计算机网络第四次实验报告

- 姓名：刘沿辰
- 学号：2012543
- 年级：2020级

实验要求

1. 将停等机制改成基于滑动窗口的流量控制机制
2. 给出实现的拥塞控制算法的原理说明
3. 完成给定测试文件的传输，显示传输时间和平均吞吐率
4. 完成详细的实验报告
5. 编写的程序应结构清晰，具有较好的可读性
6. 提交程序源码和实验报告

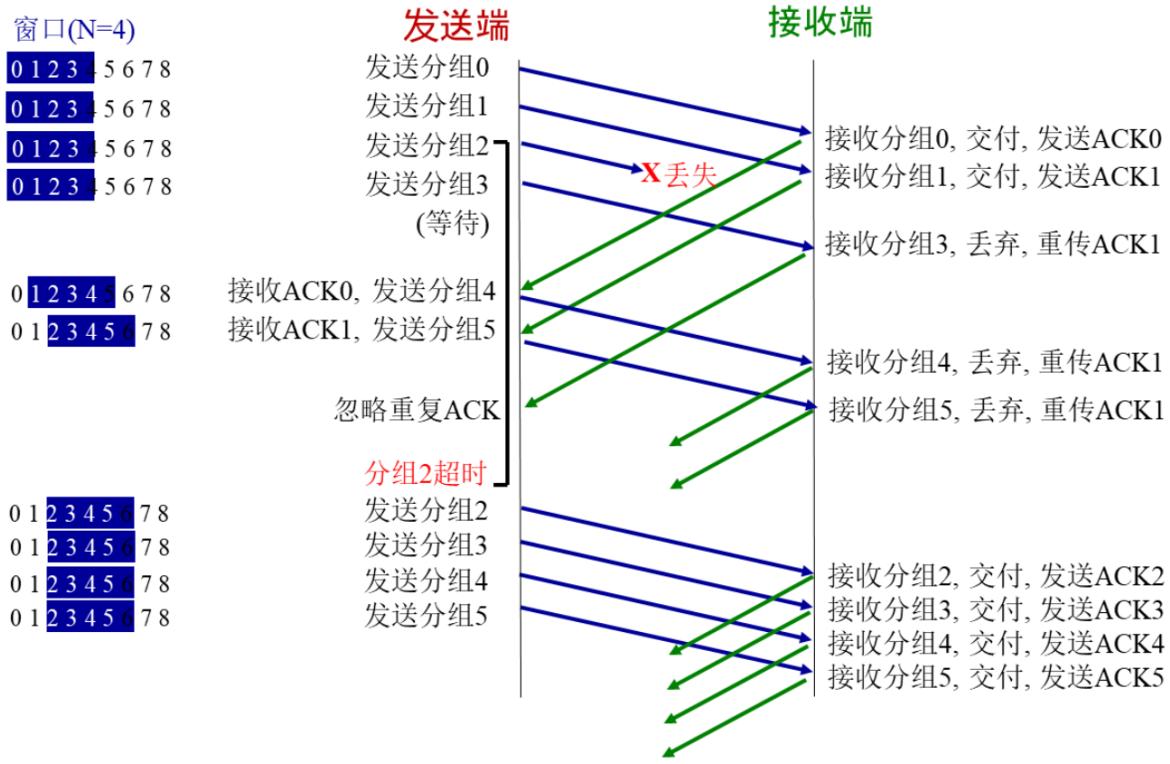
实现平台

Windows11系统
Microsoft VisualStudio2022

算法选择

本次实验目标是在实验3-2的基础上实现一种拥塞控制算法，我在原有代码的基础上参考Reno拥塞控制算法进行实现。

我的实验3-2实现滑动窗口的方式为gbn（go-back-n），示意图如下：



数据包的顺序接收对于根据每个ack来调整窗口大小的拥塞控制方式是大有裨益的，这是我选择Reno控制算法实现的原因之一。

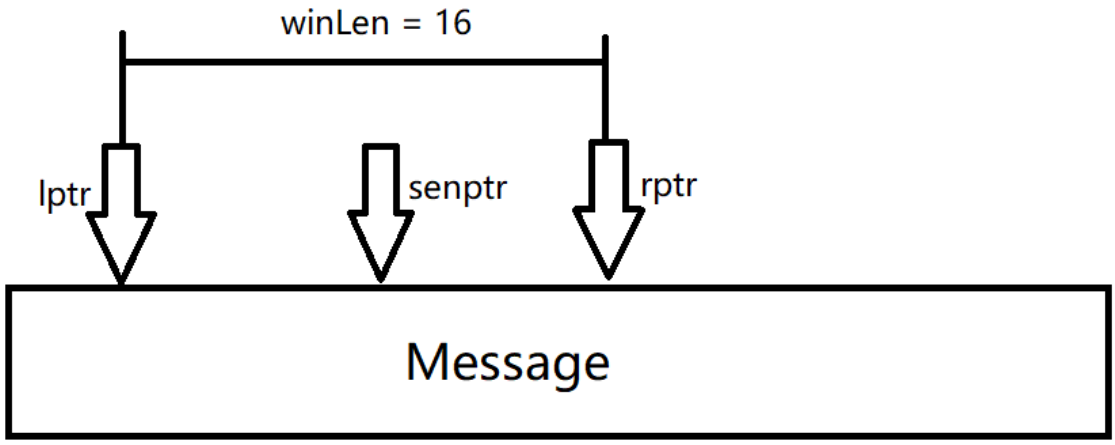
我的滑动窗口使用三个指针实现，分别是标示滑动窗口左边界的lptr，标示滑动窗口右边界的rptr和标示当前正在传输的数据包的指针senptr。窗口采用左闭右开的形式，即实际窗口范围为：

