

# 计算机网络第三次实验报告

- 姓名：刘沿辰
- 学号：2012543
- 年级：2020级

## 实验要求

1. 实现单向传输
2. 给出详细的协议设计
3. 完成详细的实验报告
4. 编写的程序应结构清晰，具有较好的可读性
5. 提交程序源码和实验报告

## 实现平台

Windows11系统

Microsoft VisualStudio2022

## 协议设计

本次实验的协议设计参照UDP协议：建立连接时经历三次握手，断开连接时经历四次挥手，差错检测使用校验和方式，确认重传包含往返两条消息并实现了传输错误和丢包的数据重传，流量控制采用最原始的停等机制。

## 文件头部

本协议的文件头设计如下：

```
struct MsgHead {  
    u_short len;           // 数据长度，16位  
    u_short checksum;      // 校验和，16位  
    unsigned char type;    // 消息类型  
    unsigned char seq;     // 序列号，可以表示0-255  
};
```

文件头共48位，前16位为数据部分（不包括文件头）长度，中间16位为校验和，最后十六位分成两个八位使用，前一个是消息类型，后一个是序列号。这个精简的文件头包含了本次传输需要的

所有信息，并且得到了充分的利用。

由于数据长度仅有16位，换算过来每次传输的消息长度不能超过8192个char型变量，于是本次实验中将消息内容长度限制在了8000个字符以内。

type字段包含八位，本协议的设计中采用一位代表一种状态的方式，因此type字段最多可以表示八种状态及其叠加，实际使用到的状态数为5种，如下图所示：

```
#define SYN 1
#define ACK 2
#define FIN 4
#define PSH 8
#define OVE 16
```

