# Lab3-3

## 1. 问题描述

在之前的实验中，已经利用数据报套接字在用户空间实现了面向连接的可靠数据传输，并且加入了流量控制机制，实现的功能主要包括：
连接建立，差错检测，GBN流水线协议，累计确认，超时重传等。
本次实验，在之前实验的基础上，增加实现了基于Reno的拥塞控制机制。

基于给定的实验要求，在原有实验的基础上，给发送端增加了拥塞控制状态机，总共有三个状态（慢启动、拥塞避免和快速恢复）。
为了简化发送端和接收端的工作，将接收端原本的累计确认改为逐次确认。发送端在接收到ACK报文之后，会判断接收到的ACK报文
是否是一个全新的序列号，如果是则根据不同的状态调整拥塞控制窗口，如果是重复ACK则考虑是否满足三次重复并进入快速恢复阶段。
发送端仍然保留了之前GBN流水线协议的超时机制，但在超时之后，增加了状态机状态的前移，一并转移到慢启动阶段，然后重发缓冲区中的报文。
需要注意，当进行了状态调整的时候，拥塞控制窗口会发生一定变化，此时需要关注拥塞控制窗口和流量控制窗口的大小，选择较小值作为发送缓冲区的大小。

## 2. 报文设计

在本次实验中，对于数据报文的格式并没有进行调整，仍采用上一次实验的数据报文格式。其中包含了发送端和接收端的端口号，确认号，序列号，标志位，长度，校验和以及数据段，整体按照16位进行对齐。

```
struct Message {
    unsigned short sourcePort{};
    unsigned short destinationPort{};
    int ack{};
    int seq{};
    unsigned short flagAndLength = 0;
    unsigned short checksum{};
    char data[MSS]{};
};
```

其中对于标志位的设计如下，由于各个数据段要按照16位对齐，因此将标志位和长度合并到同一个16位中，其中高五位为标志位，低12位表示数据段的长度。

```
| 15  | 14  | 13  | 12  | 11  |  10-0  |
| REP | FIN | SYN | ACK | FHD |  LEN   |
```

- 第15位表示当前的数据包是否为重传数据包
- 第14位表示FIN标志位
- 第13位表示SYN标志位
- 第12位表示ACK标志位

- 第11位表示当前的数据包的数据段是否为文件头信息
- 第10-0位表示数据段的长度

对于伪首部，同样参考了TCP的伪首部设计，其中包含了发送端和接收端的IP地址，zero，协议号，以及数据报的长度。

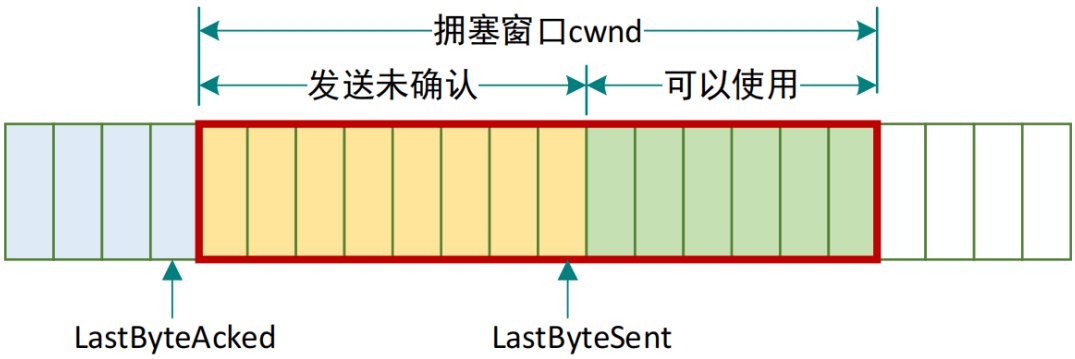
```
struct PseudoHeader {
    unsigned long sourceIP{};
    unsigned long destinationIP{};
    char zero = 0;
    char protocol = 6;
    short int length = sizeof(struct Message);
};
```