$$[lptr, rptr)$$

即满足

$$lptr <= senptr < rptr$$

示意图如下：

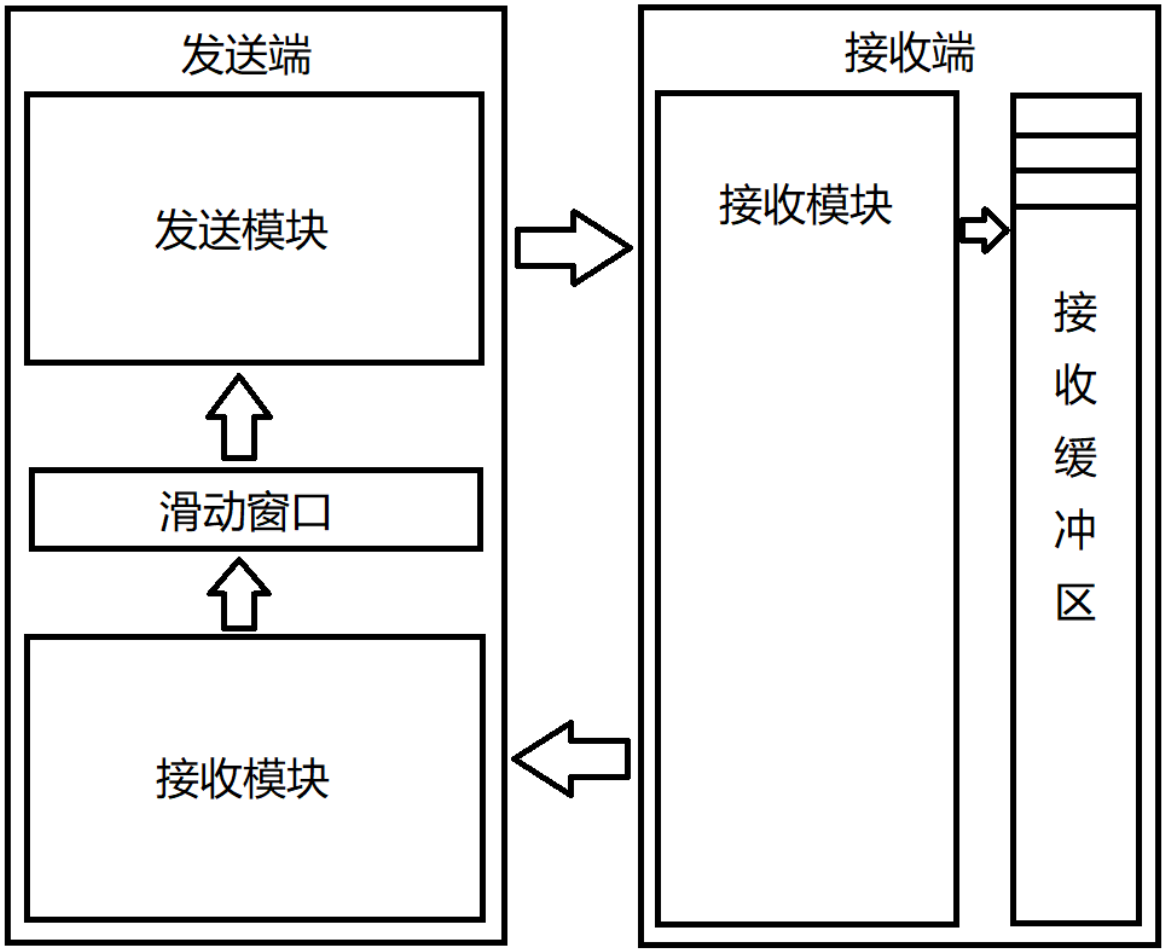


在实现滑动窗口时，我对窗口的长度设置了单独变量winLen（本次实验中初始化为1），并在实际实现中保证了这个变量与其他代码的低耦合度，在滑动窗口运行时也可以对winLen进行操作。以下图为例：

```
if (lptr - (int)h1.ack == 0) {
    change_win(1);
    lptr++;
    rptr = lptr + (int)winLen;
    cout << "窗口左侧移动至" << lptr << ", 窗口右侧移动至" << rptr << endl;
    if (lptr == maxPackge) {
        wmtx.unlock();
        mtx.unlock();
        return;
    }
}
else if (lptr - (int)h1.ack < 0) {
    cout << endl;
    change_win(lptr - (int)h1.ack + 1);
    cout << "senptr从 " << senptr;
    senptr = (int)h1.ack + 1;
    cout << " 移动至" << senptr << endl;
    lptr = (int)h1.ack;
    rptr = lptr + (int)winLen;
    cout << "窗口左侧移动至" << lptr << ", 窗口右侧移动至" << rptr << endl;
}
```