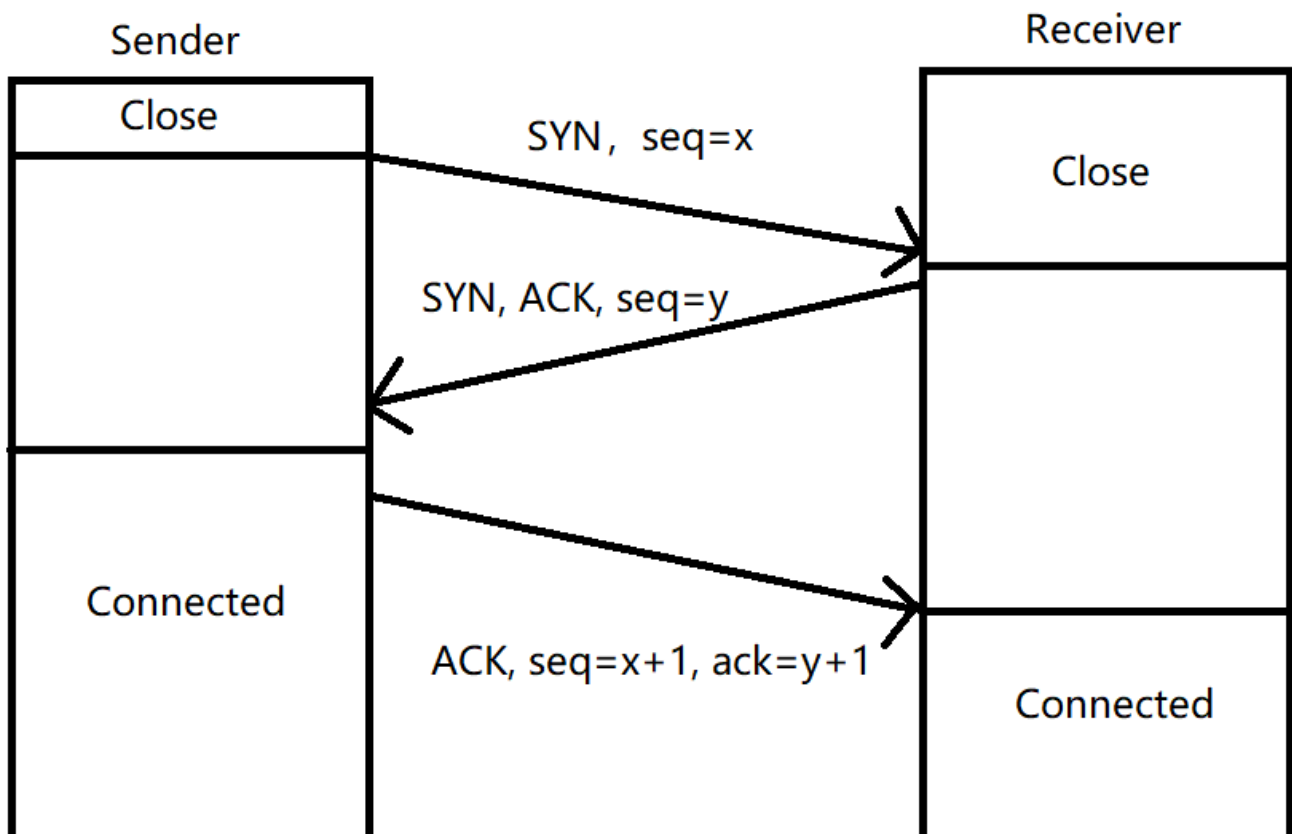
前三种状态和TCP中状态完全一致，在此不做赘述。

第四种状态PSH表示发送的消息包含实际内容（而非简单的校验式消息）。

第五种状态OVE（over）在传输的文件很大的时候使用。在这种场景下文件无法一次传输完毕，需要多次传输，OVE用于标识当前传输的文件已经传输完毕，接收端可以将数据包进行整合。