# 基于Reno算法的拥塞控制

在本次实验中，基于Reno算法实现了拥塞控制机制。

首先分析一下发送端的动作，在发送端增加了拥塞控制状态机，总共包含三个状态（慢启动，拥塞避免和快速恢复）。在原本的流量控制机制的基础上，增加了拥塞控制窗口和阈值。
当发送端接收到一个ACK报文之后，会判断这个ACK是否确认了一个新的序列号。根据不同的状态和是否确认新的序列号，调整当前的状态。
上层应用轮询当前的发送缓冲区是否有空余，如果有空余空间，则发送新的数据报文，并且更新nextseq。空余空间的判断需要综合拥塞控制窗口大小和流量控制窗口大小，选择较小值最为缓冲区大小。
当收到一个新的ACK之后，需要调整baseseq，并且将报文从缓冲区中刷出。而如果收到了重复ACK之后，则需要判断当前是否收到了三个重复ACK，满足三个重复ACK之后则进入快速恢复状态。
在Reno中沿用了之前GBN中的定时器超时机制，当收到了新的ACK之后就会刷新计时器。当超时之后，则会切换状态到慢启动状态，并且重新发送缓冲区中未确认的数据报文。

接收端的动作和之前实验的处理基本一致，但是对于接收端在3-2中采用的累计确认方式进行了调整。由于在慢启动阶段发送端的发送窗口大小只有1，而如果采用累计确认的话，可能会导致等待产生死循环。
为了简化这个处理，在本次实验中，将原本的累计确认改回了一次一应答。




$$LastByteSent - LastByteAcked \leq CongestionWindow\ (cwnd)$$

## 校验和

为了保证数据在传输过程中没有发生错误，或者是检查数据在传说过程中是否发生了错误，在实验中仿照UDP的校验和完成了差错检验。
这里的实现原理同课内理论完全一致，就不再赘述，直接上代码。

```
void Message::setChecksum(struct PseudoHeader *pseudoHeader) {
    this->checksum = 0;
    unsigned long long int sum = 0;
    for (int i = 0; i < sizeof(struct PseudoHeader) / 2; i++) {
        sum += ((unsigned short int *) pseudoHeader)[i];
    }
    for (int i = 0; i < sizeof(struct Message) / 2; i++) {
        sum += ((unsigned short int *) this)[i];
    }
    while (sum >> 16) {
        sum = (sum & 0xffff) + (sum >> 16);
    }
    this->checksum = ~sum;
}

bool Message::checksumValid(struct PseudoHeader *pseudoHeader){
    unsigned long long int sum = 0;
    for (int i = 0; i < sizeof(struct PseudoHeader) / 2; i++) {
        sum += ((unsigned short int *) pseudoHeader)[i];
    }
    for (int i = 0; i < sizeof(struct Message) / 2; i++) {
        sum += ((unsigned short int *) this)[i];
    }
    while (sum >> 16) {
        sum = (sum & 0xffff) + (sum >> 16);
    }
    return sum == 0x000ffff;
}
```

## 超时重传

在发送端，专门使用一条线程来解决超时重传的工作。这条线程负责检查当前是否触发了超时机制。
如果触发了超时机制，则需要将发送缓冲区内未被确认的报文全部重新发送。
这里要注意的事情是，由于需要对发送缓冲区进行操作，而同一时间在发送线程里可能会有向发送缓冲区增加报文的操作。
为了避免冲突，需要给发送缓冲区加锁。

