# 基于UDP服务设计可靠传输协议并编程实现

李娅琦
2213603
计算机科学与技术

## 实验要求

在实验3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。

实验要求：

（1）实现单向传输。
（2）对于每个任务要求给出详细的协议设计。
（3）给出实现的拥塞控制算法的原理说明。
（4）完成给定测试文件的传输，显示传输时间和平均吞吐率。
（5）性能测试指标：吞吐率、文件传输时延，给出图形结果并进行分析。

## 实验设置

### 代码文件

本次文件的代码分为三个文件：

- packet.h: 对UDP数据包的协议设计
- client.cpp: 客户端代码
- server.cpp: 服务器代码

### 条件设置

- 设置服务器和客户端的IP都为：127.0.0.1
- 服务器端口号：8888
- 客户端端口号：9999
- 设置丢包率：10%，延时：20ms

## 实验原理

### 拥塞控制

本次实验采用端到端的拥塞控制方法：**Reno算法**

- 采用基于窗口的方法，通过拥塞窗口的增大或者减小控制发送速率。
    - 实际发送窗口取决于**接收通告窗口和拥塞控制窗口的较小值**
- 基于实验3-2的GBN算法，此次实验设置：
    - 接收端窗口大小为：1
    - 发送端窗口大小实时变化

关于Reno算法，有三个状态阶段(**cwnd为窗口大小, ssthresh为阈值**)：

- 慢启动：
    - 当**连接初始建立**或者**超时未收到ACK**，进入慢启动阶段
    - 每接收到一个新的ACK，cwnd+1，此时如果收到窗口内所有ack，cwnd就变成原来的二倍，类似于**cwnd翻倍**
    - 出现超时
        - ssthresh=cwnd/2, cwnd=1
        - 进入慢启动阶段
    - 如果收到三次重复的ACK
        - ssthresh=cwnd/2
        - cwnd=ssthresh+3
        - 进入快速恢复阶段
- 拥塞避免阶段：
    - 当**拥塞窗口到达阈值**，慢启动阶段结束，进入拥塞避免阶段
    - 每接收到一个新的ACK，cwnd不变，当收到窗口内所有的包的ACK（即经过每个RTT），所以**cwnd+=1（线性增长）**
    - 三次重复的ACK
        - ssthresh=cwnd/2
        - cwnd=ssthresh+3
        - 进入快速恢复阶段
    - 出现超时
        - ssthresh=cwnd/2, cwnd=1
        - 进入慢启动阶段
- 快速恢复阶段：
    - 接收到重复ACK，**cwnd+1**
    - 接收到新的ACK
        - cwnd=ssthresh
        - 进入拥塞避免阶段
    - 出现超时
        - ssthresh=cwnd/2, cwnd=1
        - 进入慢启动阶段