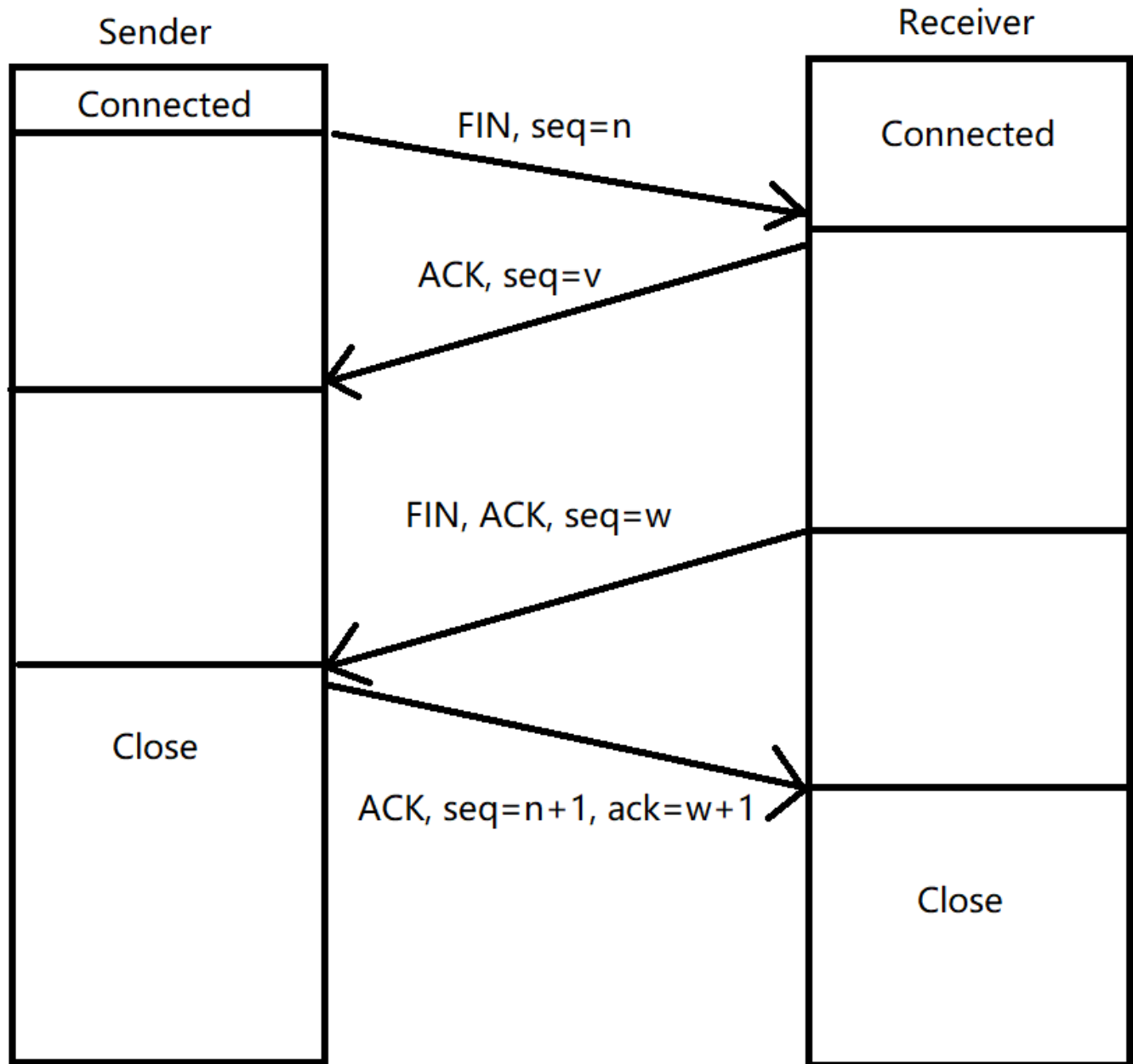
## 建立连接与断开连接

建立连接时，协议需要三次握手，过程如下：



首先由Sender向Receiver发出带SYN标记的连接申请，Receiver收到后返回带SYN，ACK标记的消息，Sender收到后又向Receiver发送带ACK标记的消息，然后便进入连接模式；receiver收到带ACK标记的消息后，也进入连接模式，然后就可以传送文件了。三次握手保证了双方均可正确发送消息与收到对方消息，保证了连接建立的稳定。

类似的，在断开连接时，协议有四次挥手，过程如下：



四次挥手保证了双方的信息传送完整，双方均知道对方断开，能合理关闭资源。

## 校验和

在上文中可以看到校验和包含在文件头中，是该网络协议下每个传输消息的第16-31位。其计算方法是将一则消息每16位叠加相加，得到一个32位的数（若溢出则加到最低位），当32位数的高

16位不为0时，就反复将高十六位加到低十六位上，直到高16位数字为0。此时低十六位就是我们需要的校验位。

当消息到达另一边时，使用相同的方法计算校验和，最终得到全0，说明传输正确。在接收每一条传输的消息时，都应计算校验和来判断是否接受。

## 序列号

程序中保存着一个8位u\_short来保存seq，代表当前发送的消息序列号，用于后续的文件重传等操作。由于seq的8位只能表示0-255，所以seq每次计算都应对256取余。

## 文件传输

由于传输的文件多种多样，我们无法在这个层面上对文件进行传输，故使用更底层的数据结构——二进制文件进行传输。我们会二进制读取每个文件，将其分类打包后传到另一方手上，然后整合保存，无论什么文件都可以这样传输。

前文提到消息的发送方维护着一个seq表示发送的消息序列号，而消息接收方维护着last\_Ack表示最后正确接收到的消息序列号。有这两个信息，程序便可以判断其发送的包是否在发送途中遗失，以及确认消息是否遗失等。具体的发送模块实现思路如下：

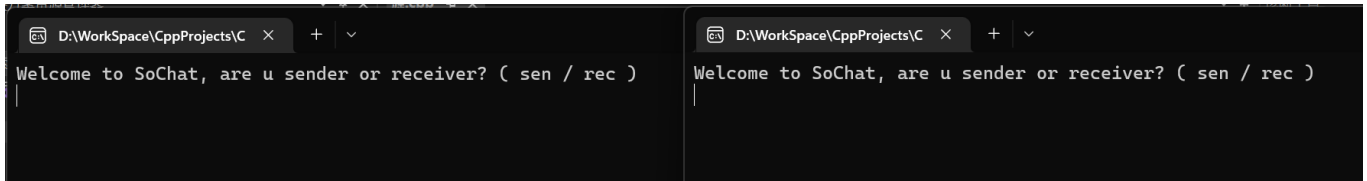
- 根据已经发送的消息长度和维护的seq变量确认要分段发送的消息，并设置消息头；
- 发送消息，等待接收Receiver发送的确认信息，并开启时长为1秒的计时；
- 若收到ACK消息，则计算校验和，如果不为0则不接受，认为消息出现差错，并重发消息；
- 若1秒内未收到ACK消息，则不再等待，认为消息没有正确送达，已经超时，重发消息；
- 若收到ACK消息且校验和为0，则确认本条消息已经正确发出，修改“已发送消息长度”和seq，准备发送下一条消息（然后回到第一条）；
- 若所有消息发送完毕，则发出一条包含type字段OVE==1的消息，表示文件传输完毕；如果这条消息被正确收到，则跳出循环，文件发送结束，如果没有被正确收到，则再次发送，直到对方正确收到。