由于在拥塞控制机制中，超时是对于网络阻塞的一种很好的反应，因此一旦遇到超时，则说明网络延迟很大，发生了拥塞，需要对于发送端的发送窗口大小进行调整。
当发生超时的时候，首先需要调整发送端的状态，将下一状态设置为慢启动状态。
然后调整拥塞控制窗口大小，将阈值设置为拥塞窗口大小的一般，将拥塞窗口大小设置为1。
最终对于发送缓冲区中未被确认的博文进行重传，此时需要注意的是，由于拥塞控制窗口进行了调整，因此对于发送缓冲区边界的判断需要综合考虑拥塞控制窗口大小和发送缓冲区大小。

```
void beginTimeOut() {
    std::thread resendThread([&](){
        while(true) {
            while(!timer.isTimeout()) {}
            std::string log = "[STATE] : State switch from " +
state2String(currentState) + " to " + state2String(State::SLOW_START);
            logger.addLog(log);
            // update state
            ssthresh = cwnd / 2;
            cwnd = 1;
            dupACKCount = 0;
            currentState = State::SLOW_START;
            // update nextSeq
            log = "[TIMEOUT] : Package (SEQ from : " +
std::to_string(baseSeq) + " to : " + std::to_string(int(fmin(baseSeq +
cwnd, nextSeq))) + ") re-send!";
            logger.addLog(log);
            int i = baseSeq;
            do {
                std::lock_guard<std::mutex> lockGuard(bufferMutex);
                struct Message message = sendBuffer[i-baseSeq];
                sendto(serverSocket, (char*) &message, sizeof(struct
Message), 0, (struct sockaddr*) &clientAddress, sizeof(SOCKADDR));
                logger.addLog("[RE-SEND] : " +
message2string(message));
            } while(++i <= fmin(baseSeq + cwnd, nextSeq));
        }
    });
    resendThread.detach();
}
```

## 发送端

发送端在发送数据报文的时候依旧采用GBN的流水线协议，其基本操作同上一个实验保持一致。而在接收到ACK数据包的时候，采用Reno算法引入拥塞控制机制，
通过状态机转换的方式实现拥塞控制，共分为慢启动，拥塞控制和快速恢复三个阶段。

发送端专门有一条接收线程来接收ACK报文。发送端在接收到ACK之后，当检查报文无误后，需要根据接收端确认的最大序号， 从发送缓冲区中将之前确认过的报文全部弹出。
而这个过程需要上锁，避免发送线程和接收线程同时操作发送缓冲区导致冲突。接下来就是根据接收到的被确认最大序号，更新baseSeq，也就是实现了窗口的滑动。
发送端的接收线程除了要完成上述的窗口推动工作外，还需要根据当前的状态以及收到的ACK情况调整下一阶段的状态机状态。

1. 慢启动阶段

- 当收到一个ACK报文之后，如果确认的是一个重复序列号，则将`duplicateACK`递增，并且判断`duplicateACK>3`是否成立，如果成立的话，表明当前网络中出现了大量的丢包，说明网络延迟很大存在拥塞。
  此时需要调整状态到快速恢复状态。

