.front().seq << "及其之后的分组" << endl;
    //将发送窗口的分组重新发送
    int size = SendDataqueue.size();
    for (int i = 1; i <= size; i++) {
        SendData = SendDataqueue.front();
        if (sendto(clientSocket, (char*)&SendData,
sizeof(SendData), 0, (SOCKADDR*)&routerAddr, sizeof(routerAddr)) ==
SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        SendDataqueue.push(SendDataqueue.front());
        SendDataqueue.pop();
    }
}
else {
    break;
}
}
if (file.eof() && SendDataqueue.size() == 0) {
    break;
}
}
file.close();
clock_t endTime = clock();
double transferTime = static_cast<double>(endTime - startTime) /
CLOCKS_PER_SEC;
double throughput = (totalBytes / 1024) / transferTime; // 计算吞吐量 (单
位: KB/s)
cout << filepath << "传输完毕" << endl;
cout << "传输文件大小为: " << totalBytes << "B" << endl;
cout << "传输时间为: " << transferTime << "s" << endl;
cout << "吞吐量为: " << throughput << "KB/s" << endl;
//告诉对方文件传输完了

```



```

while (true) {
    memset(&SendData, 0, sizeof(SendData));
    SendData.FIN = -1;
    send_calculate_Checksum(SendData);
    //打印发送的数据报
    cout << "发送数据报: ";
    printDatagram(SendData);
    sendto(clientSocket, (char*)&SendData, sizeof(SendData), 0,
(SOCKADDR*)&routerAddr, sizeof(routerAddr));
    //定时器启动
    start = clock();
    bool istimeout = true;
    while (clock() - start <= TIMEOUT) {
        // 接收数据
        if (recvfrom(clientSocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
            continue;
        }
        istimeout = false;
        break;
    }
    if (!istimeout) {
        break;
    }
}
return 1;
}

```

- 服务器（接收端）：

接收完一个文件，再调用该函数，接收另一个文件。

```

//接收文件
int receive_file(char filepath[], SOCKET& serverSocket, sockaddr_in&
routerAddr, int& routerAddrLen) {
    cout << "开始接收" << filepath << endl;
    // 打开文件用于写入接收的数据
    ofstream outputFile(filepath, ios::binary);
    if (!outputFile) {
        cerr << "Failed to open the output file." << endl;
        closesocket(serverSocket);
        WSACleanup();
        return 0;
    }
    // 接收和重组数据包
    //目前接受的最大分组号
    int sequenceNumber = -1; //规定发送的第一个序号为0
    while (true) {
        memset(&ReceiveData, 0, sizeof(Datagram));
        int receivedBytes = recvfrom(serverSocket, (char*)&ReceiveData,
sizeof(ReceiveData), 0, (SOCKADDR*)&routerAddr, &routerAddrLen);
        if (receivedBytes == SOCKET_ERROR) {
            continue;
        }
        receive_calculate_Checksum(ReceiveData);
        //打印接受的数据报
    }
}

```

```

        cout << "接收数据报: ";
        printDatagram(ReceiveData);
        if (ReceiveData.FIN == -1 && ReceiveData.checksum == 0) {
            break;
        }
        //接收到重复数据报或错误的数据报
        if (ReceiveData.seq != ((sequenceNumber + 1)%MaxSeq) || !
(ReceiveData.checksum == 0)) {
            //将要发送的数据清空
            memset(&SendData, 0, sizeof(Datagram));
            SendData.ACK = sequenceNumber;
            send_calculate_Checksum(SendData);
            //打印发送的数据报
            cout << "发送数据报: ";
            printDatagram(SendData);
            if (sendto(serverSocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
                cerr << "发送错误!" << endl;
            }
            continue;
        }
        outputFile.write(ReceiveData.data, ReceiveData.length);
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        sequenceNumber = (sequenceNumber + 1) % MaxSeq;
        SendData.ACK = sequenceNumber;
        send_calculate_Checksum(SendData);
        //打印发送的数据报
        cout << "发送数据报: ";
        printDatagram(SendData);
        if (sendto(serverSocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
    }
    //随便发送个消息,告诉他我知道文件传输完了
    if (sendto(serverSocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    outputFile.close();
    cout << filepath << " 文件接收完毕!" << endl;
    return 1;
}

```

7.五次挥手断开连接:

- 客户端 (发送端): (只粘贴关键代码)

