

# 计算机网络lab3-3

## 实验内容与基本概述

### 协议设计与C++编程实现

#### 1 Reno算法概述

#### 2 三种状态的定义和切换实现

##### 2.1 控制Reno算法的变量定义

##### 2.2 接收线程函数对于慢启动状态的窗口调整处理

##### 2.3 滑动窗口阻塞函数对于拥塞避免状态的处理

##### 2.4 接收线程函数对于快恢复的实现

##### 2.5 慢开始门限对于状态切换的作用

##### 2.6 超时重传检测与状态跳转

##### 2.7 程序设计流程图展示

### 程序运行测试与传输结果分析

#### 1 自行设置丢包率

#### 2 传输测试：3.png

姓名：罗瑞

学号：2210529

专业：密码科学与技术

## 实验内容与基本概述

- ✓ 在实验3-2的基础上，本次实验实现了一种基于TCP Reno算法的拥塞控制机制。由于IP层不提供显式的网络拥塞反馈，TCP采用端对端拥塞控制，通过动态调整拥塞窗口（cwnd）感知网络路径的拥塞情况。当路径未发生拥塞时，发送方增加发送速率，反之则降低速率。为提升传输性能，本实验结合实际网络环境，对Reno算法进行了优化，尤其在超时重传时采用更保守的窗口退化策略，以提高传输效率并优化拥塞控制效果。

# 协议设计与C++编程实现

## 1 Reno算法概述

### ✓ TCP Reno的整体工作流程

#### 1. 连接建立时：

- `cwnd` 初始化为 `1MSS` ；
- 开始慢启动。

#### 2. 慢启动期间：

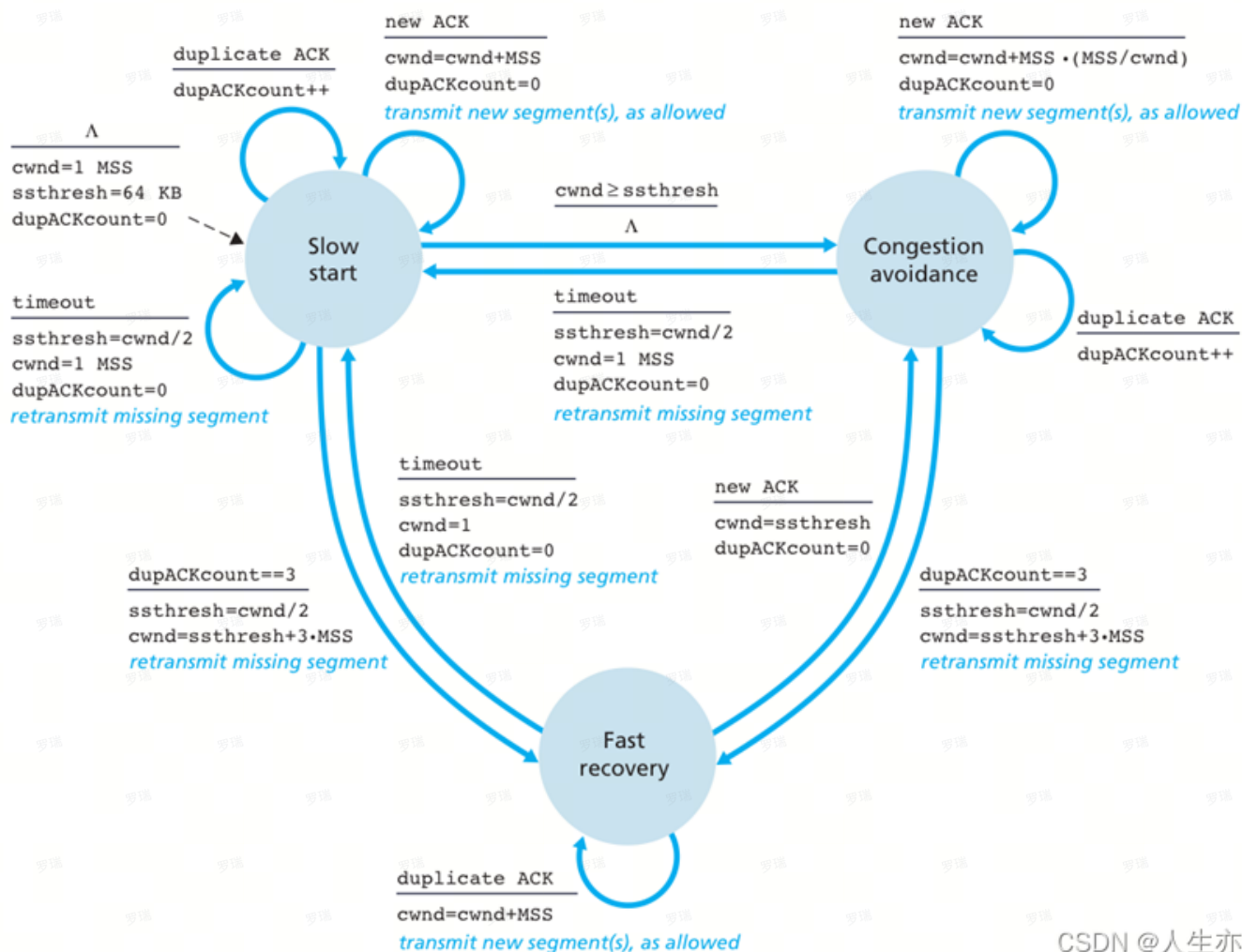
- `cwnd` 以指数增长方式探测网络带宽；
- 若达到 `ssthresh` 或出现超时/冗余ACK，则结束慢启动。