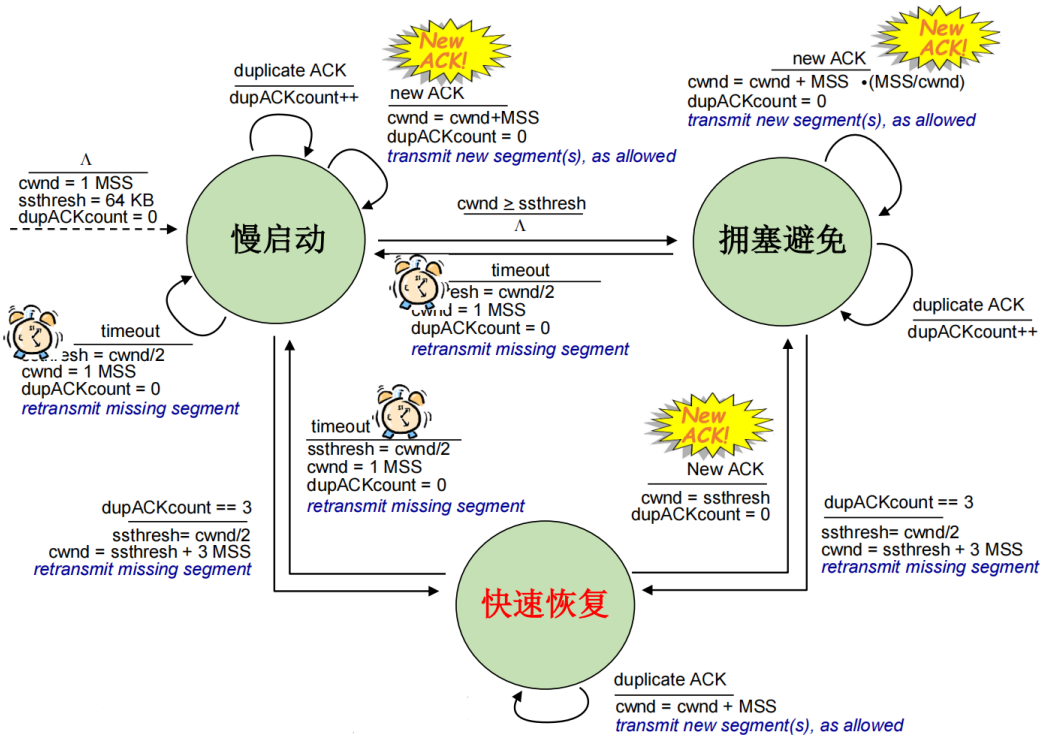
- 当收到一个ACK报文之后，如果确认的是一个新的序列号，则需要递增拥塞窗口`cwnd`，并且判断`cwnd>ssthresh`是否成立，如果成立的话则说明慢启动阶段结束，需要进入拥塞避免阶段。

2. 拥塞避免阶段

- 当收到一个ACK报文之后，如果确认的是一个重复序列号，则将`duplicateACK`递增，并且判断`duplicateACK>3`是否成立，如果成立的话，表明当前网络中出现了大量的丢包，说明网络延迟很大存在拥塞。
  此时需要调整状态到快速恢复状态。
- 当收到一个ACK报文之后，如果确认的是一个新的序列号，由于当前处于拥塞避免阶段，因此不能够直接将`cwnd`递增，而是进行小幅度增加，最终达到的效果是每一个RTT过后`cwnd`递增1。