所有移动窗口位置的地方均使用winLen。由于滑动窗口的左指针标识着最后正确传输的数据包，所以当winLen变化时，程序会调整窗口右边界来保证窗口长度。

而在实现整体程序时，我采取了多线程来操作，示意图和模块功能如下：

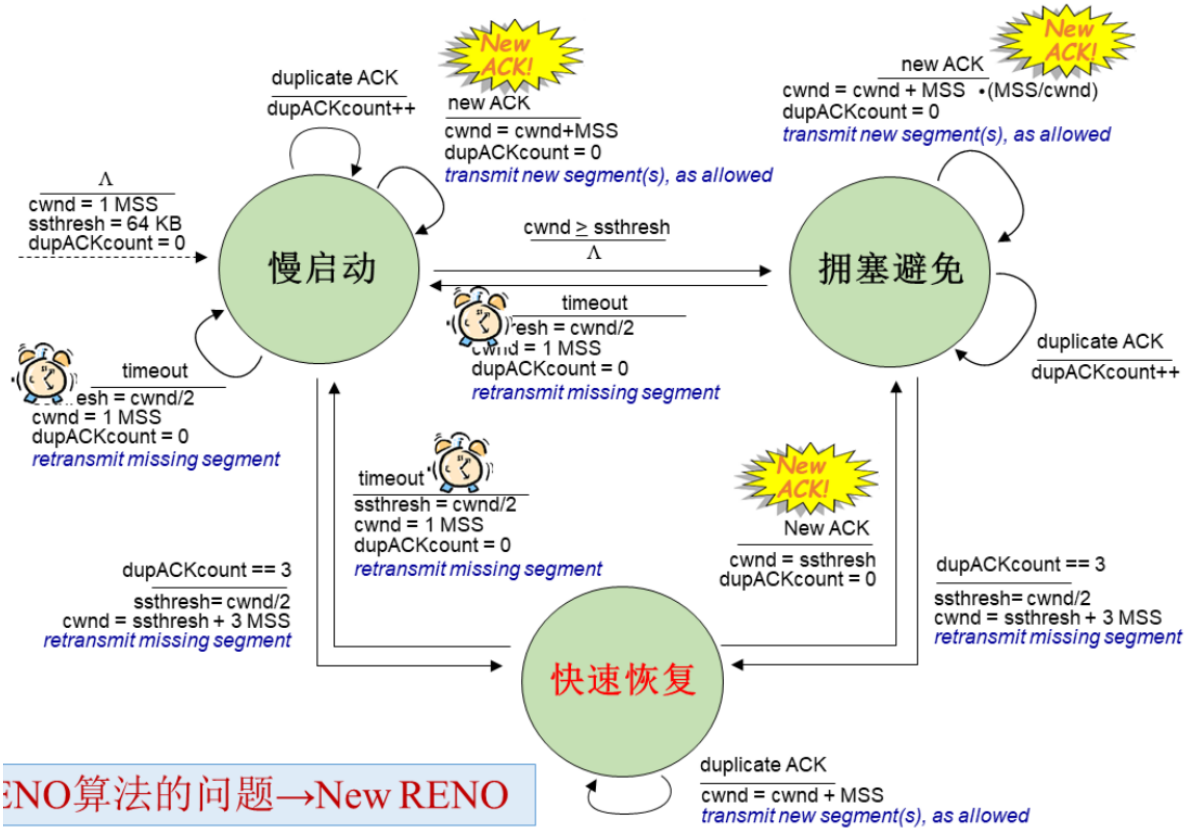


- 发送端发送模块：发送模块**只可以控制senptr**，即当前发送包的指针。其主要功能是发送senptr指向的数据包，然后将senptr指向下一个包，然后循环发送数据包……依此类推，直到数据包发完或遇到滑动窗口右边界，则卡住不动，等待接收模块调整滑动窗口。
- 接收端接收模块：接收端的接收模块不停的接收发送端发来的数据包，在收到数据包后返回一个ACK确认数据包，ack的值为上一个连续且正确收到的数据包的ack。并将正确接收到的ack包放到接收缓冲区中。
- 发送端接收模块：发送端的接收模块用于接受接收端发来的确认数据包并据此调整滑动窗口，该模块大部分时间**只能控制lptr和rptr**，只有在涉及重传时会修改senptr。若收到的确认包ack和滑动窗口左边界相同，则代表左窗口第一个数据包已经被正确收到，窗口整体右移一格；发送模块检测到窗口右移之后，就可以继续发出新的数据包。若收到的ack比左边界大，则可以直接将窗口移动到ack+1的位置，因为在此之前的数据包都已经被正确接收。