#### 3. 拥塞避免期间：

- `cwnd` 以线性方式增长；
- 若出现超时/冗余ACK，则进入慢启动或快速恢复。

#### 4. 快速恢复期间：

- 利用冗余ACK快速恢复数据传输；
- 结束后进入拥塞避免。



CSDN @人生亦逝

图1：Reno算法的FSM展示

## ✓ Reno拥塞控制算法的具体每一个模式的工作内容与切换条件：

### 1. 慢启动（Slow Start）

#### 概述

慢启动是TCP连接建立后最初用于调整发送速率的阶段。其目标是通过指数增长的方式快速探测路径的可用带宽，同时避免初始发送速率过高导致的拥塞。

#### 机制

##### • 初始状态：

拥塞窗口（ $cwnd$ ）通常被初始化为  $1MSS$ （Maximum Segment Size，单个TCP报文段的最大大小），发送方仅能发送一个报文段。

##### • $cwnd$ 增长方式：

每接收到一个确认（ACK）， $cwnd$  增加1个  $MSS$ ，即每次RTT内的发送速率翻倍。例如：

- 初始  $cwnd = 1MSS$ ，发送1个报文段，确认后， $cwnd = 2MSS$ ；

- 发送2个报文段，确认后， $cwnd = 4MSS$ ；
- 发送4个报文段，确认后， $cwnd = 8MSS$ 。

- **速率增长特点：**

慢启动阶段的发送速率呈**指数增长**。

## 结束条件

慢启动阶段的结束条件有以下三种：

1. **超时 (Timeout)：**

若在慢启动过程中发生超时，则认为发生了网络拥塞，此时：

- 设置慢启动阈值  $ssthresh = cwnd / 2$ ；
- 将  $cwnd$  重置为  $1MSS$ ，重新开始慢启动。

2. **达到慢启动阈值：**

若  $cwnd \geq ssthresh$ ，认为网络可能接近拥塞，此时切换至**拥塞避免阶段**。

3. **三次冗余ACK：**

若接收到某个报文的三次冗余ACK (Duplicate ACKs)，触发**快速重传**，进入**快速恢复阶段**：

- 更新  $ssthresh = cwnd / 2$ ；
- 将  $cwnd = ssthresh + 3MSS$ ，为快速恢复做准备。

4. **拥塞避免 (Congestion Avoidance)**

## 概述

拥塞避免阶段的主要目标是以更谨慎的方式增加发送速率，逐步探测网络的最大可用带宽。此阶段发送速率的增长从指数级别下降为线性增长。

## 机制

- **进入条件：**

当  $cwnd \geq ssthresh$  时，进入拥塞避免阶段。

- **$cwnd$  增长方式：**

在拥塞避免阶段， $cwnd$  以每RTT增加  $1MSS$  的方式增长，即线性增长。

- 若当前  $cwnd$  能够发送10个报文段，则每接收到1个ACK， $cwnd$  增加  $1/10$   $MSS$ 。
- 收到所有报文段的确认后， $cwnd$  总共增加  $1MSS$ 。

## 结束条件

拥塞避免阶段结束的条件如下：

### 1. 超时 (Timeout) :

若发生超时，认为网络出现严重拥塞，此时：

- 设置  $ssthresh = cwnd / 2$  ;
- 将  $cwnd$  重置为  $1MSS$  ，重新进入慢启动阶段。

### 2. 三次冗余ACK:

若接收到三次冗余ACK，进入快速恢复阶段：

- 更新  $ssthresh = cwnd / 2$  ;
- 将  $cwnd = ssthresh + 3MSS$  ，准备快速恢复。

### 3. 快速恢复 (Fast Recovery)

#### 概述

快速恢复是TCP Reno的推荐部分，旨在快速恢复数据传输能力，避免直接退回慢启动阶段带来的性能损失。其特点是在出现三次冗余ACK时，不立即重置  $cwnd$  ，而是通过冗余ACK逐步增加发送窗口。

#### 机制

- 进入条件：

若收到三次冗余ACK，表明丢失的报文段较少，进入快速恢复阶段。

- $cwnd$ 调整方式：

- 将  $ssthresh = cwnd / 2$  ;
- 设置  $cwnd = ssthresh + 3MSS$  ，用于弥补三次冗余ACK所对应的报文段。

- 冗余ACK的处理：

每接收到1个冗余ACK，  $cwnd$  增加  $1MSS$  ，发送新的报文段，直到收到新的ACK。

#### 结束条件

快速恢复阶段在以下两种情况下结束：

### 1. 超时 (Timeout) :

若快速恢复阶段发生超时：

- 设置  $ssthresh = cwnd / 2$  ;
- 将  $cwnd$  重置为  $1MSS$  ，重新进入慢启动阶段。

### 2. 收到新ACK:

若接收到新确认（非冗余ACK），表明丢失的报文段已恢复：

- 将  $cwnd = ssthresh$  ;
- 进入拥塞避免阶段。


## 2 三种状态的定义和切换实现

### 2.1 控制Reno算法的变量定义

ssthresh	全局变量：慢开始门限，这里初始值设为16
acked[]	记录ACK次数的数组，便于进行识别快重传
accumulate_point	一个RTT发送窗口内的累积确认指针
windowlength h	全局变量：拥塞窗口大小，根据Reno算法，初始为慢启动模式，值为1
windowtop	拥塞窗口底部指针
windowbase	拥塞窗口顶部指针
RenoMode	全局标志位变量，用来标识当前的Reno的FSM的模式，值为0代表慢启动模式，值为1代表拥塞避免模式。