3. 快速恢复阶段

- 当收到一个ACK报文之后，如果确认的是一个重复序列号，则将`cwnd`递增，并且立刻重传缺失的报文
- 当收到一个ACK报文之后，如果确认的是一个新的序列号，则将`cwnd`设置为`ssthresh`，并且进入拥塞避免阶段。




```
[[noreturn]] void beginRecv() {
    ......
    // update currentState
    switch(currentState) {
        case State::SLOW_START:
            if(newACK) {
                cwnd++;
                // if cwnd reaches ssthresh, switch to congestion
avoidance
                if(cwnd >= ssthresh) {
                    currentState = State::CONGESTION_AVOIDANCE;
```

```
                    std::string log = "[STATE] : State switch from " +
state2String(State::SLOW_START) + " to " +
state2String(State::CONGESTION_AVOIDANCE);
                    logger.addLog(log);
                }
                if(sendBuffer.size() > cwnd) {
                    sendto(serverSocket, (char*) &sendBuffer[cwnd-1],
sizeof(struct Message), 0, (struct sockaddr*) &clientAddress,
sizeof(SOCKADDR));
                    logger.addLog("[SEND] : " +
message2string(sendBuffer[cwnd-1]));
                    timer.start();
                }
            }
            // if dupACKCount reaches 3, switch to fast recovery
            if(dupACKCount >= 3) {
                ssthresh = cwnd / 2;
                cwnd = ssthresh + 3;
                currentState = State::FAST_RECOVERY;
                std::string log = "[STATE] : State switch from " +
state2String(State::SLOW_START) + " to " +
state2String(State::FAST_RECOVERY);
                logger.addLog(log);
                // resend the missing packet
                sendto(serverSocket, (char*) &sendBuffer[0],
sizeof(struct Message), 0, (struct sockaddr*) &clientAddress,
sizeof(SOCKADDR));
                log = "[RE-SEND] : " + message2string(sendBuffer[0]);
                logger.addLog(log);
            }
            break;
        case State::CONGESTION_AVOIDANCE:
            if(newACK) {
                newACKCount++;
                if(newACKCount >= cwnd) {
                    cwnd++;
                    newACKCount = 0;
                }
            }
            // if dupACKCount reaches 3, switch to fast recovery
            if(dupACKCount >= 3) {
                ssthresh = cwnd / 2;
                cwnd = ssthresh + 3;
                currentState = State::FAST_RECOVERY;
                std::string log = "[STATE] : State switch from " +
state2String(State::CONGESTION_AVOIDANCE) + " to " +
state2String(State::FAST_RECOVERY);
                logger.addLog(log);
                // resend the missing packet
                sendto(serverSocket, (char*) &sendBuffer[0],
sizeof(struct Message), 0, (struct sockaddr*) &clientAddress,
sizeof(SOCKADDR));
```

```
                log = "[RE-SEND] : " + message2string(sendBuffer[0]);
                logger.addLog(log);
            }
            break;
        case State::FAST_RECOVERY:
            if(newACK) {
                cwnd = ssthresh;
                dupACKCount = 0;
                currentState = State::CONGESTION_AVOIDANCE;
                std::string log = "[STATE] : State switch from " +
state2String(State::FAST_RECOVERY) + " to " +
state2String(State::CONGESTION_AVOIDANCE);
                logger.addLog(log);
            }
            else{
                cwnd++;
            }
            break;
    }
    ......
}
```

## 接收端

接收端的操作同上一次实验保持一致，只不过为了简化操作将之前的累计确认改成了一次一应当，具体的实现细节就不在本次实验中赘述，相关的实现细节可以参考上一次实验的报告。这里给出本次实验中接收端的核心代码块。

```
[[noreturn]] void beginRecv() {
    // receive message from server
    struct Message recvBuffer{};
    // write to file
    std::fstream file;
    unsigned int fileSize;
    unsigned int currentSize = 0;
    std::string filename;
    // cumulative ACK
    int ackNum = 0;
    while(true) {
        int serverAddressLength = sizeof(SOCKADDR);
        int recvLength = recvfrom(clientSocket, (char*) &recvBuffer,
sizeof(struct Message), 0, (struct sockaddr *) &serverAddress,
&serverAddressLength);
        if(recvLength > 0) {
            std::cout << "[RECV] : ";
            printMessage(recvBuffer);
            if(recvBuffer.checksumValid(&recvPseudoHeader) &&
recvBuffer.seq == EXPECT_SEQ) {
                // update current seq
                EXPECT_SEQ += 1;
                // update cumulative ACK
```

```
                ackNum += 1;
                // if message is SYN
                if(recvBuffer.isSYN()) {
                    if(!randomLoss()) {
                        sendACKSYN(recvBuffer.seq);
                    }
                }
                // if message is FIN
                else if(recvBuffer.isFIN()) {
                    if(!randomLoss()) {
                        sendACKFIN(recvBuffer.seq);
                    }
                    exitTime = 0;
                    waitExit();
                }
                // if message contains data
                else {
                    // if message contains file header
                    if(recvBuffer.isFHD()) {
                        struct FileDescriptor fileDescriptor{};
                        memcpy(&fileDescriptor, recvBuffer.data,
sizeof(struct FileDescriptor));
                        std::cout << "Receive file header: [Name:" <<
fileDescriptor.fileName << "] [Size:"
                                  << fileDescriptor.fileSize << "]" <<
std::endl;
                        fileSize = fileDescriptor.fileSize;
                        filename = fileDir + "/" +
fileDescriptor.fileName;
                        currentSize = 0;
                        // create file
                        file.open(filename, std::ios::out |
std::ios::binary);
                    }
                    else {
                        // write to file
                        file.write(recvBuffer.data,
recvBuffer.getLen());
                        currentSize += recvBuffer.getLen();
                        if(currentSize >= fileSize){
                            std::cout << "File receive success!" <<
filename << std::endl;
                            file.close();
                        }
                    }
                    // send ACK to server
                    if(ackNum >= CUM_ACK) {
                        if(!randomLoss()){
                            sendACK(recvBuffer.seq);
                        }
                        ackNum = 0;
                    }
```

```
                }
            }
            // if package is not valid or receive seq is not equal to
current seq
            // need to send last ACK again
            else {
                if(!randomLoss()) {
                    if(recvBuffer.isSYN()) {
                        sendACKSYN(EXPECT_SEQ - 1);
                    }
                    else if(recvBuffer.isFIN()) {
                        sendACKFIN(EXPECT_SEQ - 1);
                    }
                    else {
                        sendACK(EXPECT_SEQ - 1);
                    }
                }
            }
        }
    }
}
```