```

//主动挥手
bool active_wakehands(SOCKET& clientsocket, sockaddr_in& routerAddr, int
routerAddrLen) {
    clock_t total_time = clock();

```

```

while (true) {
    //超时挥手失败!
    if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
        return false;
    }
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
    // 将FIN设置为1, 表示要d断开连接
    SendData.FIN = 1;
    //随机设定seq(1到100)
    SendData.seq = (rand() % 100) + 1;
    //计算校验和
    send_calculate_Checksum(SendData);
    if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
        (struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误! " << endl;
    }
    //将要接受的数据清空
    memset(&ReceiveData, 0, sizeof(Datagram));
    //定时器启动
    clock_t start = clock();
    bool istimeout = true;
    while (clock() - start <= TIMEOUT) {
        // 接收数据
        if (recvfrom(clientsocket, (char*)&ReceiveData,
            sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
            continue;
        }
        istimeout = false;
        break;
    }
    if (!istimeout) {
        break;
    }
}
cout << "第一次挥手成功" << endl;
//计算校验和
receive_calculate_Checksum(ReceiveData);
//改为阻塞状态
if (!setBlocking(clientsocket)) {
    return false;
}
while (!(ReceiveData.ACK == 1 && ReceiveData.ack == SendData.seq + 1 &&
    ReceiveData.checksum == 0)) {
    //超时挥手失败!
    if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
        return false;
    }
    //将要接受的数据清空
    memset(&ReceiveData, 0, sizeof(Datagram));
    recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
        (struct sockaddr*)&routerAddr, &routerAddrLen);
    //计算校验和
    receive_calculate_Checksum(ReceiveData);
}
cout << "第二次挥手成功! " << endl;
//将要接受的数据清空
memset(&ReceiveData, 0, sizeof(Datagram));

```

```

    while (!(ReceiveData.ACK == 1 && ReceiveData.FIN == 1 && ReceiveData.ack
== SendData.seq + 1 && ReceiveData.checksum == 0)) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        if (recvfrom(clientsocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen) == SOCKET_ERROR) {
            continue;
        }
        //计算校验和
        receive_calculate_checksum(ReceiveData);
    }
    cout << "第三次挥手成功" << endl;
    //改为非阻塞状态
    if (!setNonBlocking(clientsocket)) {
        return false;
    }
    while (true) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        SendData.ACK = 1;
        SendData.seq = ReceiveData.ack;
        SendData.ack = ReceiveData.seq + 1;
        //计算校验和
        send_calculate_checksum(SendData);
        if (sendto(clientsocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(clientsocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            break;
        }
    }
    cout << "第四次挥手成功" << endl;
    cout << "第五次挥手成功" << endl;
    return true;
}

```

- 服务器（接收端）：（只粘贴关键代码）

```
//被动挥手
bool passive_wakehands(SOCKET& serversocket, sockaddr_in& routerAddr, int
routerAddrLen) {
    clock_t total_time = clock();
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }
    while (true) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        // 接收数据
        if (recvfrom(serversocket, (char*)&ReceiveData, sizeof(Datagram), 0,
        (struct sockaddr*)&routerAddr, &routerAddrLen) == SOCKET_ERROR) {
            continue;
        }
        //计算校验和
        receive_calculate_Checksum(ReceiveData);
        //打印接受的数据报
        cout << "接收数据报: ";
        printDatagram(ReceiveData);
        if (!(ReceiveData.checksum == 0 && ReceiveData.FIN == 1)) {
            continue;
        }
        break;
    }
    cout << "第一次挥手成功" << endl;
    //设置为非阻塞状态
    if (!setNonBlocking(serversocket)) {
        return false;
    }
    //将要发送的数据清空
    memset(&SendData, 0, sizeof(Datagram));
    SendData.ACK = 1;
    SendData.seq = (rand() % 100) + 1;
    SendData.ack = ReceiveData.seq + 1;
    //计算校验和
    send_calculate_Checksum(SendData);
    if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    cout << "第二次挥手成功" << endl;
    while (true) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要发送的数据清空
        memset(&SendData, 0, sizeof(Datagram));
        SendData.ACK = 1;
```