### 2.2 接收线程函数对于慢启动状态的窗口调整处理

协议设计：

- 

慢启动是TCP连接建立后最初用于调整发送速率的阶段。其目标是通过指数增长的方式快速探测路径的可用带宽，同时避免初始发送速率过高导致的拥塞。

**机制**

**初始状态：**

拥塞窗口（`cwnd`）通常被初始化为 `1MSS`（Maximum Segment Size，单个TCP报文段的最大大小），发送方仅能发送一个报文段。

**`cwnd`增长方式：**

每接收到一个确认（ACK），`cwnd` 增加1个 `MSS`，即每次RTT内的发送速率翻倍。

在接收到新的ACK时，直接增加窗口大小 `cwnd`（`windowlength`），而不是等待窗口内全部确认。在 `if (accumulate_point != ack)` 的分支中，加入 `windowlength += 1`，每次接收到新的ACK都会触发窗口调整。

**速率增长特点：**

慢启动阶段的发送速率呈**指数增长**。

## 结束条件

慢启动阶段的结束条件有以下三种：

### 超时 (Timeout) :

若在慢启动过程中发生超时，则认为发生了网络拥塞，此时：

设置慢启动阈值 `ssthresh = cwnd / 2` ;

将 `cwnd` 重置为 `1MSS` ，重新开始慢启动。

### 达到慢启动阈值：

若 `cwnd >= ssthresh` ，认为网络可能接近拥塞，此时切换至拥塞避免阶段。

### 三次冗余ACK：

若接收到某个报文的三次冗余ACK (Duplicate ACKs) ，触发快速重传，进入快速恢复阶段：

更新 `ssthresh = cwnd / 2` ;

将 `cwnd = ssthresh + 3MSS` ，为快速恢复做准备。

## 代码实现：

```
1 int receive() {
2     while (true)
3     {
4         UDP_DATAGRAM recv;
5         memset(&recv, 0, sizeof(UDP_DATAGRAM));
6         recvfrom(sock, (char*)&recv, sizeof(UDP_DATAGRAM), 0,
7             (SOCKADDR*)&server, &l);
8
9         if (recv.getstate() & REQ)
10        {
11            std::cout << "发送序列为" << recv.getack() << "的报文传输出现数据错误！"
12            << std::endl;
13            continue;
14        }
15        else if (recv.getstate() & ACK && !(recv.getstate() & SYN))
16        {
17            int ack = recv.getack();
18            if (ack == 65535)
19                ack = -1;
20
21            if (accumulate_point != ack)
22            {
23                for (int j = accumulate_point + 1; j <= ack; j++)
24                    acked[j] += 1; // 更新状态，表示累积确认
```

```

23
24      // 更新累积确认指针
25      accumulate_point = ack;
26
27      // 调整窗口大小仅在慢启动模式
28      if (!RenoMode) {
29          windowlength += 1;
30          std::cout << YELLOW << "接收到新的ACK，窗口大小增加：" <<
windowlength << "\n" << RESET;
31      }
32  }
33  else
34  {
35      std::cout << "重复ACK!!! 序列号：" << ack << " 的报文重复ACK！"
<< std::endl;
36      acked[accumulate_point] += 1;
37  }
38  }
39
40      // 检测快重传函数：关键
41      if (acked[accumulate_point] == 4) // 所谓事不过三，过三速速重传！
42      {
43          std::cout << GREEN << "三次重复ack，进行快重传，并调整窗口大小与慢启动门
限阈值.\n" << RESET;
44
45          // 快重传该报文
46          quickresend(accumulate_point + 1);
47
48          // 调整窗口大小和慢启动门限
49          ssthresh = (windowlength + 1) / 2; // 向上取整除以 2
50          windowlength = ssthresh;
51          std::cout << YELLOW << "窗口调整为：" << windowlength << "，慢启动门限
调整为：" << ssthresh << "\n" << RESET;
52
53          RenoMode = 1; // 切换到拥塞避免模式
54      }
55
56      // 线程退出条件：若滑动窗口到最后，且检查窗口内的是否都已确认
57      bool exitjudge = true;
58      for (int i = windowbase; i <= udp_num - 1; i++)
59      {
60          if (!acked[i])
61              exitjudge = false;
62      }
63
64      if (exitjudge)
65          return 1; // 如果确认号是最后一个分片，退出接收线程

```



```
66     }  
67 }  
68
```

## 2.3 滑动窗口阻塞函数对于拥塞避免状态的处理

滑动窗口阻塞函数 `window_sending` 的核心功能是管理数据传输的滑动窗口机制，结合拥塞控制的 `Reno` 算法，对传输窗口大小进行动态调整。以下是它在拥塞避免状态下的工作机制解释：

协议设计：

### ✓ 1. 滑动窗口的整体逻辑

- 滑动窗口基础：
  - `windowbase` : 当前窗口的起点（底部指针）。
  - `windowtop` : 当前窗口的终点（顶部指针）。
  - `windowlength` : 当前窗口的大小，即可以未确认的数据分组的数量。
- 窗口滑动条件：
  - 如果当前窗口顶部的分组（由 `windowtop` 指针表示）已经被确认（即 `acked>windowtop] == true`），窗口滑动到下一个分组。
  - 窗口的大小由 `windowlength` 决定，这个大小动态调整，受拥塞控制算法影响。