# 功能测试

由于在一开始代码设计的阶段并没有考虑用于测试的路由程序的影响，导致我在设计报文的时候使用了发送端和接收端的IP和端口号信息。
如果接入路由程序，发送端和接收端是分别连到路由程序的两个端口上的，也就是说其实双端之间是无感的。
这就导致我在计算校验和的时候，会因为发送端和接收端的端口号不一致导致问题。

因此在本次实验中并没有使用提供的路由程序，而是自己手动按照丢包率进行模拟丢包。模拟丢包的代码如下

```
bool randomLoss() {
    return rand() % 100 < LOSS_RATE;
}
```

接下来分别展示数据传输的效果和状态机状态转换。可以看到在下面的实验运行结果中清晰看到如下现象。

接收端的日志信息如图所示，从日志中我们可以看出，刚开始的时候发送端处于慢启动状态，因此每收到一个新的ACK之后都会直接将`cwnd`的值加1。

```
E:\project\NKU-COSC0010-Computer-Network\lab3\cmake-build-debug\server_3_3.exe
WSAStartup success!
Create socket success!
Bind success!
Waiting for client connection...
[SEND] : { Package [SYN:1] [ACK:0] [FIN:0] [ack:0] [seq:0] [Len:0] }
[RECV] : { Package [SYN:1] [ACK:1] [FIN:0] [ack:0] [seq:1] [Len:0] }
[LOG] Connection established!
[STATE] : { [state:SLOW_START] [cwnd:2] [ssthresh:16] [begin:1] [end:1] } [nextseq:1]
[LOG] : Sending file: 1.jpg size: 1857353 bytes begin!
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:1] [Len:24] }
[Seg 0 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:2] [Len:1024] }
[Seg 1 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1] [seq:2] [Len:0] }
[ACK] : Package (SEQ to : 1) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:3] [ssthresh:16] [begin:2] [end:3] } [nextseq:3]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:3] [Len:1024] }
[Seg 2 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:4] [Len:1024] }
[Seg 3 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:4] [ssthresh:16] [begin:3] [end:5] } [nextseq:5]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:5] [Len:1024] }
[Seg 4 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:6] [Len:1024] }
[Seg 5 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:4] [ssthresh:16] [begin:3] [end:7] } [nextseq:7]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:4] [ssthresh:16] [begin:3] [end:7] } [nextseq:7]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : State switch from SLOW_START to FAST_RECOVERY
```