```

        SendData.FIN = 1;
        SendData.seq = (rand() % 100) + 1;
        SendData.ack = ReceiveData.seq + 1;
        //计算校验和
        send_calculate_Checksum(SendData);
        if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
            cerr << "发送错误!" << endl;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        //定时器启动
        clock_t start = clock();
        bool istimeout = true;
        while (clock() - start <= TIMEOUT) {
            // 接收数据
            if (recvfrom(serversocket, (char*)&ReceiveData,
sizeof(Datagram), 0, (struct sockaddr*)&routerAddr, &routerAddrLen) <= 0) {
                continue;
            }
            istimeout = false;
            break;
        }
        if (!istimeout) {
            break;
        }
    }
    cout << "第三次挥手成功" << endl;
    //计算校验和
    receive_calculate_Checksum(ReceiveData);
    //设置为阻塞状态
    if (!setBlocking(serversocket)) {
        return false;
    }
    while (!(ReceiveData.ACK == 1 && ReceiveData.seq == SendData.ack &&
ReceiveData.ack == SendData.seq + 1 && ReceiveData.checksum == 0)) {
        //超时挥手失败!
        if ((clock() - total_time) / CLOCKS_PER_SEC > maxtime) {
            return false;
        }
        //将要接受的数据清空
        memset(&ReceiveData, 0, sizeof(Datagram));
        recvfrom(serversocket, (char*)&ReceiveData, sizeof(Datagram), 0,
(struct sockaddr*)&routerAddr, &routerAddrLen);
        //计算校验和
        receive_calculate_Checksum(ReceiveData);
    }
    cout << "第四次挥手成功" << endl;
    //随便发送个消息,告诉他挥手结束
    if (sendto(serversocket, (char*)&SendData, sizeof(Datagram), 0, (struct
sockaddr*)&routerAddr, routerAddrLen) == SOCKET_ERROR) {
        cerr << "发送错误!" << endl;
    }
    cout << "第五次挥手成功" << endl;
    return true;
}

```

四、实验结果：

1.四次握手建立连接:

①发送端:

```
发送数据报: Seq: 95  ACK: 0  FIN: 0
  校验和: 65439  数据长度: 0
第一次握手成功
接收数据报: Seq: 37  ACK: 1  FIN: 0
  校验和: 0  数据长度: 0
第二次握手成功!
发送数据报: Seq: 96  ACK: 1  FIN: 0
  校验和: 65400  数据长度: 0
第三次握手成功
第四次握手成功
连接成功!
请输入你想传送文件的数量:
|
```

②接收端:

```
UDP server is listening on port 8080...
接收数据报: Seq: 95  ACK: 0  FIN: 0
  校验和: 0  数据长度: 0
第一次握手成功
发送数据包: Seq: 37  ACK: 1  FIN: 0
  校验和: 65400  数据长度: 0
第二次握手成功
接收数据报: Seq: 96  ACK: 1  FIN: 0
  校验和: 0  数据长度: 0
第三次握手成功
第四次握手成功
连接成功!
```

2.文件传输:

图片1:

发送端开始发送:

```
请输入你想传送文件的数量:
4
发送数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 65531 数据长度: 0
发送数据报: Seq: 0 ACK: 0 FIN: -1
校验和: 0 数据长度: 0
请输入你想传送的第1个文件:
1
发送数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 65534 数据长度: 0
发送数据报: Seq: 0 ACK: 0 FIN: -1
校验和: 0 数据长度: 0
现在开始发送 1.jpg
发送数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 47914 数据长度: 10240
```

接收端开始接收:

```
将接收4个文件
现在接收第1个文件
接收数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 0 数据长度: 0
发送数据包: Seq: 0 ACK: 1 FIN: 0
校验和: 65534 数据长度: 0
接收数据报: Seq: 0 ACK: 0 FIN: -1
校验和: 0 数据长度: 0
接受的文件为文件1
开始接收 ./1.jpg
接收数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 0 数据长度: 10240
发送数据报: Seq: 0 ACK: 0 FIN: 0
校验和: 65535 数据长度: 0
接收数据报: Seq: 1 ACK: 0 FIN: 0
校验和: 0 数据长度: 10240
```

数据传输过程中, 窗口边界及其大小变化: (窗口大小为33, 序号空间大小为100)

①刚开始发送时, 窗口的变化


