南開大學

《计算机网络》课程实验报告

实验 3-1

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2112060

姓　　名　孙蕗

班　　级　信息安全 1 班

一、实验要求

1. 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、接收确认、超时重传等。流量控制采用停等机制，完成给定测试文件的传输。

➢ 数据报套接字：UDP

➢ 协议设计：数据包格式，发送端和接收端交互，详细完整

➢ 建立连接、断开连接：类似 TCP 的握手、挥手功能

➢ 差错检验：校验和

➢ 接收确认、超时重传：rdt2.0、rdt2.1、rdt2.2、rtd3.0 等，亦可自行设计协议

➢ 单向传输：发送端、接收端

➢ 日志输出：收到/发送数据包的序号、ACK、校验和等，传输时间与吞吐率

➢ 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

2. 路由器程序（该程序类似于一个代理服务器）主体为一个转发线程，线程不停获取发向路由器的数据包，通过 IP 地址和端口号判断是客户端还是服务器端发来的包，若为客户端发来的包，则进行丢包、延时处理后发向服务器端；若为服务器端发来的包，则不进行处理，直接转发给客户端。

（1） 客户端发出的包，目的 IP，端口号请设置为路由器的 IP 和端口号（即在程序界面中设置的）；

（2） 服务器端发出的包，目的 IP，端口号请设置为路由器的 IP 和端口号（即在程序界面中设置的）；

丢包率为 double 类型，延时为 int 类型，单位为 ms，设置好丢包率后，日志中会打印出参数 misscount，“miscount:n”意味着每过 n 个包丢一个包，每次丢包或延时均会打印日志。

二、程序设计

1. 数据报套接字

根据实验要求数据报套接字采用 UDP。

UDP 是无连接、不可靠、面向数据报的传输层通信协议。不需要建立连接，发送端和接收端不需要维护连接状态。无拥塞控制，数据报头部短，额外开销小，具有简单、传输效率较高的特点。

UDP 的首部有 4 个字段，每个字段由两个字节构成。其中长度字段主要是记录 UDP 报文段中的字节数（首部+数据段），检验和字段的作用是接收方收到 UDP 报文后检查该报文段中是否出现了差错。

2. 协议设计

```
// 定义消息结构体
typedef struct Message
{
    WORD source_port = 0, dest_port = 0;    //源端口和目标端口
    DWORD seq_num = 0, ack_num = 0;         //序列号和确认号
    WORD  length = 0;                      //长度
    BYTE   flag = 0;                        //标志位
    WORD checksum = 0;                     //校验和
    char msg[Max_Size] = { 0 };// 消息内容
}msg;
```

数据包格式：含有 2 字节的源端口，2 字节的目标端口，4 字节的 seq，4 字节的 ack，2 字节的长度，2 字节的校验和，1 字节的标志，若干字节数据。长度为传输报文时，记录当前报文所传输数据的有效字节数。

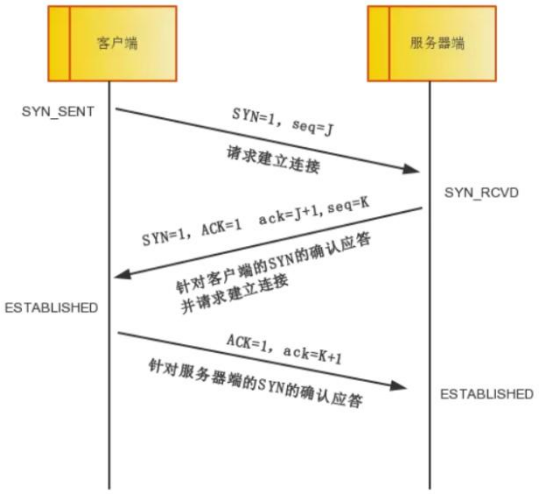
标志位 flag：从低到高分别是 ACK、SYN、FIN、END。END 为传输文件结束后发送报文的标志位。ACK=0x01，SYN=0x02，FIN=0x04，END=0x08。

```cpp
// 判断是否是ACK
bool isACK(msg* msg)
{
    return msg->flag & 0x01;
}
// 判断是否是SYN
bool isSYN(msg* msg)
{
    return msg->flag & 0x02;
}
// 判断是否是FIN
bool isFIN(msg* msg)
{
    return msg->flag & 0x04;
}
// 判断是否是END
bool isEND(msg* msg)
{
    return msg->flag & 0x08;
}
```

```cpp
// 设置ACK标志位
void setACK(msg* msg)
{
    msg->flag |= 0x01;
}
// 设置SYN标志位
void setSYN(msg* msg)
{
    msg->flag |= 0x02;
}
// 设置FIN标志位
void setFIN(msg* msg)
{
    msg->flag |= 0x04;
}
// 设置END标志位
void setEND(msg* msg)
{
    msg->flag |= 0x08;
}
```

3. **建立连接断开连接（类似 TCP 的握手和挥手）**

（1） 三次握手：



A. 客户端：

a. 客户进程创建传输控制块 TCB，发送第一次握手同步报文，标记位为 SYN=1,seq=0,ack=0，客户端进程进入了 SYN_SENT 同步已发送状态。客户端向服务器端发送报文，证明客户端的发送能力正常

```cpp
// 第一次握手：Client发送SYN
setSYN(sed);        //设置SYN
sed->seq_num = 0;   //设置seq=0
sed->ack_num = 0;   //设置ack=0
sed->source_port = port_client;//源端口
sed->dest_port = port_server;//目的端口
setChecksum(sed, &ph);  //设置校验和

//将SYN数据包发送到服务器
sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
cout << "Client: Sent SYN (First Handshake)" << endl;
```

b. 开始计时，接收服务器第二次握手报文，若超时则重传同步报文。

```
// 第二次握手：Client接收SYN, ACK
while (recvfrom(sockClient, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len) <= 0)
{
    //如果在指定的时间内（MAX_TIME）未收到响应，则重新传输SYN数据包
    if (clock() - start >= MAX_TIME)
    {
        //超时重传
        sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
        start = clock();
    }
}
cout << "Client: Received SYN, ACK (Second Handshake)" << endl;
```

```
//检查接收到的数据包是否是有效的SYN-ACK数据包
if (isACK(rec) && isSYN(rec) && verfiyChecksum(rec, &ph))
{
    //接收成功
    cout << "Client: Received packet (SYN, ACK) validation successful!(Second Shake)" << endl;
    //如果数据包有效，则准备一个ACK数据包并发送以确认服务器
    memset(sendBuffer, 0, sizeof(msg));
    setACK(sed);    //设置ack
```

c. 判断接收的报文是否为标志位（SYN，ACK），seq=0，ack=1。若是,发送第三次握手的报文,向服务器确认:标志位 ACK=1,seq=1,ack=1,客户端进入 ESTABLISHED 已建立连接状态，否则返回退出。第三次握手中客户端向服务器发送报文，服务器确认客户端的接收能力正常。

```
//第三次握手
sed->seq_num = 1;
sed->ack_num = 1;
sed->source_port = port_client;
sed->dest_port = port_server;

setChecksum(sed, &ph);
// 第三次握手：Client发送ACK
sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
cout << "Client: Sent ACK (Third Handshake)" << endl;
```

B. 服务器端：

a. 阻塞，接收客户端第一次握手报文，如果是 SYN=1,seq=0,ack=0。则发送报文:标志位（SYN=1，ACK=1），seq=0，ack=1；否则循环继续等待接收报文。

```
//第一次握手接收SYN
recvfrom(sockServer, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, &len);

if (isSyn(rec) && vertifyChecksum(rec, &ph) && rec->seq_num == 0)
{
    cout << "Server: Received packet (SYN) validation successful! (First Shake)" << endl;
```

```
else
{
    cout << "Server: Received packet (SYN) validation failed." << endl;
    continue;
}
```

b. 发送第二次握手报文。非阻塞，开始计时，接收客户端第三次握手报文,若超时则重传第二次握手报文,服务器进程进入了 SYN_RCVD 同步收到状态。第二次握手中，服务器端接收到报文并向客户端发送报文，证明服务器端的接收能力、发送能力正常，客户端得出客户端发送接收能力正常，但此时服务器不能确认客户端的接收能力有没有问题。