在慢启动阶段，当接收端收到了三个重复ACK之后，就会切换到快速恢复阶段，并且将`ssthresh`的值设置为`cwnd`的一半，然后将`cwnd`的值设置为`ssthresh`的值加3。
并且可以看到立刻重传了缺失的报文。

```
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:4] [ssthresh:16] [begin:3] [end:5] } [nextseq:5]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:5] [Len:1024] }
[Seg 4 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:6] [Len:1024] }
[Seg 5 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:4] [ssthresh:16] [begin:3] [end:7] } [nextseq:7]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:SLOW_START] [cwnd:4] [ssthresh:16] [begin:3] [end:7] } [nextseq:7]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : State switch from SLOW_START to FAST_RECOVERY
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:7] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:3] [Len:1024] }
[STATE] : { [state:FAST_RECOVERY] [cwnd:5] [ssthresh:2] [begin:3] [end:8] } [nextseq:8]
```

在快速恢复阶段，可以看到，当收到重复ACK的时候，`cwnd`的值直接加1，而收到新的ACK之后，则会转向拥塞避免状态，并且将`ssthresh`的值赋值給`cwnd`。

```
[STATE] : State switch from SLOW_START to FAST_RECOVERY
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:7] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:3] [Len:1024] }
[STATE] : { [state:FAST_RECOVERY] [cwnd:5] [ssthresh:2] [begin:3] [end:8] } [nextseq:8]
[Seg 6 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:2] [seq:3] [Len:0] }
[ACK] : Package (SEQ to : 2) sent successfully!
[STATE] : { [state:FAST_RECOVERY] [cwnd:6] [ssthresh:2] [begin:3] [end:8] } [nextseq:8]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:8] [Len:1024] }
[Seg 7 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:3] [seq:4] [Len:0] }
[ACK] : Package (SEQ to : 3) sent successfully!
[STATE] : State switch from FAST_RECOVERY to CONGESTION_AVOIDANCE
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:2] [ssthresh:2] [begin:4] [end:6] } [nextseq:9]
```

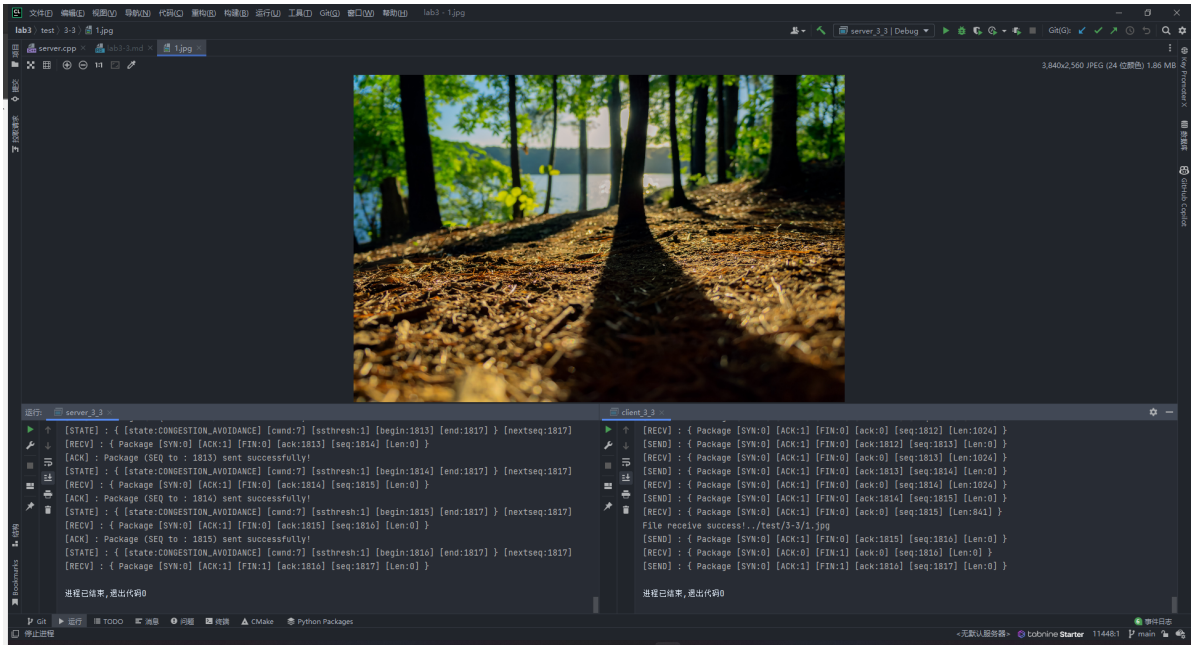
在拥塞避免阶段，每收到一个新的ACK，`cwnd`的值就会加1，但是每次加1的值不是1，而是`cwnd`的值除以`cwnd`的值。也就是每一个RTT之后，`cwnd`的值都会加1。

```
[ACK] : Package (SEQ to : 10) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:3] [ssthresh:1] [begin:11] [end:12] } [nextseq:12]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:12] [Len:1024] }
[Seg 11 in 1814]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:13] [Len:1024] }
[Seg 12 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:11] [seq:12] [Len:0] }
[ACK] : Package (SEQ to : 11) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:3] [ssthresh:1] [begin:12] [end:14] } [nextseq:14]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:14] [Len:1024] }
[Seg 13 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:12] [seq:13] [Len:0] }
[ACK] : Package (SEQ to : 12) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:3] [ssthresh:1] [begin:13] [end:15] } [nextseq:15]
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:15] [Len:1024] }
[Seg 14 in 1814]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:13] [seq:14] [Len:0] }
[ACK] : Package (SEQ to : 13) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:4] [ssthresh:1] [begin:14] [end:16] } [nextseq:16]
```