```
现在开始发送 1.jpg
发送数据报: Seq: 0    ACK: 0    FIN: 0
    校验和: 47914    数据长度: 10240
发送窗口大小为: 1
发送窗口的下沿为:0
发送窗口的上沿为:0
发送数据报: Seq: 1    ACK: 0    FIN: 0
    校验和: 57230    数据长度: 10240
发送窗口大小为: 2
发送窗口的下沿为:0
发送窗口的上沿为:1
发送数据报: Seq: 2    ACK: 0    FIN: 0
    校验和: 43515    数据长度: 10240
发送窗口大小为: 3
发送窗口的下沿为:0
发送窗口的上沿为:2
发送数据报: Seq: 3    ACK: 0    FIN: 0
    校验和: 35123    数据长度: 10240
发送窗口大小为: 4
发送窗口的下沿为:0
发送窗口的上沿为:3
发送数据报: Seq: 4    ACK: 0    FIN: 0
    校验和: 16189    数据长度: 10240
发送窗口大小为: 5
发送窗口的下沿为:0
发送窗口的上沿为:4
```

②发送窗口满了后的变化

```

发送窗口大小为: 31
发送窗口的下沿为:0
发送窗口的上沿为:30
发送数据报: Seq: 31 ACK: 0 FIN: 0
校验和: 34511 数据长度: 10240
发送窗口大小为: 32
发送窗口的下沿为:0
发送窗口的上沿为:31
发送数据报: Seq: 32 ACK: 0 FIN: 0
校验和: 2223 数据长度: 10240
发送窗口大小为: 33
发送窗口的下沿为:0
发送窗口的上沿为:32
接收数据报: Seq: 0 ACK: 2 FIN: 0
校验和: 0 数据长度: 0
发送数据报: Seq: 33 ACK: 0 FIN: 0
校验和: 40821 数据长度: 10240
发送窗口大小为: 33
发送窗口的下沿为:1
发送窗口的上沿为:33
接收数据报: Seq: 0 ACK: 3 FIN: 0
校验和: 0 数据长度: 0
发送数据报: Seq: 34 ACK: 0 FIN: 0
校验和: 41849 数据长度: 10240
发送窗口大小为: 33
发送窗口的下沿为:2
发送窗口的上沿为:34

```

发送过程中，分组丢失情况：

```

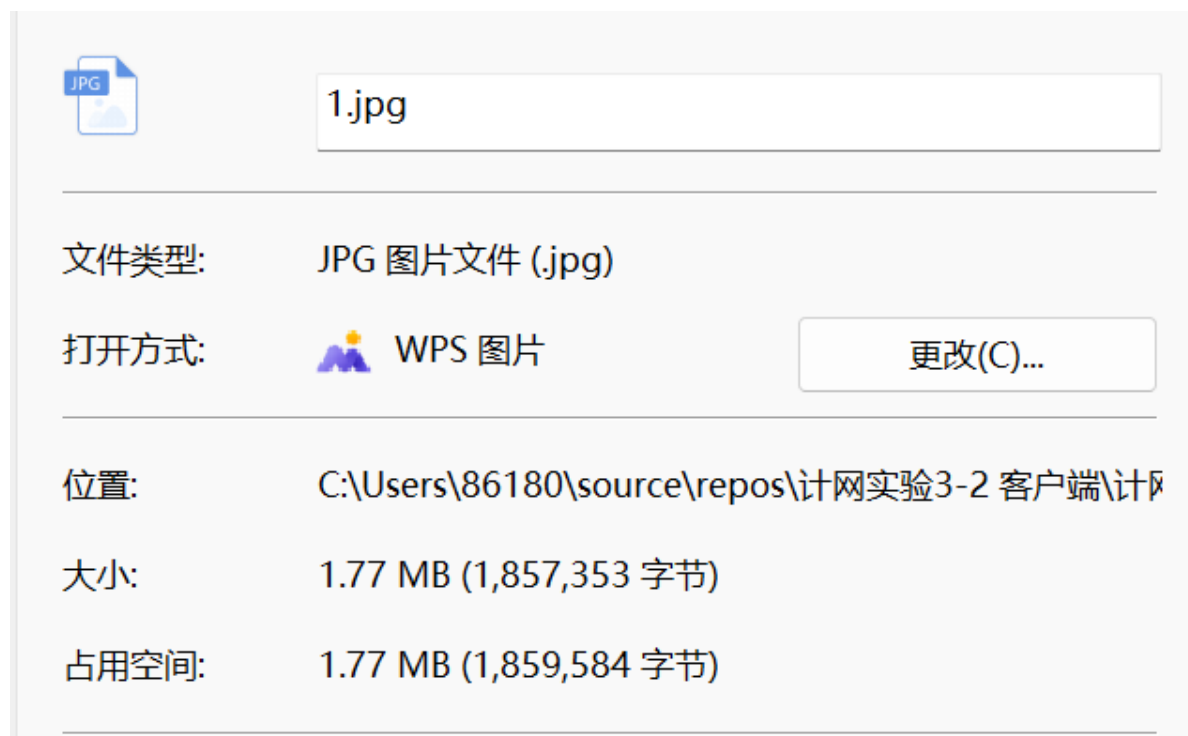
接收数据报: Seq: 0 ACK: 15 FIN: 0
校验和: 0 数据长度: 0
接收数据报: Seq: 0 ACK: 15 FIN: 0
校验和: 0 数据长度: 0
分组16丢了!!! 重新发送分组16及其之后的分组
接收数据报: Seq: 0 ACK: 15 FIN: 0
校验和: 0 数据长度: 0
接收数据报: Seq: 0 ACK: 15 FIN: 0
校验和: 0 数据长度: 0

```

发送端发送完毕，并显示传输时间和吞吐量：