停等机制需要双方的配合，具体的接收模块实现思路如下：

- 接收端等待接收消息，并预设好缓冲区保存消息内容；
- 收到消息后，计算校验和并检查type等字段是否正确；
- 若正确，则发送一个确认消息，代表这条消息已经收到，lastAck=seq；
- 若不正确，查看seq是否等于lastAck-1，若等于则什么也不做；
- 若是校验和或type字段的错误，则认为没有收到正确的消息，即消息出现差错，发出确认消息要求发送端重传；
- 正确接收一条消息后，将消息头去除，并把消息内容和前面收到的消息内容拼接起来；
- 若正确收到的消息type字段OVE==1，则说明文件已传输完毕，在接收完这个字段后便跳出接收循环，将文件写到指定位置。

## 运行结果

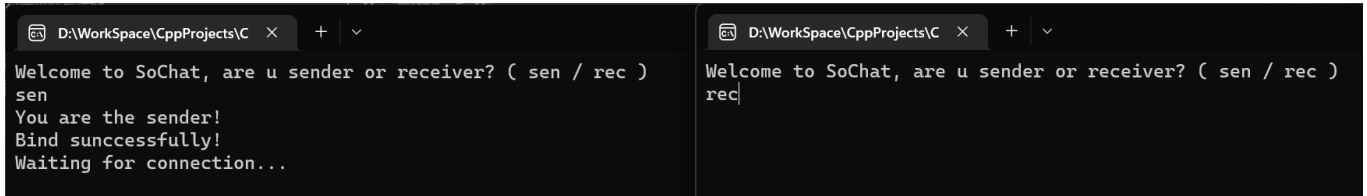
由于程序是端对端传输，所以需要双开程序，一个作为发送端，一个作为接收端。



```
D:\Workspace\CppProjects\C x + v
Welcome to SoChat, are u sender or receiver? ( sen / rec )

D:\Workspace\CppProjects\C x + v
Welcome to SoChat, are u sender or receiver? ( sen / rec )
```

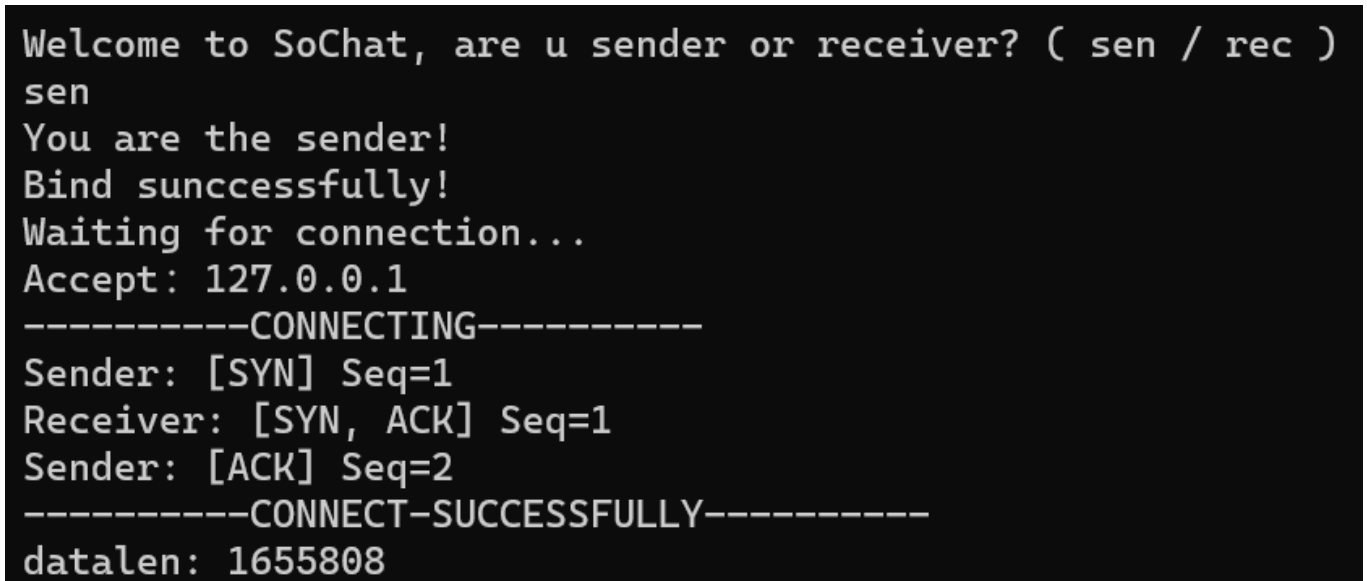
输入sen代表发送端，rec代表接收端。



```
D:\Workspace\CppProjects\C x + v
Welcome to SoChat, are u sender or receiver? ( sen / rec )
sen
You are the sender!
Bind suncccessfully!
Waiting for connection...

D:\Workspace\CppProjects\C x + v
Welcome to SoChat, are u sender or receiver? ( sen / rec )
rec
```