无论在那个状态，一旦遇到超时，都是转移到慢启动阶段，并且将`ssthresh`的值设置为`cwnd`的一半，然后将`cwnd`的值设置为1。

```
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:3] [ssthresh:1] [begin:7] [end:10] } [nextseq:10]
[STATE] : State switch from CONGESTION_AVOIDANCE to SLOW_START
[TIMEOUT] : Package (SEQ from : 7 to : 8) re-send!
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:7] [Len:1024] }
[RE-SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:8] [Len:1024] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:7] [seq:8] [Len:0] }
[ACK] : Package (SEQ to : 7) sent successfully!
[STATE] : State switch from SLOW_START to CONGESTION_AVOIDANCE
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:2] [ssthresh:1] [begin:8] [end:10] } [nextseq:10]
```

最后展示最终的图像传输结果


server_3_3
client_3_3
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:7] [ssthresh:1] [begin:1813] [end:1817] } [nextseq:1817]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1813] [seq:1814] [Len:0] }
[ACK] : Package (SEQ to : 1813) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:7] [ssthresh:1] [begin:1814] [end:1817] } [nextseq:1817]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1814] [seq:1815] [Len:0] }
[ACK] : Package (SEQ to : 1814) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:7] [ssthresh:1] [begin:1815] [end:1817] } [nextseq:1817]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1815] [seq:1816] [Len:0] }
[ACK] : Package (SEQ to : 1815) sent successfully!
[STATE] : { [state:CONGESTION_AVOIDANCE] [cwnd:7] [ssthresh:1] [begin:1816] [end:1817] } [nextseq:1817]
[RECV] : { Package [SYN:0] [ACK:1] [FIN:1] [ack:1816] [seq:1817] [Len:0] }
进程已结束,退出代码0
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:1812] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1812] [seq:1813] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:1813] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1813] [seq:1814] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:1814] [Len:1024] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1814] [seq:1815] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:0] [seq:1815] [Len:841] }
File receive success!../test/3-3/1.jpg
[SEND] : { Package [SYN:0] [ACK:1] [FIN:0] [ack:1815] [seq:1816] [Len:0] }
[RECV] : { Package [SYN:0] [ACK:0] [FIN:1] [ack:0] [seq:1816] [Len:0] }
[SEND] : { Package [SYN:0] [ACK:1] [FIN:1] [ack:1816] [seq:1817] [Len:0] }
进程已结束,退出代码0