```
1.jpg传输完毕
传输文件大小为： 1813KB
传输时间为： 23.691s
吞吐量为： 76.527KB/s
发送数据报： Seq: 0   ACK: 0   FIN: -1
    校验和： 0   数据长度： 0
请输入你想传送的第2个文件：
```

图片1的字节数如下图， $1857353/1024=1813.82$ ，因为是整型变量，所以结果为1813，结果正确。



接收端接收完毕：

```
./1.jpg 文件接收完毕！
现在接收第2个文件
```

注：由于文件传输过程基本差不多，接下来只显示文件的大小、传输时间和吞吐量。

图片2：

发送端开始发送：

请输入你想传送的第2个文件：

2

发送数据报： Seq: 0 ACK: 0 FIN: 0

校验和: 65533 数据长度: 0

发送数据报： Seq: 0 ACK: 0 FIN: 0

校验和: 65533 数据长度: 0

发送数据报： Seq: 0 ACK: 0 FIN: -1

校验和: 0 数据长度: 0

接收端开始接收：

接受的文件为文件2

开始接收 ./2.jpg

接收数据报： Seq: 0 ACK: 0 FIN: 0

校验和: 0 数据长度: 10240

发送数据报： Seq: 0 ACK: 0 FIN: 0

校验和: 65535 数据长度: 0

发送端发送完毕，并显示传输时间和吞吐量：

2.jpg传输完毕

传输文件大小为： 5760KB

传输时间为： 87.348s

吞吐量为： 65.9431KB/s

发送数据报： Seq: 0 ACK: 0 FIN: -1

校验和: 0 数据长度: 0

图片2的字节数如下图， $5898505/1024=5760.26$ ，因为是整型变量，所以结果为5760，结果正确。



2.jpg

文件类型: JPG 图片文件 (.jpg)

打开方式:  WPS 图片

更改(C)...