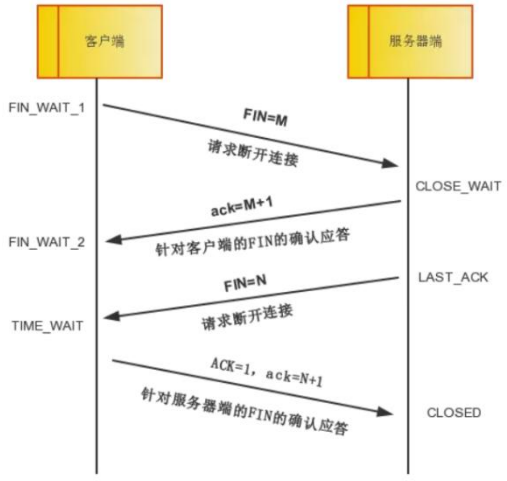
```
//第二次握手设置SYN，ACK报文
setAck(sed);
setSyn(sed);
sed->seq_num = 0;
sed->ack_num = 1;
sed->source_port = port_server;
sed->dest_port = port_client;
setChecksum(sed, &ph);

//发送SYN，ACK
sendto(sockServer, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, len);
cout << "Server: Send packet (SYN,ACK) (Second Shake)" << endl;
break;
```

```
//设为非阻塞
u_long imode = 1;
ioctlsocket(sockServer, FIONBIO, &imode);//非阻塞

clock_t start = clock(); //开始计时
while (recvfrom(sockServer, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, &len) <= 0)
{
    if (clock() - start >= MAX_TIME)
    {
        //超时重传
        sendto(sockServer, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, len);
        cout << "Server: Resend packet (SYN,ACK) (Second Shake) ..." << endl;
        start = clock();
    }
}
```

c. 判断接收的客户端第三次握手报文是否为：标志位（ACK=1），seq=1，ack=1。若是，连接成功。否则返回退出。

```
//第三次握手
if (isAck(rec) && vertifyChecksum(rec, &ph))
{
    cout << "Server: Received packet (ACK) validation successful! (Third Shake)" << endl;
}
else
{
    return false;
}
```

（2） 四次挥手



A. 客户端：

a. 发送结束报文，第一次挥手标记位为 FIN，客户端进入 FIN_WAIT_1（终止等待 1）状态。

```
//第一次挥手，客户端发送FIN报文
setFIN(sed);        //设置Fin
sed->seq_num = 0;   //设置seq=0
sed->ack_num = 0;   //设置ack=0
sed->source_port = port_client;
sed->dest_port = port_server;
setChecksum(sed, &ph);  //设置校验和

//发送
sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
cout << "Client: Sent packet (FIN) (First wave)" << endl;
```

b. 开始计时，接收服务器第二次挥手报文，若超时则重传第一次挥手报文。

```
//第二次挥手，接收确定服务器端的FIN，ACK报文
while (recvfrom(sockClient, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len) <= 0) {

    // over time
    if (clock() - start >= MAX_TIME)
    {
        //超时重传
        sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
        cout << "Client: Sent packet (FIN), retransmission" << endl;
        start = clock();
    }
}
```

c. 判断接收的服务器端的第二次挥手报文是否为：标志位（FIN，ACK）。否则继续接收。客户端接收到服务器端的确认请求 ACK=1 后，客户端就会进入 FIN_WAIT_2（终止等待 2）状态，等待服务器发送连接释放报文。

```
if (isACK(rec) && isFIN(rec) && verfiyChecksum(rec, &ph))
{
    cout << "Client: Received packet (FIN, ACK), verification successful.(Second wave)" << endl;
}
else
{
    return false;
}
```

d. 阻塞，接收第三次挥手报文，判断是否为：标志位（FIN）。若是，发送第四次挥手报文，标志位（FIN，ACK）。否则继续接收。

```
    //第三次挥手，接收确定服务器端的FIN报文

    while (true)
    {
        recvfrom(sockClient, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len);

        if (isFIN(rec) && verfiyChecksum(rec, &ph))
        {
            cout << "Client: Received packet (FIN), verification successful.(Third wave)" << endl;
            break;
        }
    }
```

```
//第四次挥手，发送FIN，ACK报文
cleanflag(sed);
setFIN(sed);
setACK(sed);
setChecksum(sed, &ph);  //设置校验和
sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
cout << "Client: Sent packet (FIN, ACK) (Fourth wave)" << endl;
```

e. 客户端就进入了 TIME_WAIT（时间等待）状态，但此时 TCP 连接还未终止，经过 2MSL 后（最长报文寿命），当客户端撤销相应的 TCB 后，客户端才会进入 CLOSED 关闭状态。等待 2MSL，如果在 2MSL 中收到服务器的 FIN 报文，重传第四次挥手确定报文。

```
//等待2MSL
start = clock(); //开始计时

while (clock() - start <= 2 * MAX_TIME)
{
    //if(clock() - start>= MAX_TIME)
    if (recvfrom(sockClient, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len) > 0 && isFIN(rec) && verfiyChecksum(rec, &ph)) {
        sendto(sockClient, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
        cout << "Client: Sent packet (FIN, ACK), retransmission" << endl;
    }
}
cout << "Client: Connection closed" << endl;
closesocket(sockClient);
return true;
```

f. 返回退出。

B. 服务器：

a. 阻塞，接收客户端第一次挥手报文，如果是 FIN，发送第二次挥手报文标志位（FIN，ACK），服务端就进入了 CLOSE_WAIT 关闭等待状态。若服务器端还有数据要发送给客户端，客户端还会接受，服务器端会持续一段时间。否则循环继续等待接收第一次挥手报文。

```
//第一次挥手，接收验证客户端的FIN报文
while (1)
{
    recvfrom(sockServer, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len);

    if (isFin(rec) && vertifyChecksum(rec, &ph))
    {
        cout << "Server: Received packet(FIN),vertification is successful.(First wave)" << endl;
        break;
    }
}
```

```
//第二次挥手：服务器设置发送ACK，FIN报文

setFin(sed);        //设置Fin
setAck(sed);        //设置Ack
sed->seq_num = 0;   //设置seq=0
sed->ack_num = 0;   //设置ack=0
sed->source_port = port_client;
sed->dest_port = port_server;
setChecksum(sed, &ph);  //设置校验和

//发送
sendto(sockServer, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, len);
cout << "Server:Sent packet (FIN, ACK), verification successful.(Second wave)" << endl;
```

b. 若无要传输的数据，发送第三次挥手 FIN 报文。服务器将最后的数据发送完毕后，就向客户端发送连接释放报文 FIN=1，服务器就进入了 LAST_ACK（最后确认）状态。

```
//第三次挥手发送FIN
cleanflag(sed);
setFin(sed);        //设置Fin
setChecksum(sed, &ph);
sendto(sockServer, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, len);
cout << "Server:Sent packet (FIN), verification successful.(Third wave)" << endl;
```