# Packet协议设计

数据包协议设计和校验和和数据包的差错校验函数，都与上次实验相同。

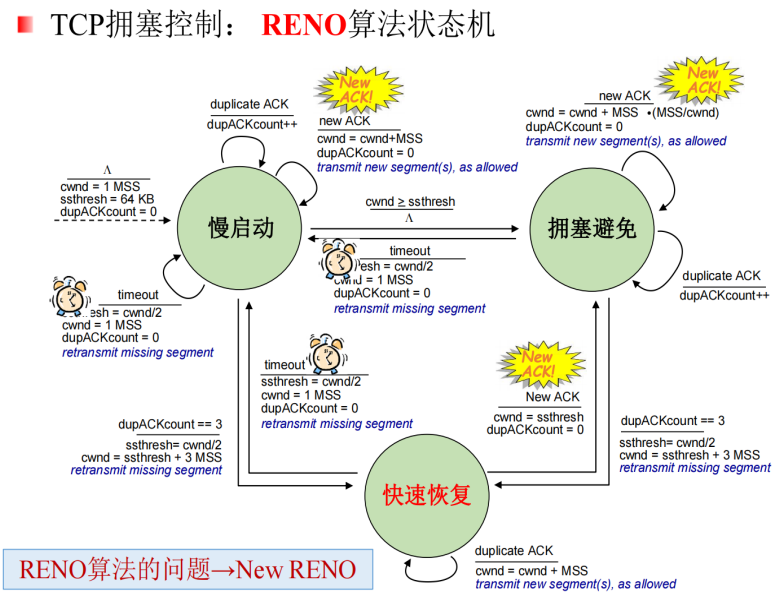
# 三次握手和四次挥手

与上次实验相同。

# 文件传输

### 整体逻辑

参考理论知识，关于文件传输，采用new Reno算法进行实现，具体的逻辑如下：




基于此，设置了以下变量

- **lastbyteacked：** 窗口起始位置
- **lastbytesent：** 下一个可用且待发送的序列号
- **last_recvack：** 上一个接收到的ack序列号
- **stime=clock_t[]：** 记录每个包的发送时间（实时更新）
- **cwnd：** 拥塞窗口大小，初始值为 1
- **ssthresh：** 阈值，初始为 64
- **count：** 计数，记录连续几个重复的ack
- **attribute：** 目前的状态
  - 1：慢启动阶段
  - 2：拥塞避免阶段
  - 3：快速恢复阶段
- **first：** 是否是初次进入拥塞控制阶段，初始为 false
- **yscount：** 拥塞阶段的计数，用来判断窗口内所有数据包的ACK是否都被接收到

## 拥塞控制

基于new Reno算法，大概的思路如下：

- 发送窗口内的可用还未发送数据包，并记录发送时间
- 每次发送完，都尝试接收ACK数据包
- 接收端进行不同情况的处理

大致的代码框架如下（具体细节后面详细解释）：

```
while (lastbyteacked < packnum) {
    if (lastbytesent < lastbyteacked + cwnd && lastbytesent < packnum) {// 窗口内
有数据包可以发送就发送并记录发送时间
        stime[lastbytesent] = clock();
        ...sendto(...);
        lastbytesent++; // 发送下一个包
```

```
    }
    ...recvfrom(...);// 设置非阻塞模式并接收
    if (clock() - stime[lastbyteacked] > TIMEOUT) {
        ...continue;//超时重传（后面详细解释）
    }
    if (sresult == -1) {continue;}//收不到接着发送
    if (ifcorrect(ackp, sizeof(ackp)) && ackp.sign == ACK) {//接收到ACK
        if (acknum >= lastbyteacked-1) {
            if (last_recvack != acknum) {//新的ack
                count = 0;
                if (cwnd >= ssthresh) {
                    ...//进入拥塞避免阶段
                }
                //不同状态下的处理（后面详细解释）
                if (attribute==1) {...}
                else if(attribute==2){...}
                else if(attribute==3){...}
                lastbyteacked = acknum + 1; // 窗口左边界向前滑动
            }
            else if (last_recvack == acknum) {
                ...//重复ack处理（后面详细解释）
            }
            if (count == 2&& (attribute == 1 || attribute==2)) {
                ...//三次重复ack处理（后面详细解释）
            }
        }
    }...// 改回阻塞模式
}
```

1. **超时重传**

- 通过判断**窗口内待接收ACK的第一个数据包**，决定是否超时
- 超时进行窗口内所有数据包重传，因此**lastbytesent = lastbyteacked**，lastbytesent是下一个可用待发送的数据包
- 对于拥塞控制方面
  - 窗口大小变为1，**cwnd=1**
  - 阈值变为原来窗口大小的一半
  - 对重复的ACK计数和拥塞控制阶段计算接收到的ACK数，都重置为 0
  - 将状态变为**慢启动状态**

```
if (clock() - stime[lastbyteacked] > TIMEOUT) {
    //超时重传
    lastbytesent = lastbyteacked;
    ssthresh = cwnd / 2;
    cwnd = 1;
    count = 0;
    yscount = 0;
    attribute = 1;//进入慢启动状态
    continue;
}
```

2. **重复与三次ACK**

- 采用count计数，采用last_recvack记录上一个收到的ACK
- 如果处于**慢启动或拥塞控制状态**
  - 每次接收到新的ACK，count重置为0
  - 重复ACK，则count+1
  - 当**count==2时(三次：0，1，2)**，代表出现三次重复ACK，出现丢包
    - 阈值设置为：**cwnd/2**
    - 窗口大小设置为：**ssthresh+3**
    - 窗口内所有数据包重传，因此**lastbytesent = lastbyteacked**，lastbytesent是下一个可用待发送的数据包
    - 将状态改为**快速恢复阶段**
- 如果是在**快速恢复阶段**
  - 接收到重复的ACK
    - 窗口大小：cwnd+1，保证还可以继续发送一个包
    - 其他条件不变

```
if (last_recvack == acknum) {
    if (attribute==3) {
        cwnd += 1;
        yscount = 0;
    }
    if (attribute == 1||attribute==2) {
        count++;
    }
}
last_recvack = acknum;
if (count == 2&& (attribute == 1 || attribute==2)) {//三次重复
    ssthresh = cwnd / 2;
    cwnd = ssthresh + 3;
    lastbytesent = lastbyteacked;// 丢包重传
    attribute = 3;
}
```