位置: C:\Users\86180\source\repos\计网实验3-2 客户端\计网

大小: 5.62 MB (5,898,505 字节)

占用空间: 5.62 MB (5,902,336 字节)

接收端接收完毕:

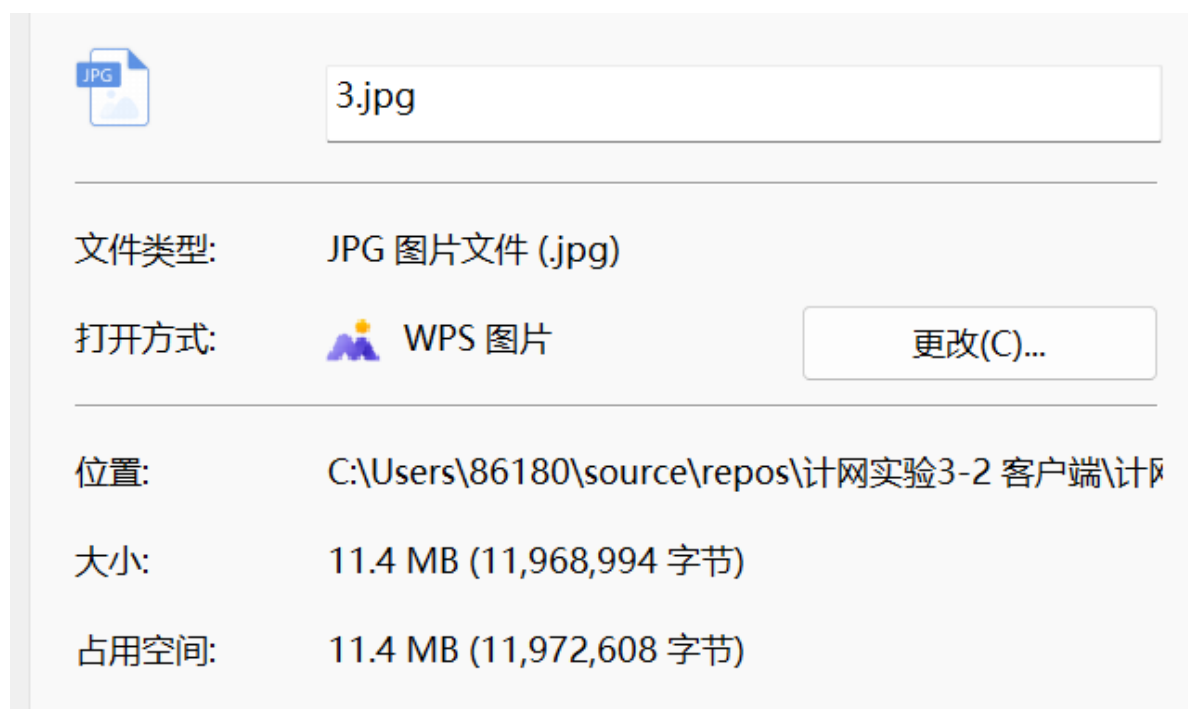
```
./2.jpg 文件接收完毕!  
现在接收第3个文件
```

图片3:

发送端发送完毕，并显示传输时间和吞吐量:

```
3.jpg传输完毕  
传输文件大小为: 11688KB  
传输时间为: 180.707s  
吞吐量为: 64.6793KB/s  
发送数据报: Seq: 0 ACK: 0 FIN: -1  
校验和: 0 数据长度: 0
```

图片3的字节数如下图， $11968994/1024=11688.47$ ，因为是整型变量，所以结果为11688，结果正确。



接收端接收完毕:


```
./3.jpg 文件接收完毕!  
现在接收第4个文件
```

文档:


发送端发送完毕，并显示传输时间和吞吐量:

```
helloworld.txt传输完毕
传输文件大小为: 1617KB
传输时间为: 23.304s
吞吐量为: 69.3872KB/s
发送数据报: Seq: 0 ACK: 0 FIN: -1
校验和: 0 数据长度: 0
发送数据报: Seq: 34 ACK: 0 FIN: 1
校验和: 65500 数据长度: 0
```

文档的字节数如下图， $1655808/1024=1617$ ，正好整除，所以结果为1617，结果正确。

 helloworld.txt

文件类型: 文本文档 (.txt)

打开方式:  记事本 更改(C)...

位置: C:\Users\86180\source\repos\计网实验3-2 客户端\计网

大小: 1.57 MB (1,655,808 字节)

占用空间: 1.58 MB (1,658,880 字节)

接收端接收完毕:

```
./helloworld.txt 文件接收完毕!
```

3.路由器日志:

Router

路由器IP:

127 . 0 . 0 . 1

服务器IP:

127 . 0 . 0 . 1

端口:

8081

服务器端口:

8080

丢包率:

5

%

延时:

5

ms

确定

修改

日志

count:17.

Delay 5 ms.

count:18.

Delay 5 ms.

count:19.

Delay 5 ms.

Miss a packet.

Delay 5 ms.

count:1.

Delay 5 ms.

count:2

4.和客户端的文件对比:

名称	修改日期	类型	大小
x64	2023/11/7 18:10	文件夹	
1.jpg	2023/11/1 11:35	JPG 图片文件	1,814 KB
2.jpg	2023/11/1 11:35	JPG 图片文件	5,761 KB
3.jpg	2023/11/1 11:35	JPG 图片文件	11,689 KB
helloworld.txt	2023/11/1 11:35	文本文档	1,617 KB
计网实验3-1 客户端.cpp	2023/11/12 15:55	C++ Source	16 KB
计网实验3-1 客户端.vcxproj	2023/11/8 18:52	VC++ Project	7 KB
计网实验3-1 客户端.vcxproj.filters	2023/11/8 18:52	VC++ Project Filter...	1 KB
计网实验3-1 客户端.vcxproj.user	2023/11/7 17:18	Per-User Project O...	1 KB

x64	2023/11/22 20:54	文件夹	
计网实验3-2 服务器.cpp	2023/11/23 18:34	C++ Source	18 KB
计网实验3-2 服务器.vcxproj	2023/11/18 17:13	VC++ Project	7 KB
计网实验3-2 服务器.vcxproj.filters	2023/11/18 17:13	VC++ Project Filter...	1 KB
计网实验3-2 服务器.vcxproj.user	2023/11/18 16:56	Per-User Project O...	1 KB
1.jpg	2023/11/24 18:02	JPG 图片文件	1,814 KB
2.jpg	2023/11/24 18:13	JPG 图片文件	5,761 KB
3.jpg	2023/11/24 18:28	JPG 图片文件	11,689 KB
helloworld.txt	2023/11/24 18:30	文本文档	1,617 KB

文件大小一致，且可以正常打开

5.五次挥手断开连接：

发送端：

```

发送数据报： Seq: 34  ACK: 0  FIN: 1
  校验和： 65500  数据长度： 0
第一次挥手成功
接收数据报： Seq: 36  ACK: 1  FIN: 0
  校验和： 0  数据长度： 0
第二次挥手成功！
接收数据报： Seq: 26  ACK: 1  FIN: 1
  校验和： 0  数据长度： 0
第三次挥手成功
发送数据报： Seq: 35  ACK: 1  FIN: 0
  校验和： 65472  数据长度： 0
第四次挥手成功
第五次挥手成功
已断开连接
请按任意键继续...

```

接收端：


```
接收数据报: Seq: 34 ACK: 0 FIN: 1
 校验和: 0 数据长度: 0
第一次挥手成功
发送数据包: Seq: 36 ACK: 1 FIN: 0
 校验和: 65463 数据长度: 0
第二次挥手成功
发送数据包: Seq: 26 ACK: 1 FIN: 1
 校验和: 65472 数据长度: 0
第三次挥手成功
接收数据报: Seq: 35 ACK: 1 FIN: 0
 校验和: 0 数据长度: 0
第四次挥手成功
第五次挥手成功
已断开连接
请按任意键继续. . .
```

五、实验总结

- 1.本次实验为了测试方便，**新增**通过输入的方式来选择想传的文件数量和文件序号的功能。客户端会把想传文件的数量和文件序号发给服务器。（由于不是关键功能，未在实验报告附上代码，详见.cpp文件）
- 2.本次实验让我更加了解了recvfrom()函数机制，以及体会了发送过快而接收较慢的情况。recvfrom()函数是从缓冲区读取数据，如果发送过快，会造成缓冲区溢出，导致数据丢失，所以需要协调双方的发送和接收速度。
- 3.ACK延迟确认机制：接收方在收到数据后，并不会立即回复ACK，而是延迟一定时间。一般ACK延迟发送的时间为**500ms**，这样做的目的是
 - ①ACK是可以合并的，也就是指如果连续收到两个TCP包，并不一定需要ACK两次，只要回复最终的ACK就可以了，可以降低网络流量。
 - ②如果接收方有数据要发送，那么就会在发送数据的TCP数据包里，带上ACK信息。这样做，可以避免大量的ACK以一个单独的TCP包发送，减少了网络流量。
- 4.累计确认这个概念是**针对发送方的**！