各个模块有自己的功能，并且有明确的控制项，这对窗口大小的调整也提供了便利。如若实现Reno算法，只需要在调整窗口的发送端接收模块中修改窗口长度即可。综上，本次的拥塞控制算法采用Reno算法作为参考实现。

算法明细

Reno控制算法主要将程序的发送分为三个状态：慢启动状态，拥塞避免状态和快速恢复状态。三状态的状态转换图如下：



本次实验中设置`ssthresh`值为65，为慢启动阶段窗口大小的上限。

- 慢启动阶段：当`winLen`的值小于`ssthresh`时，TCP则处于`slow start`阶段，每收到一个ACK，`cwnd`的值就会加1。该阶段窗口大小对RTT呈指数增长。
- 拥塞避免阶段：当`winLen`的值超过`ssthresh`时，就会进入拥塞避免阶段。该阶段下，每收到一个ACK，`winLen`增加`1 / winLen`，该阶段窗口大小对RTT接近线性增长。
- 快速恢复阶段：当收到三个重复的ACK时，进入快速恢复状态。Reno算法会把`ssthresh`的值设置为当前`winLen`的一半，并将`winLen`设置为（更新后的）`ssthresh + 3`，回到拥塞避免阶段。
- 超时：如若发生超时，则将`ssthresh`置为`winLen`的一半（最小为2），并将winLen置为1，回到慢启动阶段。