### 2. 拥塞避免模式处理

- 在拥塞避免状态（`RenoMode == 1`），窗口的调整遵循线性增长规则。

### 3. 逻辑细节：

- 线性增长：
  - `windowlength += 1` : 窗口大小每次确认时，增加一个单位。这种增长速度相较于慢启动的指数增长要缓慢得多。
  - 线性增长的目的是确保网络接近带宽极限时，不会因为窗口的快速膨胀引发拥塞。
- 窗口滑动操作：
  - `windowbase = windowtop + 1` : 当窗口的顶部（`windowtop`）被确认后，窗口的底部滑动到下一分组。
  - `windowtop = windowbase + windowlength - 1` : 窗口顶部调整为新的窗口大小，同时检查是否超出分组总数的限制。
- 打印日志：

```
print_yellow_message(windowbase, windowtop):
```

打印当前窗口的底部（`windowbase`）和顶部（`windowtop`）的位置，用于监控窗口移动的状态。

#### 4. 结合 Reno 窗口调整函数

滑动窗口机制调用了 `Reno` 函数，用于调整窗口大小。

- 拥塞避免状态的 `Reno` 调整：

- 当滑动窗口顶部的分组被确认后，窗口大小在拥塞避免状态下会按线性规则增加。
- `windowlength` 的增加确保滑动窗口可以逐渐扩大，同时避免过快导致拥塞。

#### 5. 调整后的窗口行为：

- 滑动窗口在顶部确认后，会尝试滑动到下一个分组，调整后的窗口大小（`windowlength`）决定了可以传输的数据分组数量。

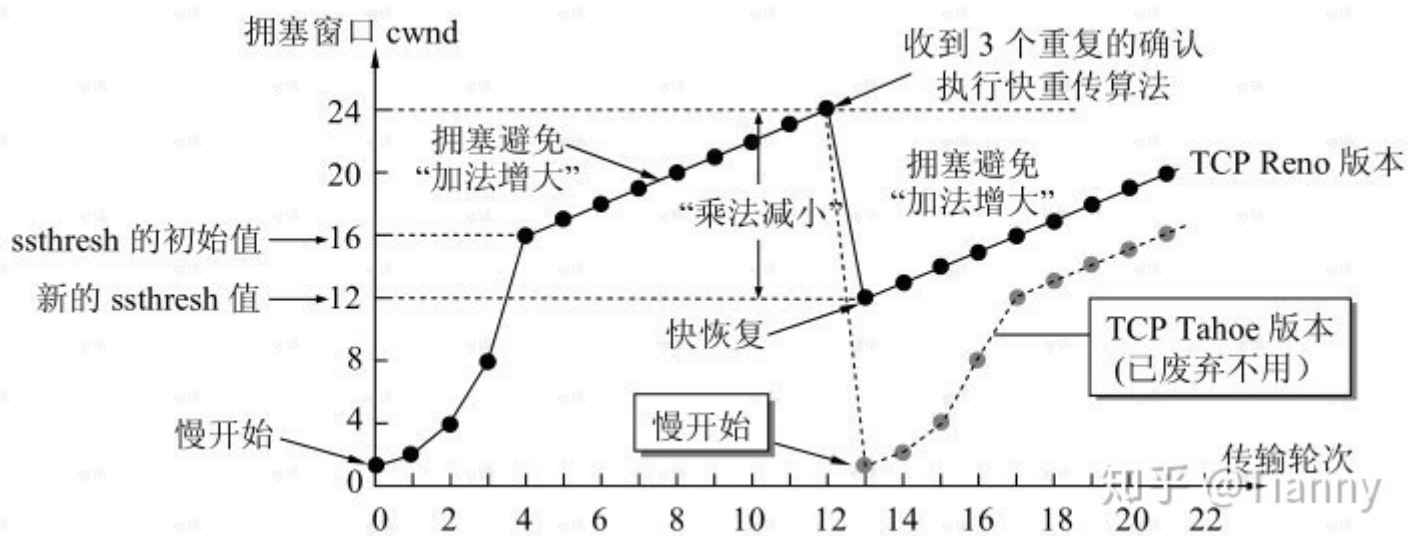


图2：拥塞窗口实际大小随传输模式的变化

代码实现：

```
1 //阻塞函数处理滑动窗口，一直循环到所有数据报文发送且接收确认完毕
2 void windowsending() {
3     while (true) {
4         //前面窗口内全部确认，进入下一分组
5         if (acked>windowtop)
6         {
7             Reno(); //窗口大小和慢开始门限调整
8         }
9         if (acked[udp_num-1])
```

```

10         break; //发送序列全部确认完毕
11
12         //滑动窗口进入下一个分组
13         windowbase=windowtop+1;
14         //根据windowlength来调整窗顶指针
15         windowtop = windowbase + windowlength-1;
16         if (windowtop > udp_num - 1)
17             windowtop= udp_num - 1;
18         //打印当前窗口底和顶位置
19         print_yellow_message(windowbase, windowtop);
20     }
21
22 }
23 return;
24 }
25
26 // 拥塞控制函数，核心算法
27 void Reno() {
28     if (!RenoMode) // 慢启动
29     {
30         std::cout << GREEN << "Mode: 慢启动\n" << RESET;
31         if (2 * windowlength > ssthresh) // 达到阈值，进入拥塞避免状态
32         {
33             RenoMode = 1; // 调整模式标志位
34             windowlength = ssthresh;
35             std::cout << GREEN << "Threshold reached, 切换到拥塞避免模式.\n" <<
RESET;
36         }
37         else {
38             //windowlength *= 2;
39             //实际上在慢启动状态下接收时，每接收一个ACK，长度则加一。是实现慢启动状态下
指数增长的微观实现。
40         }
41     }
42     else // 拥塞避免
43     {
44         std::cout << GREEN << "Mode: 拥塞避免\n" << RESET;
45         windowlength += 1;
46     }
47
48     // 打印当前窗口大小
49     std::cout << YELLOW << "当前窗口大小: " << windowlength << "\n" << RESET;
50     std::cout << YELLOW << "当前门限大小:" << ssthresh << "\n" << RESET;
51 }
52