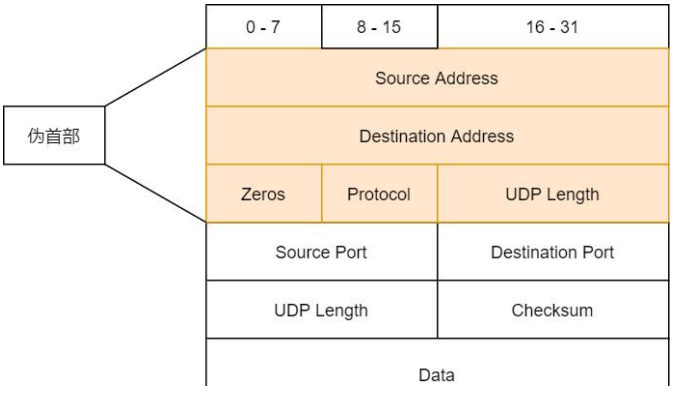
c. 非阻塞，超时重传第三次挥手 FIN 报文。判断接收的第四次挥手报文是否为(FIN,ACK)，如是则马上断开连接，立即进入 CLOSED 关闭状态。

```
//第三次挥手接收
clock_t  start = clock(); //开始计时

while (recvfrom(sockServer, recBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len) <= 0 || !(isAck(rec) && isFin(rec) && vertifyChecksum(rec, &ph)))
{
    if (clock() - start >= MAX_TIME)
    {
        //超时重传
        sendto(sockServer, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, len);
        cout << "Server: Sent packet (FIN), retransmission" << endl;
        start = clock();
    }
}
```

```
//第四次挥手
if (isAck(rec) && isFin(rec) && vertifyChecksum(rec, &ph))
{
    cout << "Server: Received packet (FIN, ACK), verification successful.(Fourth wave)" << endl;
}
else
{
    return false;
}
cout << "Connection close..." << endl;
closesocket(sockServer);
return true;
```

4. 差错检验：校验和

UDP 数据校验和和 TCP 数据的校验和计算的方法是一致的，包括：UDP 伪首部，UDP 头部和 UDP 数据。



UDP 的伪首部：

0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源IP地址（Source IP address）			
目的IP地址（Destionation IP address）			
0	协议（Protocol）	长度（Length）	

自定义的伪首部包含 2 字节的源端口，2 字节的目标端口，2 字节长度，版本号和填充 0。

```
// 定义伪首部结构体
typedef struct PseudoHead
{
    DWORD source_ip = 0, dest_ip = 0;        //源IP和目标IP
    char zero = 0;// 预留字段，填充0
    char protocol = 0;// 协议字段
    WORD length = sizeof(msg);// 长度字段
}pseudoHead;
```

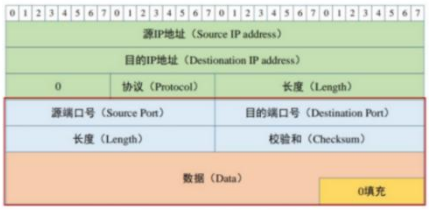
■ UDP校验和的计算方法

发送端：

- 产生伪首部，校验和域段清0，将数据报用0补齐为16位整数倍
- 将伪首部和数据报一起看成16位整数序列
- 进行 16 位二进制反码求和运算，计算结果取反写入校验和域段

接收端：

- 产生伪首部，将数据报用0补齐为16为整数倍
- 按16位整数序列，采用 16 位二进制反码求和运算
- 如果计算结果位全1，没有检测到错误；否则，说明数据报存在差错




```
// 发送消息时设置校验和
void setChecksum(msg* message, pseudoHead* ph)
{
    //消息的校验和先设为0
    message->checksum = 0;
    int sum = 0;
    int len_pseudo = sizeof(pseudoHead);
    int len_msg = sizeof(msg);

    // 计算伪首部的校验和，将伪首部的每两个字节作为16位的整数相加
    for (int i = 0; i < len_pseudo / 2; i++)
    {
        sum += ((WORD*)ph)[i];
    }
    // 计算消息的校验和，将消息的每两个字节作为16位的整数相加
    for (int i = 0; i < len_msg / 2; i++)
    {
        sum += ((WORD*)message)[i];
    }
    // 处理进位
    while (sum >> 16)
    {
        sum = (sum & 0xffff) + (sum >> 16);
    }
    // 将校验和取反并赋值给消息的校验和字段
    message->checksum = ~sum;
};
```

在发送数据时，发送端利用自己产生的伪首部和发送的 UDP 数据报计算校验和，按如下步骤计算校验和：

① 把校验和字段设置为 0；

② 伪首部和消息的每两个字节作为 16 位字进行累加。如果有进位，就将进位部分加回到 sum 的低 16 位中。

③ 把累加结果取反存入校验和字段中

```
// 接收消息时验证校验和
bool verfiyChecksum(msg* message, pseudoHead* ph)
{
    int sum = 0;
    int len_pseudo = sizeof(pseudoHead);
    int len_msg = sizeof(msg);

    // 计算伪首部的校验和
    for (int i = 0; i < len_pseudo / 2; i++)
    {
        sum += ((WORD*)ph)[i];
    }

    // 计算消息的校验和
    for (int i = 0; i < len_msg / 2; i++)
    {
        sum += ((WORD*)message)[i];
    }

    // 处理进位
    while (sum >> 16)
    {
        sum = (sum & 0xffff) + (sum >> 16);
    }

    // 验证校验和是否为0xffff
    return sum == 0xffff;
};
```

在接收数据时，接收端利用自己产生的伪首部和接收的 UDP 数据报计算校验和，按如下步骤计算检验和：

① 伪首部和消息的每两个字节作为 16 位字相加，如果累加的结果有进位，则将进位部分加到低 16 位中，直到没有进位

② 验证最终的 16 位累加结果是否为 0xFFFF。如果是，说明数据在传输过程中没有发生错误，校验和有效；否则，校验和无效，数据可能已经损坏。

5. 可靠数据传输协议 rdt

（1） rdt1.0

完全可靠的信道，可以保证从应用层的一侧到另一侧数据不丢失，因此此时的发送方和接收方只会有一个状态。

发送方： 等待应用程序下发调用指令，发送数据

① 应用进程调用 rdt_send（data），将数据推送至传输层

② 传输层调用 make_pkt，将源自于应用程序的报文分组打包成报文段

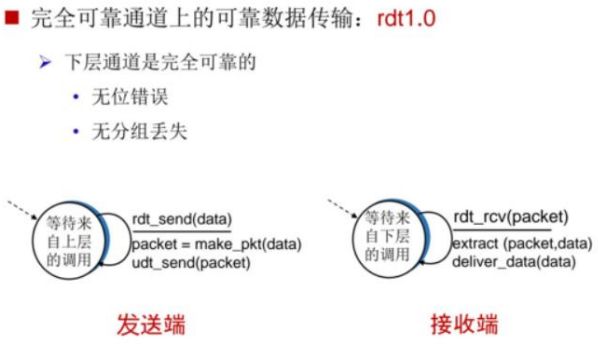
③ 传输层调用 udt_send 方法，将报文段推送至信道

接收方： 等待来自下层的调用指令，接受数据并缓存

① 下层调用 ret_rev，将数据推送到传输层

② 传输层调用 extract，从报文段中提取出数据

③ 传输层调用 deliver_data 将数据推送至应用层



（2） rdt2.0

在 rdt 2.0 考虑比特差错出现的情况，比特差错通常会出现在可能受损的物理部件之中，因此引入比特差错校验。在考虑出现比特差错的 rdt 2.0 中，需要加入肯定确认（ACK positive acknowledgement）、否定确认（NAK negative acknowledgement）的情况。

对于否定确认的报文，需要提示发送方重新发送该数据。基于这种重传机制的可靠数据传输协议称为自动重传协议（Automatic Repeat reQuest，ARQ）。Sender 需要增加一个状态，用于等待 ACK 或 NAK。