3. **慢启动**

在慢启动阶段，每收到一个ACK，窗口大小就会 +1 ，此时如果窗口内所有的数据包都收到了对应的ACK，那么窗口大小会增加一倍，就等于**cwnd*=2（增倍）**
但，问题在于**GBN累计确认**，因此有可能出现这样的情况：

- 收到5号ACK之后，收到7号ACK
- 此时代表6号和7号数据包，都被确认收到
- 因此，此时窗口大小应该增加 2
- 窗口也会实时移动

所以，窗口大小增加的值应该为：**acknum-lastbyteacked+1**

- acknum：收到的序列号

- lastbyteacked：窗口左边界

```
if (attribute==1) {
    cwnd += (acknum-lastbyteacked+1);
}
```

4. **拥塞控制**

在拥塞控制阶段，每经过一个RTT（即只有收到**窗口内所有数据包对应ACK**），窗口大小才会 +1 ，每接收到一个ACK，窗口大小是不变的。
但，因为窗口时实时移动的，因此只能通过计数来判断。

- 设置**yscount=0**，**first=false（是否是第一次进入拥塞控制阶段）**
- 当第一次进入拥塞控制阶段，从此时接收到的ACK为开始计数
  - 此时不需要增加yscount，因此需要first来判断
  - 并会把 first 置为 false
- 如果从其他的状态转为拥塞控制状态
  - first会重置
  - yscount会重置
- 之后每次又进入拥塞控制阶段时，对yscount进行增加
  - 根据接收到的acknum与上一次接收到的ACK的差，增加yscount
  - **yscount += acknum - last_recvack**
- 一旦**yscount与窗口大小相等**，此时窗口大小 +1
- 之后yscount会被重置，窗口大小也会被重新记录

```
if(attribute==2){//拥塞避免
    if (first) {
        cout << "初次进入不处理" << endl;
    }
    else {
        cout << "不是初次进入，进行计数" << endl;
        yscount += acknum - last_recvack;
    }
    first = false;
    lastwindow = cwnd;
    if (cwnd == yscount) {
        cwnd += 1;
        yscount = 0;
    }
}
```

5. **状态与窗口阈值转换**

- 处于慢启动状态
  - 如果窗口大小 >= 阈值，进入**拥塞控制状态**
  - 如果超时，进行1中的超时处理，状态不变

    - 三次重复ACK，进行2中的重复与三次ACK处理，进入**快速恢复阶段**

- 处于拥塞控制阶段
    - 如果超时，进行1中的超时处理，进入**慢启动状态**
    - 三次重复ACK，进行2中的重复与三次ACK处理，进入**快速恢复阶段**

- 处于快速恢复阶段
    - 收到新的ACK，进入**拥塞控制阶段**
    - 如果超时，进行1中的超时处理，进入**慢启动状态**
    - 收到重复的ACK，状态不变，窗口增加 1

具体的窗口阈值与状态转换代码如下：

```
if (clock() - stime[lastbyteacked] > TIMEOUT) {//超时
    ...
    ssthresh = cwnd / 2;
    cwnd = 1;
    attribute = 1;//进入慢启动状态
    continue;
}...
if (...) {
    if (...) {
        if (last_recvack != acknum) {//新的ack
            if (cwnd >= ssthresh) {
                ...
                attribute = 2;//进入拥塞避免阶段
            }
            if (attribute==1) {
                ...
                cwnd += (acknum-lastbyteacked+1);
            }
            else if(attribute==2){//拥塞避免
                ...if (cwnd == yscount) {
                    cwnd += 1;
                }
            }else if(attribute==3){//快速恢复
                cwnd = ssthresh;
                attribute = 2;//进入拥塞避免阶段
            }
        }
        else if (last_recvack == acknum) {//重复
            ..
        }
        if (count == 2&& (attribute == 1 || attribute==2)) {//三次重复的ack检测丢失
            ssthresh = cwnd / 2;
            cwnd = ssthresh + 3;
            attribute = 3;//进入快速恢复阶段
        }
    }
}
```