双方确认后，由于端口号等信息已经在程序中包含，无需输入，程序会直接建立连接，然后读取要传输的文件。sender端信息输出更完整，故此处以sender端为例。



```
Welcome to SoChat, are u sender or receiver? ( sen / rec )
sen
You are the sender!
Bind suncccessfully!
Waiting for connection...
Accept: 127.0.0.1
-----CONNECTING-----
Sender: [SYN] Seq=1
Receiver: [SYN, ACK] Seq=1
Sender: [ACK] Seq=2
-----CONNECT-SUCCESSFULLY-----
datalen: 1655808
```

可以看到三次握手的连接建立过程，以及读取到的文件大小为1655808比特。

```
Send length: 8000, seq: 20, rate: 8%
Send successfully
Send length: 8000, seq: 21, rate: 8%
Send successfully
Send length: 8000, seq: 22, rate: 9%
Send successfully
Send length: 8000, seq: 23, rate: 9%
Send successfully
Send length: 8000, seq: 24, rate: 10%
Send successfully
Send length: 8000, seq: 25, rate: 10%
Send successfully
Send length: 8000, seq: 26, rate: 11%
Send successfully
Send length: 8000, seq: 27, rate: 11%
Send successfully
Send length: 8000, seq: 28, rate: 12%
Send successfully
```

传输过程中，每传输一个包都会打印包的大小，当前seq以及传输了多少。

```
Send length: 7808, seq: 209, rate: 99%
Send successfully
Time cost: 371millisecond
Through output: 4451096 bit/second
-----DISCONNECTING-----
Sender: [FIN] Seq=210
Receiver: [ACK] Seq=2
Receiver: [FIN ACK] Seq=3
Sender: [ACK] Seq=211
-----DISCONNECT-SUCCESSFULLY-----
```

当最后一个包传输完成后，整个文件传输完成，程序自动计算花费时间以及每秒吞吐量。然后断开连接，四次挥手过程正确显示。

## 测试

由于UDP数据传输已经很稳定，很少出现丢包和错传等情况，所以我使用了随机数用于模拟丢包，来测试协议可靠性。

```
int error_rate = 0;
int rerror_rate = 0;
int lost_rate = 0;
int rlost_rate = 0;
```

上述四个字段用于模拟实际情况，分别代表发送端发包数据错误率、接收端发包数据错误率、发送端发包丢失率、接收端发包丢失率。将发送端的错误率和丢包率修改为5%后，程序运行情况如下：

```
Send length: 8000, seq: 10, rate: 3%
Send fail!
Send length: 8000, seq: 10, rate: 3%
Send successfully
Send length: 8000, seq: 11, rate: 3%
Send successfully
Send length: 8000, seq: 12, rate: 4%
Send successfully
Send length: 8000, seq: 13, rate: 4%
Send successfully
Send length: 8000, seq: 14, rate: 5%
Send successfully
Send length: 8000, seq: 15, rate: 5%
Send successfully
Send length: 8000, seq: 16, rate: 6%
Send successfully
Send length: 8000, seq: 17, rate: 6%
Send successfully
Send length: 8000, seq: 18, rate: 7%
Send successfully
Send length: 8000, seq: 19, rate: 7%
Send successfully
Sender packet lost!
Send length: 8000, seq: 20, rate: 8%
Send time error!
Send length: 8000, seq: 20, rate: 8%
Send successfully
Send length: 8000, seq: 21, rate: 8%
Send successfully
```

可以看到，程序在运行时遇到的Send fail（发送数据包错误），Sender Packet lost（发送数据包丢失）以及Send time error（接收端回应超时）都得到了正确解决，协议设计有效。