```

## 2.4 接收线程函数对于快恢复的实现

快恢复机制实现了基于 TCP 的 **快速重传与快速恢复**（Fast Retransmit and Fast Recovery）算法协议设计：



- **快重传阶段**：检测到三次重复 ACK 后，立即重发丢失的数据报文。
- **快恢复阶段**：调整窗口大小为门限值，避免进入慢启动，直接切换到拥塞避免模式。
- **核心目标**：快速修复丢包问题，减少传输中断时间，同时有效防止网络拥塞的进一步恶化。

具体来说：

### 1. 检测触发条件

当接收到**三次重复 ACK**时，即 `acked[accumulate_point] == 4` 条件满足时，程序认为网络传输过程中发生了丢包。

重复 ACK 表示某个分组的接收方没有收到预期数据，但却收到了后续分组的数据。因此，发送方在收到三次重复 ACK 后，会**立即进行快速重传**，而不等待传统的超时机制。

### 2. 快速重传

通过调用 `quickresend(accumulate_point + 1)` 函数，重传丢失的数据分组。重传的逻辑包括以下几步：

#### 1. 判断分组类型：

- 检查丢失的数据报是否为最后一个分组（以处理不同长度的数据报）。
- 如果是中间分组，则正常填充固定大小的数据；如果是最后一个分组，则填充剩余数据。

#### 2. 设置报文状态：

- 将数据报设置为同步状态（SYN），并计算校验和，确保传输的正确性。

#### 3. 发送重传数据：

- 使用 `sendto` 函数将丢失的数据报发送到接收端。
- 如果发送失败，则记录错误并中止操作。

通过快速重传机制，丢失的数据可以迅速补发，避免等待超时重传，从而降低了传输延迟。

### 3. 调整窗口大小与门限

在丢包后，为了防止网络拥塞进一步恶化，程序会调整传输控制参数：

- 慢启动门限（`ssthresh`）：

设置为当前窗口大小的一半（向上取整后再加 3），以减少网络负载，同时保留一定的传输能力。

- 窗口大小（`windowlength`）：

设置为与新门限相同的值。窗口的缩小可以减缓网络的拥塞程度。

通过上述调整，程序不会完全缩小窗口（如传统慢启动中的窗口大小重置为 1），而是直接进入拥塞避免模式。

## 4. 进入拥塞避免模式

调整完窗口和门限后，程序将标志位 `RenoMode` 设置为 1，表示从慢启动切换到拥塞避免模式。在这种模式下，窗口大小将以线性方式缓慢增加，避免进一步造成网络拥塞。

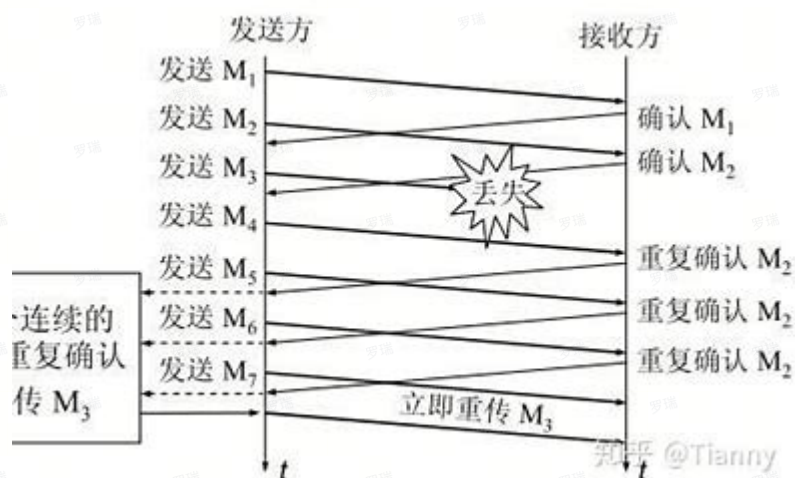


图3：快恢复的部分实现：快重传过程示意图

代码实现：

```
1 //快重传函数
2 int quickresend(int tri_ack) {
3     int seq = tri_ack + 1; //ack重复三次的变量说明下一个数据报丢包了
4     UDP_DATAGRAM udp_now;
5     memset(&udp_now, 0, sizeof(UDP_DATAGRAM));
6     if (seq != udp_num - 1)
7     {
8         memcpy(udp_now.getmessage(), buffer[seq], UDP_DATAGRAM_SIZE);
9         udp_now.setlength(UDP_DATAGRAM_SIZE);
10        udp_now.setseq(seq);
11    }
12    else
13    {
```

```

14     memcpy(udp_now.getmessage(), buffer[seq], rest);
15     udp_now.setlength(rest);
16     udp_now.setseq(seq);
17 }
18 udp_now.setstate(SYN);
19 udp_now.calculate_checksum(); // 计算并设置校验和
20
//=====
//=====
21 //发送数据报
22 if (sendto(sock, (char*)&udp_now, sizeof(udp_now), 0, (SOCKADDR*)&server,
    1) == SOCKET_ERROR) {
23     cout << "序列为" << seq << "的数据报文发送错误, 错误码: " <<
        WSAGetLastError() << endl;
24     return 0;
25 }
26 //cout << "序列为" << seq << "的数据报文发送成功!" << endl;
27
28 return 1;
29 }
30
31 void receive(){
32     .....
33     //=====
34     //检测快重传函数: 关键
35     if (acked[accumulate_point] == 4) //所谓事不过三, 过三速速重传!
36     {
37         std::cout << GREEN << "三次重复ack, 进行快重传, 并调整窗口大小与慢启动门限阈
            值.\n" << RESET;
38         //快重传该报文
39         quickresend(accumulate_point + 1);
40         //调整窗口大小和慢启动门限
41         //向上取整
42         ssthresh = (windowlength + 1) / 2 + 3; // 向上取整除以 2 再加三
43         windowlength = ssthresh;
44
45         RenoMode = 1; //进入拥塞避免状态
46     }
47
48     .....
49 }