### 结束标志

当文件传输结束时（具体代码参考.cpp文件）：

- 客户端发送**OVER数据包**
- 服务器接收到后，**返回OVER数据包**
- 客户端接收到服务器的OVER数据包，表明此刻数据传输结束
- 最后会返回接收到的所有数据及长度进行**文件写入**

### 吞吐率和传输时间

在客户端进行文件名字和数据的发送，并且在发送数据时记录吞吐率和传输时间：

```
while (true)
{
    ...
    sendfile((char*)(file.c_str()), file.length(), true);// 发送文件名
    clock_t start1 = clock();
    sendfile(buffer, len, false);// 发送文件内容
    clock_t end1 = clock();
    cout << "传输总时间为:" << (end1 - start1) / CLOCKS_PER_SEC << "s" << endl;
    cout << "吞吐率为:" << fixed << setprecision(2) << (((double)len) / ((end1 -
start1) / CLOCKS_PER_SEC)) << "byte/s" << endl;
}
```

## 结果展示

### 建立连接

先运行server端，再运行client端：

```
Info:  正在监听客户端.....

-----开始握手-----

First:
Receive:  接收到客户端连接请求.....

Second:
Send:  服务器发送ACK确认.....

Third:
Receive:  接收到客户端ACK确认.....
Success:  服务器与客户端成功连接
Choice:  是否接收文件(Y/N)
```

```
-----开始握手-----

First:
Send:  发送连接请求成功,等待响应中...

Second:
Receive:  接收到来自服务器的ACK...

Third:
Send:  客户端发送ACK确认.....
Success:  服务器与客户端成功连接

Choice:  请选择是否输入文件(Y/N)
```

### 拥塞控制

1. **慢启动**

从下图可以看到，慢启动阶段，每收到一个ACK，窗口大小增加 1 ，窗口范围发生变化，然后又有新的可以发送的数据包。

```
窗口大小: 1  阈值: 64
Send:  发送包, 序列号: 0
Receive:  接收到来自服务器的ACK 0
新ACK   状态: 1
窗口大小: 2  阈值: 64
窗口范围: 1  ~  2
Send:  发送包, 序列号: 1
Send:  发送包, 序列号: 2
Receive:  接收到来自服务器的ACK 0
重复   状态: 1
窗口大小: 2  阈值: 64
窗口范围: 1  ~  2
Receive:  接收到来自服务器的ACK 1
新ACK   状态: 1
窗口大小: 3  阈值: 64
窗口范围: 2  ~  4
Send:  发送包, 序列号: 3
Send:  发送包, 序列号: 4
Receive:  接收到来自服务器的ACK 2
新ACK   状态: 1
窗口大小: 4  阈值: 64
窗口范围: 3  ~  6
Send:  发送包, 序列号: 5
Send:  发送包, 序列号: 6
Receive:  接收到来自服务器的ACK 3
新ACK   状态: 1
窗口大小: 5  阈值: 64
窗口范围: 4  ~  8
Send:  发送包, 序列号: 7
Send:  发送包, 序列号: 8
```