发送方：

① 应用层调用 rdt_send 方法， 将数据推送至应用层

② 应用层调用 make_pdt 将数据打包成报文段，并在报文段中封装进一个校验和

③ 应用层调用 udt_send 方法将打包完成的报文段推送至信道

④ 发送端此时状态迁移为等待 ACK 应答或者 NAK 应答状态，处于阻塞状态

接收方：

① 下层通过 rdt_rcv 方法，将数据推送到传输层

② 传输层接收到报文段，对报文段数据进行校验处理，校验成功则执行第 4 步，校验失败则跳转到第 3 步

③ 发送 NAK 指令， 继续等待下层的调用

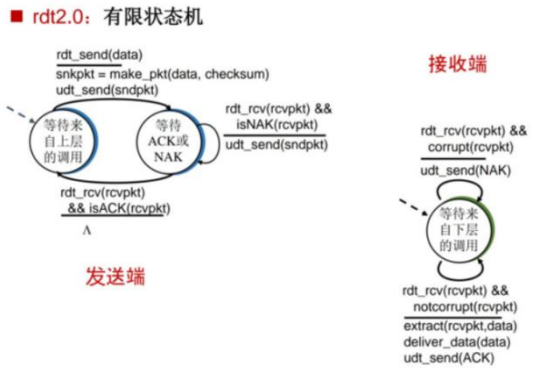
④ 发送 ACK 指令， 继续等待下层的调用

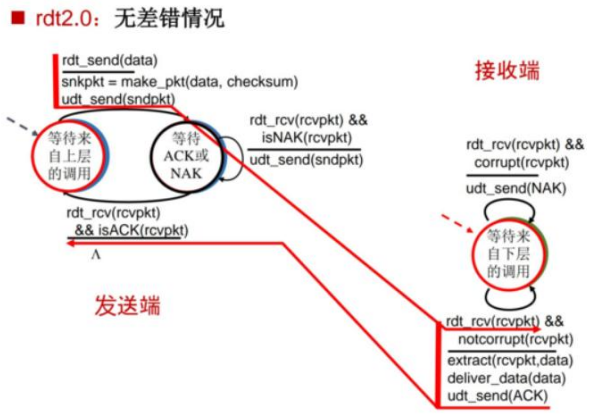
发送方：

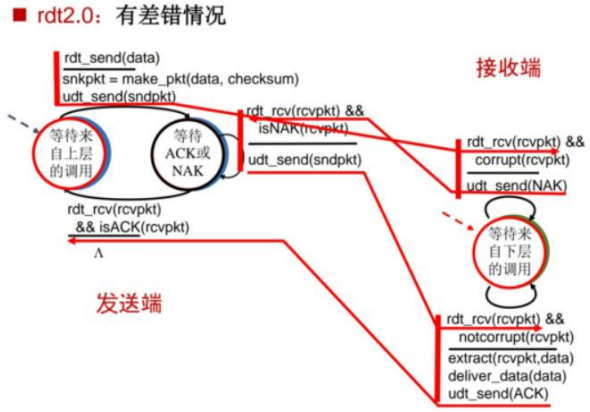
① 接收到 NAK 应答指令执行第 2 步， 接收到 ACK 执行第 3 步

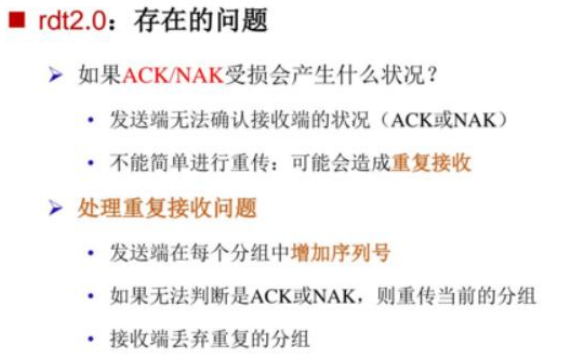
② 接收到 NAK，发送方直接将打包好的数据再一次通过 udt_send 方法推送到信道，保持等待 ACK 或 NAK 指令状态

③ 接受到 ACK，发送方不再阻塞，可以发送新的数据，状态迁移为等待上层调用状态





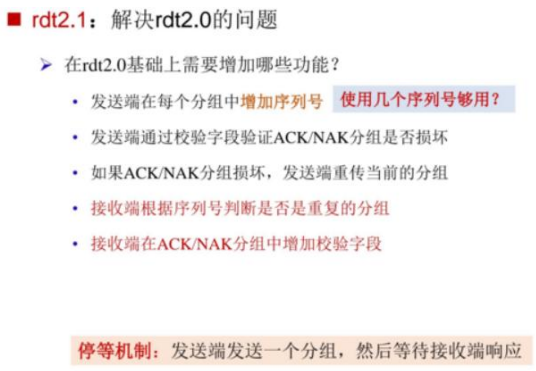




（3） rdt2.1

ACK/NAK 可能存在受损的可能性，无法保证接收端发送的 ACK/NAK 是否存在比特差错的情况。当发送方收到含糊不清的 ACK 和 NAK 分组时，只需重传当前数据分组即可。这种方法在发送方到接收方的信道中引入了冗余分组。但是接收方不知道它上次所发送的 ACK 或 NAK 是否被正确的接收到。

为了解决冗余分组的问题，在数据分组中添加一个新的字段，让发送方对其数据分组编号，即将发送数据分组的序号。



发送方：

① 应用层调用 rdt_send 方法， 将数据推送至应用层

② 应用层调用 make_pdt 将数据打包成报文段，并在报文段中封装进一个 校验码和一个值为 0 或 1 的序号

③ 应用层调用 udt_send 方法将打包完成的报文段推送至信道。

④ 发送端此时状态迁移为等待序号为 0 或 1 的报文段 ACK 应答或 NAK 应答状态。

接收端：

① 下层通过 rdt_rcv 方法， 将数据推送到传输层

② 传输层接收到报文段，对报文段数据进行校验处理，校验成功则执行第 4 步，校验失败则执行第 3 步

③ 发送 NAK 指令，继续等待下层调用

④ 检测数据序号，如果是冗余数据，直接丢弃数据，发送对缓存栈中数据的 ACK 指令。对于非冗余数据，则将数据置换到缓存栈之中，发送一个确认对本次数据的 ACK 指令，继续等待来自下层的调用。

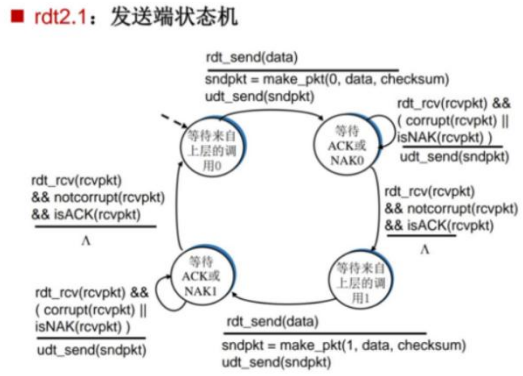
发送端：

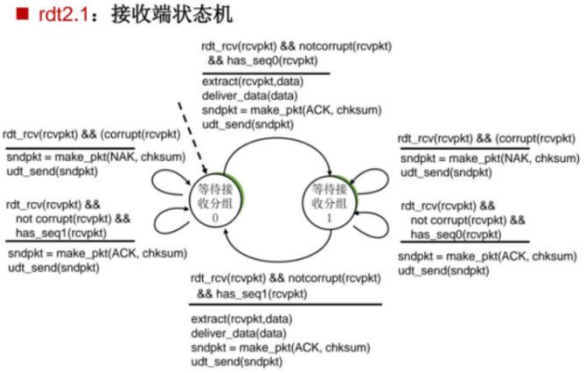
① 接收到应答指令后，进行数据校验处理，如果数据校验错误，直接重传上次数据，如果数据校验正确，则执行第 2 步。