在程序开始运行时，滑动窗口大小被初始化为1，程序处于慢启动阶段。之后每个RTT窗口大小翻倍，直到达到`ssthresh`后，进入拥塞避免阶段，窗口线性增长。若遇到三次重复ACK，则转入快速恢复状态，立即重传缺失数据包，并更新ssthresh和winLen的值。

算法实现

滑动窗口实现方式已经在前文提出，此处仅写Reno算法三个状态的具体实现。

慢启动阶段

慢启动阶段的条件是窗口长度`winLen`小于`ssthresh`，此时每收到一个正确ACK就将窗口长度加1，实现如下：

```
if (winLen < ssth) {
    cout << "窗口长度从" << winLen;
    winLen += len_chg;
    cout << "变为" << winLen << endl;

}
```

拥塞避免阶段

当`winLen > ssthresh`时，程序进入拥塞避免阶段，此时程序每收到一个正确ACK就将窗口长度加`1 / winLen`，实现如下：

```
else {
    cout << "窗口长度从" << winLen;
    winLen += 1 / winLen;
    cout << "变为" << winLen << endl;
}
```

在实际实现时，发送端的接收模块无需知道程序此时的状态，所以上述两个状态的处理被集中到`change_win`函数中执行，程序只需要调用函数，并传入窗口的变化参数即可。

```
void change_win(int len_chg = 0) {
    if (winLen < ssth) {
        cout << "窗口长度从" << winLen;
        winLen += len_chg;
        cout << "变为" << winLen << endl;

    }
    else {
        cout << "窗口长度从" << winLen;
        winLen += 1 / winLen;
        cout << "变为" << winLen << endl;
    }
}
```

快速重传阶段

当程序连续接收到3个重复ACK时，便触发快速重传阶段。为了记录重复ACK，引入两个域：

```
int rec_last_ack = -5;
int rec_time = 1;
```

其中`rec_last_ack`域记录着上一次收到的ack，`rec_time`域记录该ack收到的次数，维护代码如下：

```
if ((int)h1.ack == rec_last_ack) {
    rec_time++;
}
else {
    rec_last_ack = (int)h1.ack;
    rec_time = 0;
}
```

最后，程序会检查rec_time是否为3，来判断是否进入快速重传阶段：

```
if (rec_time == 3) {
    senptr = rec_last_ack + 1;
    cout << "数据包" << senptr << "三次重复ack，需要重传" << endl;

    cout << "ssthresh从" << ssth;
    ssth = winLen / 2;
    cout << "变为" << ssth << endl;

    cout << "窗口长度从" << winLen;
    winLen = ssth + 3;
    cout << "变为" << winLen << endl;
}
```

此时，接收端最后收到的正确数据包为`rec_last_ack`，所以我们重传的数据包指针赋值为`rec_last_ack + 1`，然后更新`ssthresh`的值为当前`winLen`的一半，并将`winLen`设置为（更新后的）`ssthresh + 3`，然后退出函数。下一次收到ack时，程序将回到正常数据处理流中，调用`change_win`函数修改窗口大小。

超时

程序使用内置的时间数组记录超时情况，如果遇到数据包超时，则程序输出超时信息，并将`ssthresh`置为`winLen`的一半（最小为2），winLen置为1，回到慢启动阶段。

运行结果

连接建立阶段和三次握手阶段已在之前的实验报告中给出，此处主要观察窗口的变化情况。

刚开始运行程序，程序处于慢启动阶段，窗口大小对RTT呈指数增长。在第一个RTT中，窗口大小为1，程序只发出一个数据包，收到反馈后将窗口长度调整为2：

```
Send length: 8000, senptr: 0, rate: 0%
time out
lptr 0, rptr 1, senptr 1
time out
lptr 0, rptr 1, senptr 1
senptr从 1 移动至0
Send0 successfully
窗口长度从1变为2
窗口左侧移动至1，窗口右侧移动至3
```