```

## 2.5 慢开始门限对于状态切换的作用

协议设计:

### ✓ 触发条件:

慢启动模式下，窗口大小按照指数增长规则进行扩展。如果窗口大小即将超越阈值（ $2 * windowlength > ssthresh$ ），表示当前网络可能会面临拥塞风险。

### 切换到拥塞避免模式:

在拥塞避免模式下，窗口大小将按照线性增长规则逐步扩展，即经过每个RTT，窗口大小以固定步长（+1）增长。

### 代码实现:

```
1 // 拥塞控制函数，核心算法
2 void Reno() {
3     if (!RenoMode) // 慢启动
4     {
5         std::cout << GREEN << "Mode: 慢启动\n" << RESET;
6         if (2 * windowlength > ssthresh) // 达到阈值，进入拥塞避免状态
7         {
8             RenoMode = 1; // 调整模式标志位
9             windowlength = ssthresh;
10            std::cout << GREEN << "Threshold reached, 切换到拥塞避免模式.\n" <<
            RESET;
11        }
12        else {
13            //windowlength *= 2;
14        }
15    }
16    else // 拥塞避免
17    {
18        std::cout << GREEN << "Mode: 拥塞避免\n" << RESET;
19        windowlength += 1;
20    }
21
22    // 打印当前窗口大小
23    std::cout << YELLOW << "当前窗口大小: " << windowlength << "\n" << RESET;
24    std::cout << YELLOW << "当前门限大小: " << ssthresh << "\n" << RESET;
25 }
```

## 2.6 超时重传检测与状态跳转

### 协议设计:

✓ 发送数据报文: 将特定序列号的 UDP 数据报发送到服务器。



**处理确认机制：**检测报文是否被确认（`acked[seq]` 表示是否确认）。

**超时重传：**如果在规定时间内（`timelimit_udp`）内未收到确认，触发**超时重传机制**。

这里若发生超时，此时：

- 设置 `ssthresh = cwnd / 2`；
- 将 `cwnd` 重置为 `1MSS`，重新进入慢启动阶段。

**拥塞控制：**在超时重传时，调整拥塞控制参数，切换到拥塞避免状态。

## 代码实现：


```
1 //发送一个数据报的核心函数，处理方式：丢包&差错不会引发确认，最终会触发超时重传，
2 // 注意每一个窗口内的数据报不能互相影响，每一个数据报的处理都应该是一个线程
3 int udpdatagramcopyandsend(int seq) {
4     UDP_DATAGRAM udp_now;
5     memset(&udp_now, 0, sizeof(UDP_DATAGRAM));
6     if (seq != udp_num - 1)
7     {
8         memcpy(udp_now.getmessage(), buffer[seq], UDP_DATAGRAM_SIZE);
9         udp_now.setlength(UDP_DATAGRAM_SIZE);
10        udp_now.setseq(seq);
11    }
12    else
13    {
14        memcpy(udp_now.getmessage(), buffer[seq], rest);
15        udp_now.setlength(rest);
16        udp_now.setseq(seq);
17    }
18    udp_now.setstate(SYN);
19    udp_now.calculate_checksum(); // 计算并设置校验和
20
21    //=====
22    //发送数据报
23    if (sendto(sock, (char*)&udp_now, sizeof(udp_now), 0, (SOCKADDR*)&server,
24        1) == SOCKET_ERROR) {
25        cout << "序列为" << seq << "的数据报文发送错误，错误码：" <<
26        WSAGetLastError() << endl;
27        return 0;
28    }
29    //cout << "序列为" << seq << "的数据报文发送成功！" << endl;
30
31    // 注意差错的报文也是未确认，启动超时检测重传机制
```



```

29     int start = clock(); // 记录发送起始时间
30
31     while (true)
32     {
33         int end = clock(); // 获取当前时间
34
35         // 等待报文确认, 超时重传
36         if (end - start > timelimit_udp)
37         {
38             //向上取整
39             ssthresh = (windowlength + 1) / 2 ; // 向上取整除以 2
40             windowlength = ssthresh;
41
42             RenoMode = 1; //进入拥塞避免状态
43
44             print_red_message(seq);
45             //cout << "序列为" << seq << "的报文超时未确认, 触发重传机制" << endl;
46             //发送数据报
47             if (sendto(sock, (char*)&udp_now, sizeof(udp_now), 0,
48 (SOCKADDR*)&server, 1) == SOCKET_ERROR) {
49                 //cout << "序列为" << seq << "的数据报文发送错误, 错误码: " <<
50                 WSAGetLastError() << endl;
51                 return 0;
52             }
53             else
54                 //cout << "序列为" << seq << "的数据报文发送成功!" << endl;
55                 start = clock(); // 重置起始时间
56         }
57
58         //线程退出的条件
59         if (acked[seq])
60             return 1; // 返回 1 表示消息发送成功并被确认
61
62         //Sleep(50);
63     }
64 }

```

 在实际网络环境中, 为了提高拥塞控制的适应性并减轻突发性网络异常对性能的影响, 所以上述代码实现实际上采取了**保守的超时重传策略**。当发生超时重传时, 并未直接回到慢启动模式并将窗口大小 (`windowlength`) 设置为 1, 而是采用**逐步退化策略**。具体而言, 通过将窗口大小减半 (`windowlength = ssthresh = windowlength / 2`), 并直接切换到**拥塞避免模式** (`RenoMode = 1`)。