② 判断接受到的应答指令，如果指令为 ACK 执行第 3 步， 如果指令为 NAK 执行第 4 步

③ 接收到 ACK 指令，发送端不再阻塞， 可以发送新的数据 1 或 0，状态迁移到等待来自上层调用。

④ 接收到 NAK 指令，重传上次数据，状态迁移到等待 ACK 或 NAK 状态。

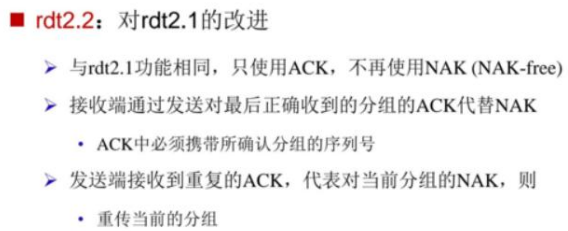




（4） rdt2.2

有了序号 seq，接收端不在需要用 NAK 指令去表示收到的数据产生了比特错误。对于比特受损的数据，接收端直接丢弃，并发送一个对缓存区数据的确认 ACK。

发送端需要维护一个缓存，用于记录上一次发送的数据，当接受到的 ACK 与缓存序列号相同，那么就表示发送的数据发生了比特差错，此时重新发送一次缓存区中的数据即可。



发送端

① 应用层调用 rdt_send 方法，将数据推送至传输层

② 应用层调用 make_pdt 将数据打包成报文段，并在报文段中封装进一个校验和，和一个值为 0 或 1 的序号，并将其存储至缓存区

③ 应用层调用 udt_send 方法将打包完成的报文段推送至信道。

④ 发送端此时状态迁移为 等待序号为 0 或 1 的报文段 的 ACK 或 NAK 应答状态

接收端

① 下层通过 rdt_rcv 方法，将数据推送到传输层

② 传输层接收到报文段，对报文段数据进行校验处理，校验成功则执行第 4 步，校验失败则执行第 3 步

③ 发送缓存区数据序号的确认 ACK 指令，保持等待下层调用的状态

④ 检测本次数据序号，如果是冗余数据，直接丢弃数据，发送缓存栈中数据序号的 ACK 指令。对于非冗余数据，将数据置换到缓存区之中，并发送一个对本次数据序号的确认 ACK 指令

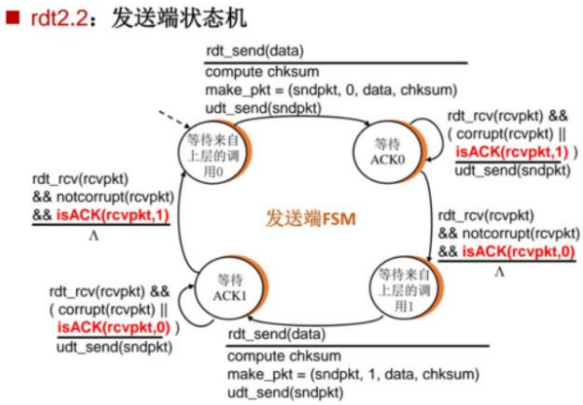
发送端：

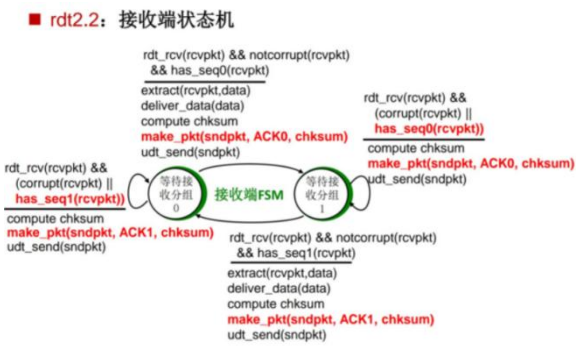
① 接收到应答指令后进行数据校验处理，如果数据校验错误， 直接重新发送上次数据，如果正确则执行第 2 步。

② 判断接受到的应答指令，如果指令为 ACK 中的序号等于缓存区的序号，执行第 3 步，如果 ACK 指令序号不等于缓存区序号则执行第 4 步

③ 重新发送上次数据，状态迁移到 等待 ACK 应答指令

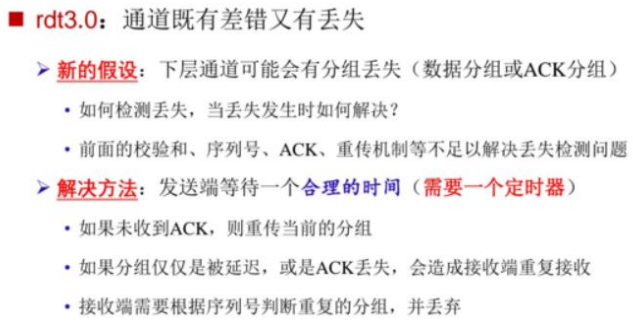
④ 发送端不再阻塞， 可以发送新的序号为 1 或者 0 的数据，状态迁移到等待来自上层调用





（5） rdt3.0

不仅有比特差错，还有丢包异常，即发送方或者接受方由于网络阻塞等状况，并没有收到来自于对方的应答数据。加入一个定时器来处理丢包现象，当发送一个报文段的时候，就开启一个定时器，在定时器结束期间，如果没有收到对应数据的应答报文，则重传数据。



发送端：

① 应用层调用 rdt_send 方法， 将数据推送至应用层

② 应用层调用 make_pdt 将数据打包成报文段，并在报文段中封装进一个校验和，和一个值为 0 或 1 的序号，并将其存储至缓存区

③ 应用层调用 udt_send 方法将打包完成的报文段推送至信道，并启动一个定时器事件。

④ 发送端此时状态迁移为等待序号为 0 或 1 报文段的 ACK 应答状态

⑤ 倘若在定时器等待时间内，没有收到响应，则重新执行第 3 步

接收端：

① 下层通过 rdt_rcv 方法，将数据推送到传输层

② 传输层接收到报文段，对报文段数据进行校验处理，校验成功则执行第 4 步，校验失败则执行第 3 步

③ 发送缓存区数据序号的确认 ACK 指令，同时开启一个定时器， 保持等待下层调用的状态。

④ 检测数据序号，如果是冗余数据，直接丢弃数据，发送缓存区中数据序号的 ACK 指令。对于非冗余数据，将数据置换到缓存区中，发送一个对该数据序号的确认 ACK 指令，同时开启一个定时器。

⑤ 倘若在定时器等待时间内，没有收到响应，则重新执行第 3 或第 4 步

发送端：

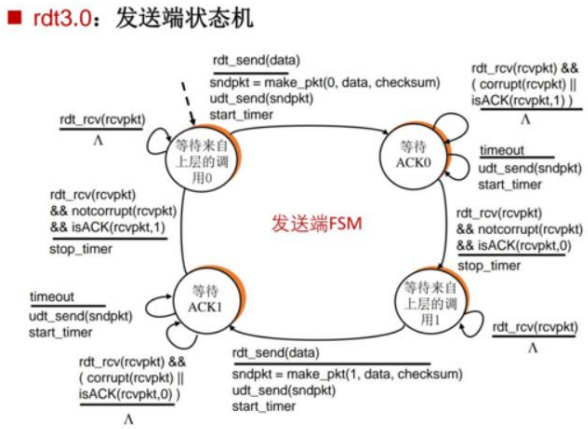
① 接收到应答指令后进行数据校验处理，如果数据校验错误，直接重新发送上次数据，如果正确则执行步第 2 步

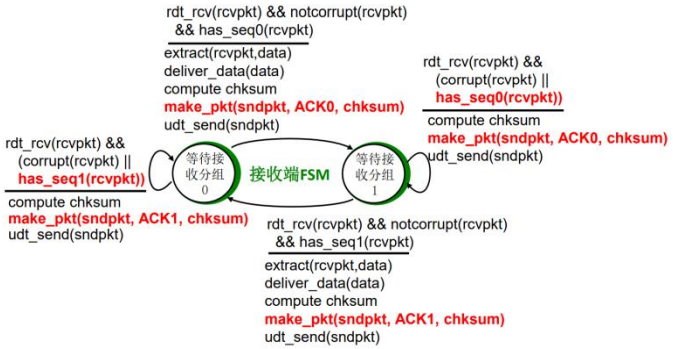
② 判断接受到的应答指令，如果指令为 ACK 中的序号等于缓存区的序号，执行第 3 步，如果 ACK 指令序号不等于缓存区序号则执行 4