到了第二个RTT，程序将会发出两个数据包，收到反馈后将窗口长度调整为4：

```
senptr: 0
Send length: 8000, senptr: 0, rate: 0%
senptr: 1
Send length: 8000, senptr: 1, rate: 0%
senptr: 2
Send1 successfullySend length: 8000, senptr: 2, rate: 1%

time out
lptr 1, rptr 3, senptr 3
窗口长度从2变为3
窗口左侧移动至2，窗口右侧移动至5
Send2 successfully
窗口长度从3变为4
窗口左侧移动至3，窗口右侧移动至7
```

类似的，在第三个RTT中，窗口长度将被调整为8，然后以此类推……

```
senptr: 3
Send length: 8000, senptr: 3, rate: 1%
senptr: 4
Send3 successfully
Send length: 8000, senptr: 4, rate: 2%
senptr: 5
Send length: 8000, senptr: 5, rate: 2%
senptr: 6
Send length: 8000, senptr: 6, rate: 3%
time out
lptr 3, rptr 7, senptr 7
窗口长度从4变为5
窗口左侧移动至4，窗口右侧移动至9
Send4 successfully
窗口长度从5变为6
窗口左侧移动至5，窗口右侧移动至11
Send5 successfully
窗口长度从6变为7
窗口左侧移动至6，窗口右侧移动至13
Send6 successfully
窗口长度从7变为8
窗口左侧移动至7，窗口右侧移动至15
```

而当窗口长度达到`ssthresh`时，程序进入拥塞避免阶段，此时窗口长度每次增加`1 / winLen`，对RTT呈类线性增长：

```
Send length: 8000, senptr: 126, rate: 54%
time out
lptr 63, rptr 127, senptr 127
窗口长度从64变为65
窗口左侧移动至64，窗口右侧移动至129
Send71 successfully

窗口长度从65变为65.0154
senptr从 127 移动至72
窗口左侧移动至71，窗口右侧移动至136
Send72 successfully

窗口长度从65.0154变为65.0308
senptr从 72 移动至73
窗口左侧移动至72，窗口右侧移动至137
Send73 successfully

窗口长度从65.0308变为65.0461
senptr从 73 移动至74
窗口左侧移动至73，窗口右侧移动至138
Send74 successfully
```

程序保持拥塞避免阶段，直到数据包出现丢失：

```
senptr: 176
Send length: 8000, senptr: 176, rate: 76%
senptr: 177
Send length: 8000, senptr: 177, rate: 76%
senptr: 178
----------------------------------------------------------------
Sender packet lost!
----------------------------------------------------------------
Send length: 8000, senptr: 178, rate: 77%
senptr: 179
----------------------------------------------------------------
Sender packet lost!
----------------------------------------------------------------
Send length: 8000, senptr: 179, rate: 77%
senptr: 180
----------------------------------------------------------------
Sender packet lost!
```

此处从178号数据包开始遇到网络拥塞，此后连续n个数据包均被丢弃，所以接收端将一直返回ack=177

```
Send177 successfully
窗口长度从66.596变为66.611
窗口左侧移动至178，窗口右侧移动至244
Send177 successfully
Send177 successfully
数据包178三次重复ack，需要重传
ssthresh从65变为33.3055
窗口长度从66.611变为36.3055
senptr: 178
Send length: 8000, senptr: 178, rate: 77%
senptr: 179
Send178 successfully
```

当连续三次`Send177 successfully`时，程序转入快速恢复状态，调整`ssthresh`大小为窗口长度的`1 / 2`，并将`winLen`置为`ssthresh + 3`，然后立即重传数据包178，回到拥塞避免状态。

由此，便可不停调整窗口，直到文件传输完毕。可以看到，使用了Reno拥塞避免方式后，程序仅用1.2秒就完成了传输，效率得到极大提升，具体详细的对比将在实验3-4报告中写出。

```
----------File-transfer-succesfully!----------
Receiver error number: 0, Sender lost number: 9
Time cost: 1219 millisecond
Through output: 1523669 bit/second
```