这种设计避免了因个别报文的突发超时而导致窗口大小急剧收缩至最小值，从而有效缓解了性能的过度下降。同时，通过减半窗口大小，能够迅速降低网络的发送速率，缓解拥塞状态，并在拥塞避免模式下以线性增长的方式逐步恢复发送速率。

### 逐步退化策略的优点：

1. **平衡响应敏感性与网络吞吐量：**直接回到慢启动模式（窗口大小为 1）会导致传输速率骤降，严重影响吞吐量。而逐步退化策略能够快速响应拥塞并尽量减小对网络性能的影响。
2. **应对网络环境中的突发性超时：**网络中的丢包或超时可能并非由网络拥塞引起，例如链路层错误或瞬时抖动。逐步退化避免了不必要的激进反应。

**拥塞避免模式的合理性：**在窗口减半后，立即进入拥塞避免模式（`RenoMode = 1`），表明系统假设当前网络状况并非极度拥塞，而是中度拥塞或暂时性波动。通过线性增长的窗口调整方式，能够逐渐恢复到合理的发送速率，而不会因过于激进的指数增长引发新的拥塞。

**对经典 TCP Reno 的改进：**相较于传统 TCP Reno，在发生超时后直接重置窗口大小为 1 的做法，该实现更贴近现代网络优化思想，如 TCP Vegas 或 TCP Cubic 中的保守调整机制，尤其适合在高吞吐量、低丢包率的环境下运行。

## 2.7 程序设计流程图展示

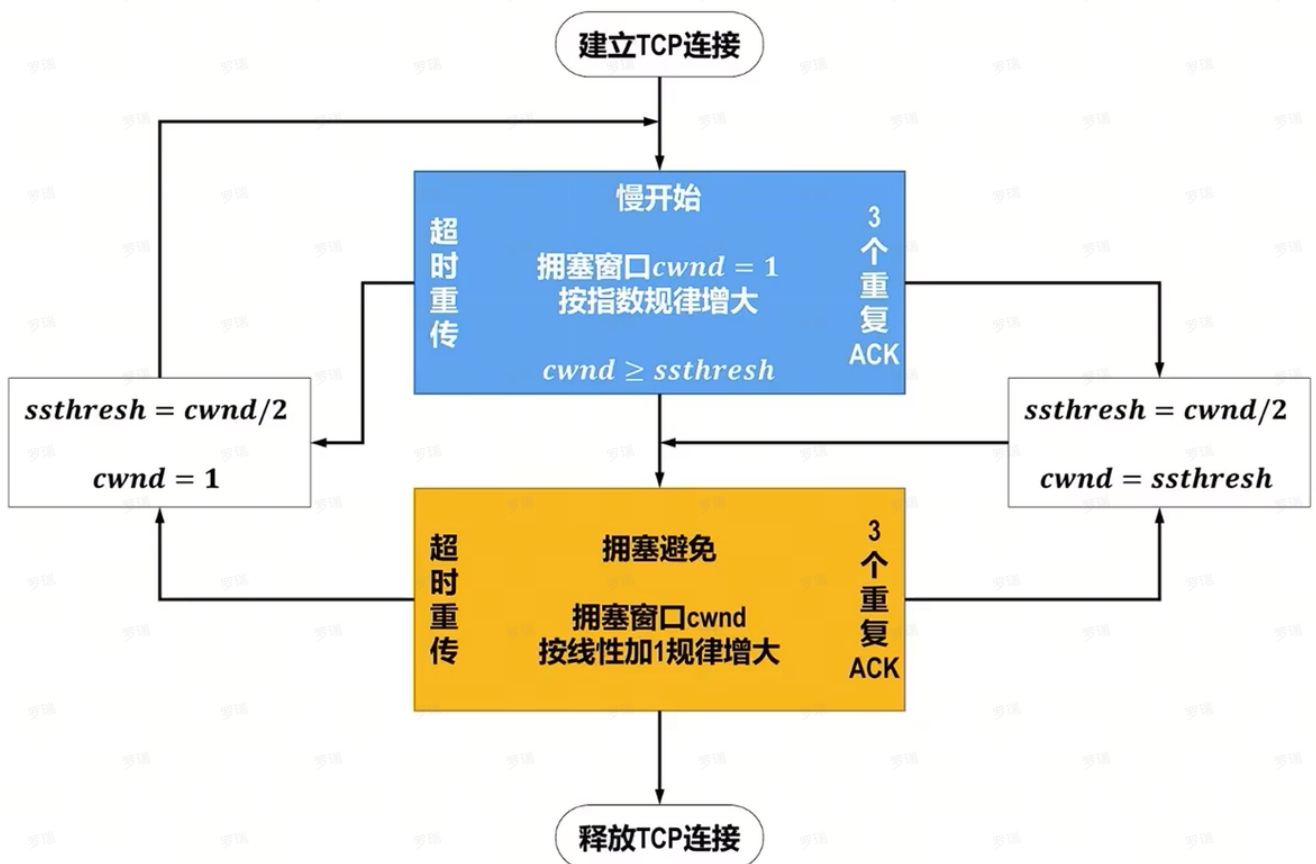


图4：TCP Reno拥塞控制程序流程图

# 程序运行测试与传输结果分析

## 1 自行设置丢包率

这个代码片段通过随机数的生成机制模拟了**随机丢包**的行为，用来测试网络协议在丢包情况下的可靠性和性能。这里我在服务器端定义，设置丢包率为1%。

```
1 void receiveandack(){
2 while (true) {
3     memset(&recv, 0, sizeof(UDP_DATAGRAM));
4     recvfrom(sock, (char*)&recv, sizeof(recv), 0, (SOCKADDR*)&client, &l);
5
6     //=====
7
8     //真：随机丢包
9     //设置随机种子为当前时间
10    countcut++;
11    // 生成范围为 1 到 10000 的随机数
12    int a = countcut % 100;//随机丢包率，1%
13    if (!a)
14        continue;//跳过后面的操作，来模拟丢包
15    .....
```