③ 重新发送上次数据，状态迁移到等待 ACK，同时开启一个定时器

④ 发送端不再阻塞，可以发送新的序号值为 1 或 0 的数据，状态迁移到等待来自上层调用

⑤ 若在定时器等待时间内，没有收到响应，则重新执行第 3 或第 4 步。





（6） 本次实验的可靠数据传输协议

本次实验使用的可靠数据传输协议基于 rdt3.0。

在本次实验中，由于三次握手的时候使用了 0 和 1.所以将此处的两个 seq 号设为 2 和 3。

客户端在发送完一个 seq 号的数据后进入该 seq 号的确定状态。如果超时则重发数据包，收到另一个 seq 的确定后维持状态不变。

而服务器端在收到当前状态的正确 seq 号的时候会发送该 seq 号的 ack，转换状态。若收到的 seq 号为另一状态或损坏，则发送其状态的 ack。以希望让客户端进入发送下一预期 seq 的状态。

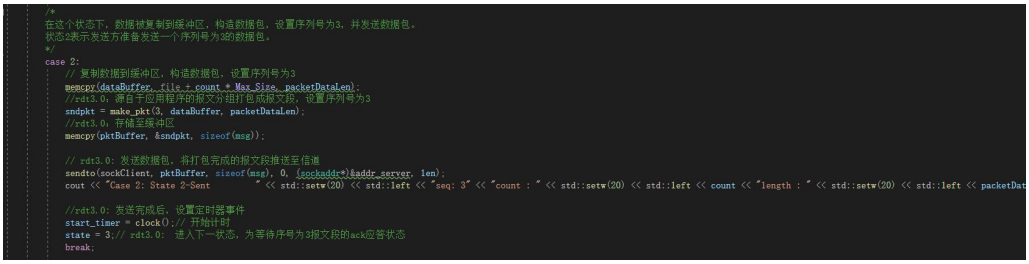
① 发送端的 4 个状态：

State 0 表示发送方准备发送一个 seq 序列号为 2 的数据包,这个状态下，数据被复制到缓冲区，构造数据包，设置 seq 为 2，并发送数据包。

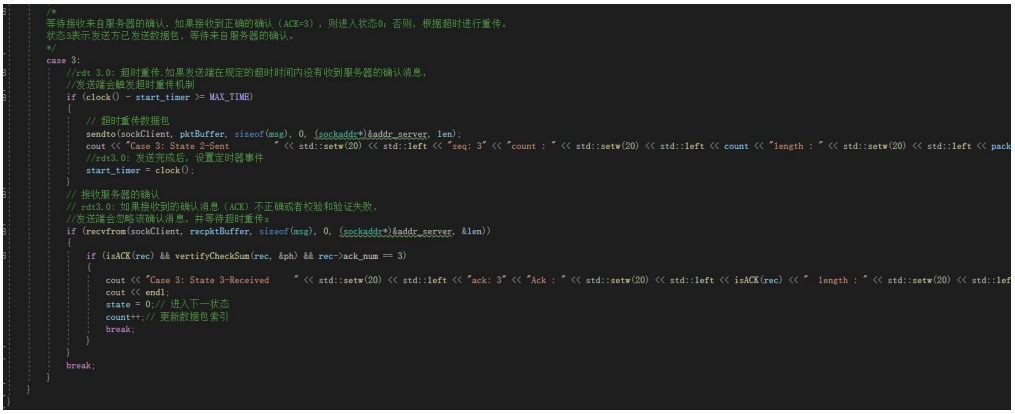
```
switch (state)
{
/*
在这个状态下，数据被复制到缓冲区，构造数据包，设置序列号为2，并发送数据包
状态0表示发送方准备发送一个序列号为2的数据包
*/
case 0:
    // 复制数据到缓冲区，构造数据包，设置序列号为2
    memcpy(dataBuffer, file + count * Max_Size, packetDataLen);

    //rdt3.0：调自于应用程序的报文分组打包成报文段，设置序列号为2
    sndpkt = make_pkt(2, dataBuffer, packetDataLen);
    //rdt3.0：存储至缓冲区
    memcpy(pktBuffer, &sndpkt, sizeof(msg));

    // rdt3.0：发送数据包，将打包完成的报文段推送至信道
    sendto(sockClient, pktBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
    //rdt3.0：发送完成后，设置定时器事件
    start_timer = clock();// 开始计时
    state = 1;// rdt3.0：进入下一状态，为等待序号为2报文段的ack应答状态
    cout << "Case 0: State 0-Sent          " << std::setw(20) << std::left << "seq: 2" << "count : " << std::setw(20) << std::left << count << "length : " << std::setw(20) << std::left << packetDat
    break;
```

State 1 表示发送端等待接收来自接收端的确认，如果接收到正确的确认（ack=2），则进入 State 2；否则，根据超时进行重传 State 0 的数据包。State 1 表示发送方已发送数据包，等待来自服务器的确认。

State 2 表示发送方准备发送一个 seq 为 3 的数据包，这个状态下，数据被复制到缓冲区，构造数据包，设置 seq 为 3，并发送数据包。



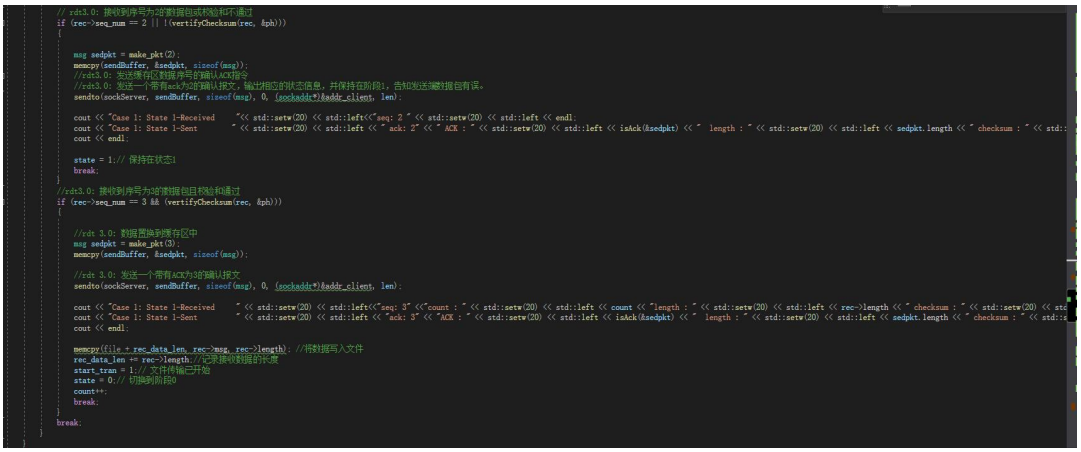
State 3 表示发送端等待接收来自接收端的确认，如果接收到正确的确认（ack=3），则进入 State 0；否则，根据超时进行重传 State 2 的数据包。State 3 表示发送方已发送数据包，等待来自服务器的确认。



② 接收端的两个状态：

State 0:接收来自客户端的数据包,State 0 预期收到的数据包的 seq=2。如果接收到 seq 为 2 的数据包且校验和正确，发送 ack 为 2 的确认，表示正确确认，进入到 State 1；如果接收到 seq 为 3 的数据包或校验和不通过，发送 ack 为 3 的确认,并保持在 State 0，告知发送端数据包有误。

State 1：接收来自客户端的数据包,State 1 预期收到的数据包的 seq=3。如果接收到 seq 为 3 的数据包且校验和通过，发送 ack 为 3 的确认，表示正确确认，进入到 State 0；如果接收到 seq 为 2 的数据包或校验和不通过，发送 ack 为 2 的确认,并保持在 State 1，告知发送端数据包有误。



6. 发送端和接收端的交互过程

（1） 发送端交互过程

a. 发送端构造数据包

```
// 构造数据包
msg make_pkt(int seq, char* data, unsigned short len)
{
    msg message;// 创建一个消息结构体
    memset(&message, 0, sizeof(msg));// 初始化消息结构体，将其内容全部设置为0
    message.source_port = port_client;// 设置消息的源端口
    message.dest_port = port_server;// 设置消息的目标端口
    message.length = len;// 设置消息的长度
    message.seq_num = seq;// 设置消息的序列号
    memcpy(message.msg, data, len);// 复制数据到消息的数据字段

    pseudoHead ph;      // 创建一个伪首部结构体
    memset(&ph, 0, sizeof(pseudoHead));// 初始化伪首部结构体，将其内容全部设置为0
    ph.source_ip = inet_addr(ip_client);// 设置伪首部的源IP地址
    ph.dest_ip = inet_addr(ip_server);// 设置伪首部的目标IP地址
    setChecksum(&message, &ph); //设置校验和

    return message;// 返回构造好的消息结构体
}
```