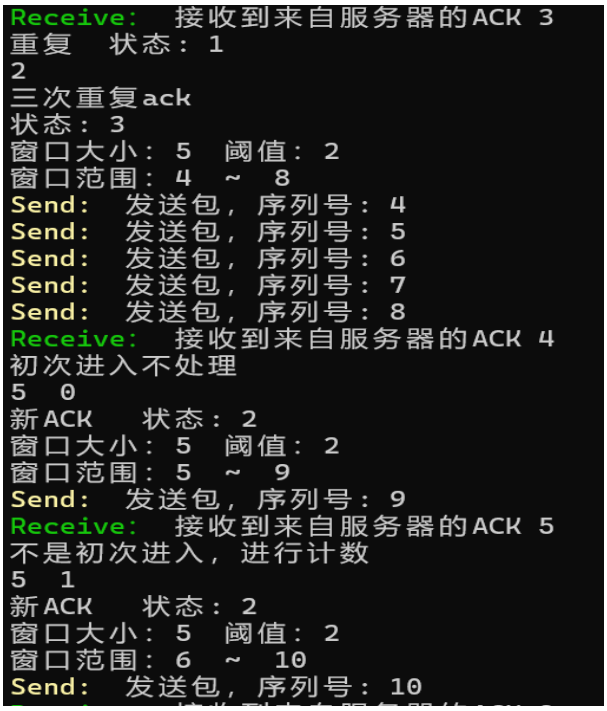
2. **三次ACK**

从下图可以看出，连续收到三个重复的ACK，会进入快速恢复状态（即状态：3），阈值变为原来窗口大小的一半（13/2=6），窗口大小变为阈值加三（6+3=9）。

```
Receive:  接收到来自服务器的ACK 11
新ACK   状态: 1
窗口大小: 13  阈值: 64
窗口范围: 12  ~  24
Send:  发送包, 序列号: 21
Send:  发送包, 序列号: 22
Send:  发送包, 序列号: 23
Send:  发送包, 序列号: 24
Receive:  接收到来自服务器的ACK 11
重复   状态: 1
窗口大小: 13  阈值: 64
窗口范围: 12  ~  24
Receive:  接收到来自服务器的ACK 11
重复   状态: 1
三次重复ack
状态: 3
窗口大小: 9  阈值: 6
窗口范围: 12  ~  20
```

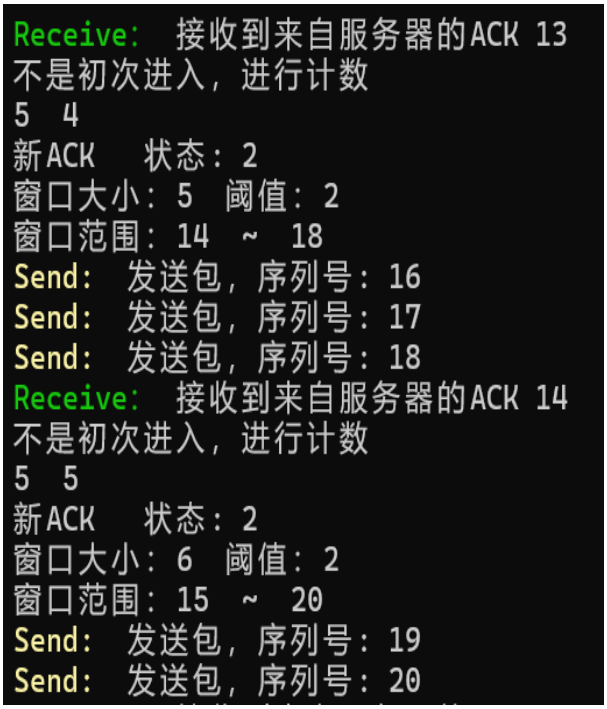
3. **快速恢复到拥塞控制**

从下图看以看出，在拥塞控制状态时，接收到一个新的ACK，就会进入拥塞控制阶段，并开始拥塞阶段的计数。




4. **拥塞控制**

当计数等于窗口大小时，即收到窗口内所有数据包的ACK，窗口大小增加 1（**cwnd+1**），如下图：




5. **累计确认**

在接收ACK时。可能会收到不连续的ACK，如下图。当收到7号ACK，代表6和7对应的数据包都已经被接收端收到，因此窗口大小会 +2 ， 下面的9和11同理。

```
Receive:  接收到来自服务器的ACK 5
新ACK   状态: 1
窗口大小: 7  阈值: 64
窗口范围: 6  ~  12
Send:  发送包, 序列号: 11
Send:  发送包, 序列号: 12
Receive:  接收到来自服务器的ACK 7
新ACK   状态: 1
窗口大小: 9  阈值: 64
窗口范围: 8  ~  16
Send:  发送包, 序列号: 13
Send:  发送包, 序列号: 14
Send:  发送包, 序列号: 15
Send:  发送包, 序列号: 16
Receive:  接收到来自服务器的ACK 9
新ACK   状态: 1
窗口大小: 11  阈值: 64
窗口范围: 10  ~  20
Send:  发送包, 序列号: 17
Send:  发送包, 序列号: 18
Send:  发送包, 序列号: 19
Send:  发送包, 序列号: 20
Receive:  接收到来自服务器的ACK 11
新ACK   状态: 1
窗口大小: 13  阈值: 64
窗口范围: 12  ~  24
```