## 2 传输测试：3.png

发送端（客户端）日志输出：

```
D:\28301\Desktop\CSNet\作业\编程3\3-3\Client\x64\Debug\Client.exe
第一次握手建连报文发送成功
建连中: *****建连成功!
请输入要传输的文件的路径
D:/28301/Desktop/new/3. jpg
[info] 文件读取成功, 大小: 11968994 字节, 完整数据报个数: 1463, 尾数据报字节数: 482
开始传输文件对应的UDP数据报.....
Mode: 慢启动
当前窗口大小: 2
当前门限大小: 16
=====滑动窗口位置展示=====
窗口底部序列: 2, 窗口顶部序列:3
=====
Mode: 慢启动
当前窗口大小: 4
当前门限大小: 16
=====滑动窗口位置展示=====
窗口底部序列: 4, 窗口顶部序列:7
=====
重复ACK!!!5的序列号报文重复ACK!
=====
序列为7的报文超时未确认, 触发重传机制, 并调整窗口大小与慢启动门限值.
Mode: 拥塞避免
当前窗口大小: 3
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 8, 窗口顶部序列:10
=====
Mode: 拥塞避免
当前窗口大小: 4
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 11, 窗口顶部序列:14
=====
Mode: 拥塞避免
当前窗口大小: 5
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 15, 窗口顶部序列:19
=====
Mode: 拥塞避免
当前窗口大小: 6
当前门限大小:2
```

图5：客户端日志输出：观察到慢启动模式到拥塞避免模式的切换

```
当前门限大小:5
=====滑动窗口位置展示=====
窗口底部序列: 543, 窗口顶部序列:548
=====
重复ACK!!!543的序列号报文重复ACK!
=====
Mode: 拥塞避免
当前窗口大小: 7
当前门限大小:5
=====滑动窗口位置展示=====
窗口底部序列: 549, 窗口顶部序列:555
=====
重复ACK!!!550的序列号报文重复ACK!
=====
重复ACK!!!550的序列号报文重复ACK!
=====
重复ACK!!!550的序列号报文重复ACK!
=====
三次重复ack, 进行快重传, 并调整窗口大小与慢启动门限值.
Mode: 拥塞避免
当前窗口大小: 8
当前门限大小:7
=====滑动窗口位置展示=====
窗口底部序列: 556, 窗口顶部序列:563
=====
重复ACK!!!555的序列号报文重复ACK!
=====
Mode: 拥塞避免
当前窗口大小: 9
当前门限大小:7
=====滑动窗口位置展示=====
窗口底部序列: 564, 窗口顶部序列:572
=====
Mode: 拥塞避免
当前窗口大小: 10
当前门限大小:7
=====滑动窗口位置展示=====
窗口底部序列: 573, 窗口顶部序列:582
=====
```

图6：客户端日志输出：观察到任意模式出现三次冗余ACK时发生快恢复现象

```
D:\28301\Desktop\CSNet\作业\编程3\3-3\Client\x64\Debug\Client.exe
=====滑动窗口位置展示=====
窗口底部序列: 1440, 窗口顶部序列:1443
重复ACK!!!1440的序列号报文重复ACK!
Mode: 拥塞避免
当前窗口大小: 5
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 1444, 窗口顶部序列:1448
Mode: 拥塞避免
当前窗口大小: 6
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 1449, 窗口顶部序列:1454
Mode: 拥塞避免
当前窗口大小: 7
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 1455, 窗口顶部序列:1461
Mode: 拥塞避免
当前窗口大小: 8
当前门限大小:2
=====滑动窗口位置展示=====
窗口底部序列: 1462, 窗口顶部序列:1462
序列为1462的报文超时未确认, 触发重传机制, 并调整窗口大小与慢启动门限阈值.
Mode: 拥塞避免
当前窗口大小: 5
当前门限大小:4
恭喜, 所有数据报文均被正确接收! 文件传输时间20.987s
传输吞吐率4469.05kb/s
断连成功
请按任意键继续. . .
```

图7：客户端日志输出：观察到文件传输性能

文件传输时间 20.987s,传输速率4469.05kb/s

```
Microsoft Visual Studio 调试控制台
UDP校验正确收到序号为1458的报文,
回复序列号为1458的累积确认!
=====
校验和为8178
UDP校验正确收到序号为1459的报文,
回复序列号为1459的累积确认!
=====
校验和为38015
UDP校验正确收到序号为1460的报文,
回复序列号为1460的累积确认!
=====
校验和为48131
UDP校验正确收到序号为1461的报文,
回复序列号为1461的累积确认!
=====
校验和为13600
UDP校验正确收到序号为1462的报文,
回复序列号为1462的累积确认!
=====
*文件接收成功!
断连成功
写文件完成!

D:\28301\Desktop\CSNet作业\编程3\3-3\server\x64\Debug\server.exe (进程 20680) 已退出, 代码为 0。
按任意键关闭此窗口. . .
```

**图8：服务器端日志输出：观察到每一个接收报文的校验和、序列号、累积确认位置等信息**