发送端通过调用 make_pkt 函数构造一个数据包，并填充数据字段。

b. 发送端设置校验和

客户端使用 setChecksum 函数计算并设置校验和。

c. 发送端发送数据包

```
// 发送数据包
sendto(sockClient, pktBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
```

使用 sendto 函数将数据包发送到目的地址。

d. 等待确认： 发送端等待接收到来自接收端的确认。

```
// 接收服务器的确认
if (recvfrom(sockClient, recpktBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, &len))
{
    if (isACK(rec) && verfiyChecksum(rec, &ph) && rec->ack_num == 2)
    {
```

e. 处理超时： 如果在规定的时间内未收到确认，发送端可能会触发超时重传机制，重新发送数据包。

```
if (clock() - start_timer >= MAX_TIME)
{
    // 超时重传数据包
    sendto(sockClient, pktBuffer, sizeof(msg), 0, (sockaddr*)&addr_server, len);
    cout << "State 0-Sent          " << std::setw(20) << std::left << "seq: 2" << "index
    start_timer = clock();
}
```

（2） 接收端的交互过程

a. 接收数据包： 接收端通过 recvfrom 函数接收从发送端发送过来的数据包。

```
if (recvfrom(sockServer, recpktBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, &len) > 0 && rec->length == 0)
{
    break;
}
```

b. 验证校验和：接收端使用 vertifyChecksum 函数验证接收到的数据包的校验和是否正确。

```
//接收到的数据包的序号是2且校验和通过，则正确接收
if (rec->seq_num == 2 && (vertifyChecksum(rec, &ph)))
{
```

c. 处理数据包： 如果校验和正确，接收端根据数据包的内容执行相应的操作。

```
memcpy(file + rec_data_len, rec->msg, rec->length); //将数据写入文件
rec_data_len += rec->length;//记录接收数据的长度
start_tran = 1;// 文件传输已开始
stage = 0;// 切换到阶段0
index++;
```

d. 构造确认包： 如果需要，接收端可以构造一个确认数据包，设置确认号，并发送给发送端。

```
//发送一个带有ACK为2的确认报文
sendto(sockServer, sendBuffer, sizeof(msg), 0, (sockaddr*)&addr_client, len);
cout << "State 0-Received     "<< std::setw(20) << std::left<<"seq: 2"<< "index :
cout << "State 0-Sent         " << std::setw(20) << std::left<<"ack: 2" << "ACK :
```

7. 文件读写

发送端：选择发送的文件，文件内容读入缓冲区，获得文件名，文件大小

```
while (1)
{
    char* filename = new char[100];
    memset(filename, 0, 100);
    string filedir;
    cout << endl;
    cout << "文件编号如下：  " << endl;
    cout << "1: 1.jpg" << endl;
    cout << "2: 2.jpg" << endl;
    cout << "3: 3.jpg" << endl;
    cout << "4: helloworld.txt" << endl;
    cout << "5: a.jpg" << endl;
    cout << "6: Exit" << endl;
    cout << "请输入要传输的文件：  " << endl;

    int i;
    cin >> i;
    cout << endl;

    if (i == 6)
    {
        cout << "Exit" << endl;
        break;
    }
```

```
switch (i)
{
case 1:
    filedir = "C:/Users/LENOVO/Desktop/test/1.jpg";
    memcpy(filename, "1.jpg", sizeof("1.jpg"));
    break;

case 2:
    filedir = "C:/Users/LENOVO/Desktop/test/2.jpg";
    memcpy(filename, "2.jpg", sizeof("2.jpg"));
    break;
case 3:
    filedir = "C:/Users/LENOVO/Desktop/test/3.jpg";
    memcpy(filename, "3.jpg", sizeof("3.jpg"));
    break;
case 4:
    filedir = "C:/Users/LENOVO/Desktop/test/helloworld.txt";
    memcpy(filename, "helloworld.txt", sizeof("helloworld.txt"));
    break;
case 5:
    filedir = "C:/Users/LENOVO/Desktop/test/a.jpg";
    memcpy(filename, "a.jpg", sizeof("a.jpg"));
    break;
default:
    break;
}
```

```
ifstream infile(filedir, ifstream::binary);
if (!infile.is_open())
{
    cout << "Failed to open" << endl;
    return 0;
}

infile.seekg(0, infile.end);
DWORD fileLen = infile.tellg();
infile.seekg(0, infile.beg);
```

接收端：获得文件内容和文件名，在指定路径下写入收到的文件

```
//接收文件的缓冲区
bool tran = 1;

while (tran)
{
    char* fileBuffer = new char[100000000];

    DWORD fileLength = 0;
    char* filename = new char[100];
    memset(filename, 0, 100);
```

```
    string dir = "C:/Users/LENOVO/Desktop/test/recv/";
    string fn = filename;
    string filenm = dir + fn;

    //写入复制文件
    ofstream outfile(filenm, ios::binary);
    outfile.write(fileBuffer, fileLength);
    outfile.close();

    cout << "Y/N: ";
    char i;
    cin >> i;

    cout << endl;
    switch (i)
    {
    case 'y':
        tran = 1;
        break;
    case 'n':
        tran = 0;
        break;

    default:
        break;
    }
```

三、程序运行

1. 设置路由

2. 三次握手

（1） 客户端

```
Client: Sent SYN (First Handshake)
Client: Received SYN, ACK (Second Handshake)
Client: Received packet (SYN, ACK) validation successful!(Second Shake)
Client: Sent ACK (Third Handshake)
Client: Connection established successfully!
```

（2） 服务器端

```
Server: Received packet (SYN) validation successful! (First Shake)
Server: Send packet (SYN,ACK) (Second Shake)
Server: Received packet (ACK) validation successful! (Third Shake)
Server: connection is established successfully
```

3. 四次挥手

（1） 客户端

```
Exit
Client: Sent packet (FIN) (First wave)
Client: Received packet (FIN, ACK), verification successful.(Second wave)
Client: Received packet (FIN), verification successful.(Third wave)
Client: Sent packet (FIN, ACK) (Fourth wave)
Client: Connection closed
```

（2） 服务器端

```
Server: Received packet(FIN),vertification is successful.(First wave)
Server:Sent packet (FIN, ACK), verification successful.(Second wave)
Server:Sent packet (FIN), verification successful.(Third wave)
Server: Received packet (FIN, ACK), verification successful.(Fourth wave)
Connection close...
```

4. 正常传输

设置丢包率为 0%，延时为 0ms。

发送端：

```
本文件数据长度为 1857353Bytes, 需要传输227个数据包
Case 0: State 0-Sent        seq: 2          count : 0          length : 8192          checksum : 36448
Case 1: State 1-Received    ack: 2          Ack : 1            length : 0             checksum : 48101
Case 2: State 2-Sent        seq: 3          count : 1          length : 8192          checksum : 48090
Case 3: State 3-Received    ack: 3          Ack : 1            length : 0             checksum : 48100
```

```
Case 0: State 0-Sent         seq: 2             count : 226               length : 5961               checksum : 4841
Case 1: State 1-Received     ack: 2             Ack : 1                   length : 0                  checksum : 48101
Client: Sent packet (END)
Client: Received server packet (END, ACK), File transfer completed.
Total time: 0.944 s
Throughput: 15.0111Mbps
```