## 超时重传

以下面为例，可以看到当出现超时重传时，窗口的大小变为 1 ，阈值变为原来窗口值的一半，并且进入3号状态（即快速恢复阶段），并对窗口内的包进行重发。

```
Receive:  接收到来自服务器的ACK 37
重复   状态: 3
2
窗口大小: 41  阈值: 3
窗口范围: 38  ~  78
Send:  发送包, 序列号: 78
Info:  超时......
窗口大小: 1  阈值: 20
窗口范围: 38  ~  38
Send:  发送包, 序列号: 38
Receive:  接收到来自服务器的ACK 38
新ACK   状态: 1
窗口大小: 2  阈值: 20
窗口范围: 39  ~  40
Send:  发送包, 序列号: 39
Send:  发送包, 序列号: 40
```

```
Receive:  接收到来自服务器的ACK 289
重复   状态: 2
1
窗口大小: 6  阈值: 3
窗口范围: 290  ~  295
Info:  超时......
窗口大小: 1  阈值: 3
窗口范围: 290  ~  290
Send:  发送包, 序列号: 290
Receive:  接收到来自服务器的ACK 290
新ACK   状态: 1
窗口大小: 2  阈值: 3
窗口范围: 291  ~  292
Send:  发送包, 序列号: 291
Send:  发送包, 序列号: 292
```

## 数据传输

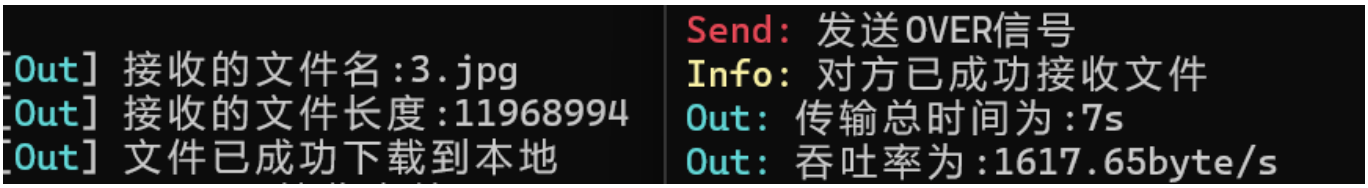
连接建立后，会给出选择，若选择Y|y，则进行文件传输，输入对应的文件名字即可，结果如下：

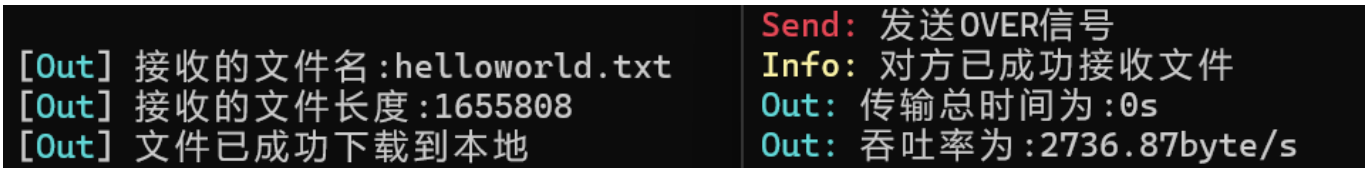
```
[Out] 接收的文件名:1.jpg
[Out] 接收的文件长度:1857353
[Out] 文件已成功下载到本地
```

```
Send: 发送OVER信号
Info: 对方已成功接收文件
Out: 传输总时间为:1s
Out: 吞吐率为:1156.51byte/s
```

```
Send: 发送OVER信号
Info: 对方已成功接收文件
Out: 传输总时间为:2s
Out: 吞吐率为:4549.22byte/s
```

```
[Out] 接收的文件名:3.jpg
[Out] 接收的文件长度:11968994
[Out] 文件已成功下载到本地
```




打开server下的文件可以看到传输得到的文件，与原有文件相同（由于检查过程中已经检查了这里不在再展示）。

### 断开连接

当文件传输结束，会给出选择，如果选择N|n，则进行四次挥手断开连接：


# 总结

总结

本次实验实现了一种拥塞控制算法---new Reno，调整拥塞窗口（cwnd）大小，并根据接收到的ACK反馈调整发送速率，以控制网络中的数据流量，避免拥塞。通过结果可以看出，传输时间低于GBN算法和停等协议，吞吐率也有所提高，这证明了拥塞控制算法的有效性。

总而言之，本实验不仅加深了我对拥塞控制算法的理解，也让我在实践中对拥塞控制算法有更深层次的掌握，也为更复杂的网络协议设计提供了实践基础。