接收端：

```
Case 0: State 0-Received     seq: 2             count : 0                 length : 8192               checksum : 36448
Case 0: State 0-Sent         ack: 2             ACK : 1                   length : 0                  checksum : 48101              BINGO!

Case 1: State 1-Received     seq: 3             count : 1                 length : 8192               checksum : 48090
Case 1: State 1-Sent         ack: 3             ACK : 1                   length : 0                  checksum : 48100              BINGO!
```

```
Case 1: State 1-Received     seq: 3             count : 225               length : 8192               checksum : 24862
Case 1: State 1-Sent         ack: 3             ACK : 1                   length : 0                  checksum : 48100              BINGO!

Case 0: State 0-Received     seq: 2             count : 226               length : 5961               checksum : 4841
Case 0: State 0-Sent         ack: 2             ACK : 1                   length : 0                  checksum : 48101              BINGO!

Transmission complete.
```

（1） 发送端的 4 个状态：

State 0 表示发送方准备发送一个序列号为 2 的数据包,这个状态下，数据被复制到缓冲区，构造数据包，设置序列号为 2，并发送数据包。

State 1 表示发送端等待接收来自接收端的确认，如果接收到正确的确认（ACK=2），则进入 State 2；否则，根据超时进行重传 State 0 的数据包。State 1 表示发送方已发送数据包，等待来自服务器的确认。

State 2 表示发送方准备发送一个序列号为 3 的数据包，这个状态下，数据被复制到缓冲区，构造数据包，设置序列号为 3，并发送数据包。

State 3 表示发送端等待接收来自接收端的确认，如果接收到正确的确认（ACK=3），则进入 State 0；否则，根据超时进行重传 State 2 的数据包。State 3 表示发送方已发送数据包，等待来自服务器的确认。

（2） 接收端的两个状态：

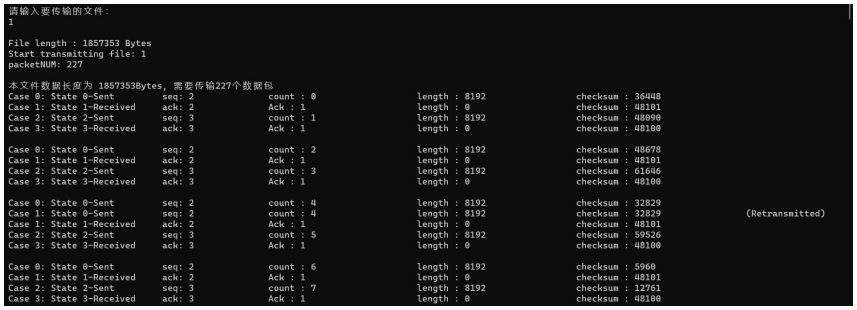
State 0:接收来自客户端的数据包,State 0 预期收到的数据包的 seq=2。如果接收到序号为 2 的数据包且校验和通过，发送 ACK 为 2 的确认，表示正确确认，进入到状态 1；如果接收到序号为 3 的数据包或校验和不通过，发送 ACK 为 3 的确认,并保持在阶段 0，告知发送端数据包有误。

State 1: 接收来自客户端的数据包,State 1 预期收到的数据包的 seq=3。如果接收到序号为 3 的数据包且校验和通过，发送 ACK 为 3 的确认，表示正确确认，进入到状态 0；如果接收到序号为 2 的数据包或校验和不通过，发送 ACK 为 2 的确认,并保持在阶段 1，告知发送端数据包有误。

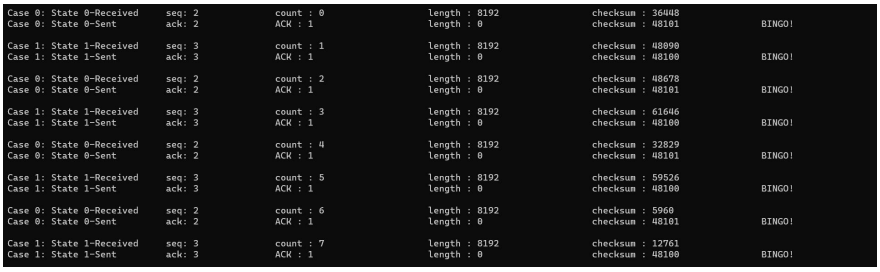
5. 丢包

设置丢包率为 20%（每 5 个包丢一个），延时为 0ms

发送端：



接收端：



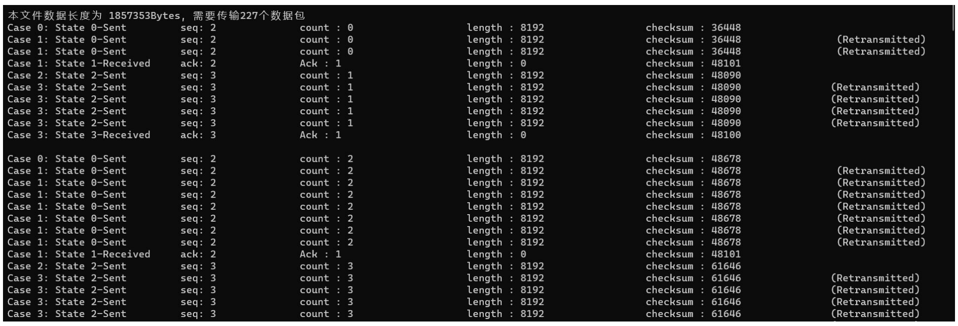
每 5 个包丢一个，接收端无法发送这个包的 ACK 确认，在发送端触发超时重传。如果发送端处于 State 1 状态，则重传 State 0 的数据包，如果处

于 State3 的状态，则重传 State 2 的数据包。

6. 超时重传

设置丢包率为 0%，延时为 110ms（程序中设置的 MAX_TIME=100）。

发送端：



接收端：

```
Case 0: State 0-Received     seq: 2          count : 0          length : 8192          checksum : 36448
Case 0: State 0-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          BINGO!

Case 1: State 1-Received     seq: 2
Case 1: State 1-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          WRONG!

Case 1: State 1-Received     seq: 2
Case 1: State 1-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          WRONG!

Case 1: State 1-Received     seq: 3          count : 1          length : 8192          checksum : 48090
Case 1: State 1-Sent         ack: 3          ACK : 1            length : 0             checksum : 48100          BINGO!

Case 0: State 0-Received     seq: 3
Case 0: State 0-Sent         ack: 3          ACK : 1            length : 0             checksum : 48100          WRONG!

Case 0: State 0-Received     seq: 3
Case 0: State 0-Sent         ack: 3          ACK : 1            length : 0             checksum : 48100          WRONG!

Case 0: State 0-Received     seq: 3
Case 0: State 0-Sent         ack: 3          ACK : 1            length : 0             checksum : 48100          WRONG!

Case 0: State 0-Received     seq: 3
Case 0: State 0-Sent         ack: 3          ACK : 1            length : 0             checksum : 48100          WRONG!

Case 0: State 0-Received     seq: 2          count : 2          length : 8192          checksum : 48678
Case 0: State 0-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          BINGO!

Case 1: State 1-Received     seq: 2
Case 1: State 1-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          WRONG!

Case 1: State 1-Received     seq: 2
Case 1: State 1-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          WRONG!

Case 1: State 1-Received     seq: 2
Case 1: State 1-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          WRONG!

Case 1: State 1-Received     seq: 2
Case 1: State 1-Sent         ack: 2          ACK : 1            length : 0             checksum : 48101          WRONG!
```

发送端 State 1 状态（State 3 状态）发送端没能收到 ACK，超时重传，重发 State 0（State 2 状态）的数据包，但并不知道是因为数据包丢失还是延时导致的没收到 ACK。因为本次设置的是延时，超时重传会导致冗余分组，rdt2.1 为解决这个问题引入了 seq，rdt2.2 取消了 NAK。

接收端检测数据序号，如果是冗余数据，告诉发送端收到的信息是错误的（接收端 State 0 接收 seq=3 的数据包，State 1 接收 seq=2 的数据包就是错误的信息），丢弃数据。

7. 在接收文件夹中查看传输结果