

# Lab3-3拥塞控制算法实现

## 许健2013018

之前两次实验中，我们实现了一个简单的基于UDP协议的可靠传输，主要使用rdt3.0协议，完成了差错检测和确认重传功能。流量控制方便，采用流水线机制的GBN算法代替停等机制，在确认未返回之前允许发送多个数据包，提高了性能。

GBN 的特点如下：

- 1) 允许发送端发出  $N$  个未得到确认的分组
- 2) 分组首部中增加  $k$  位的序列号，序列号空间为  $[0, 2^k-1]$
- 3) 采用累积确认，只确认连续正确接收分组的最大序列号
- 4) 可能接收到重复的 ACK，所以需要在发送端设置定时器，定时器超时，重传所有未确认的分组



为了防止主机发送的数据过多或过快，造成网络中的路由器（或其他设备）无法及时处理，从而引入时延或丢弃。我们在发送消息时，还需要进行拥塞控制。TCP协议采用基于窗口的方法，通过拥塞窗口的增大或减小控制发送速率。我们也借鉴这样的做法，使用**New RENO**算法实现拥塞控制。

## New RENO算法

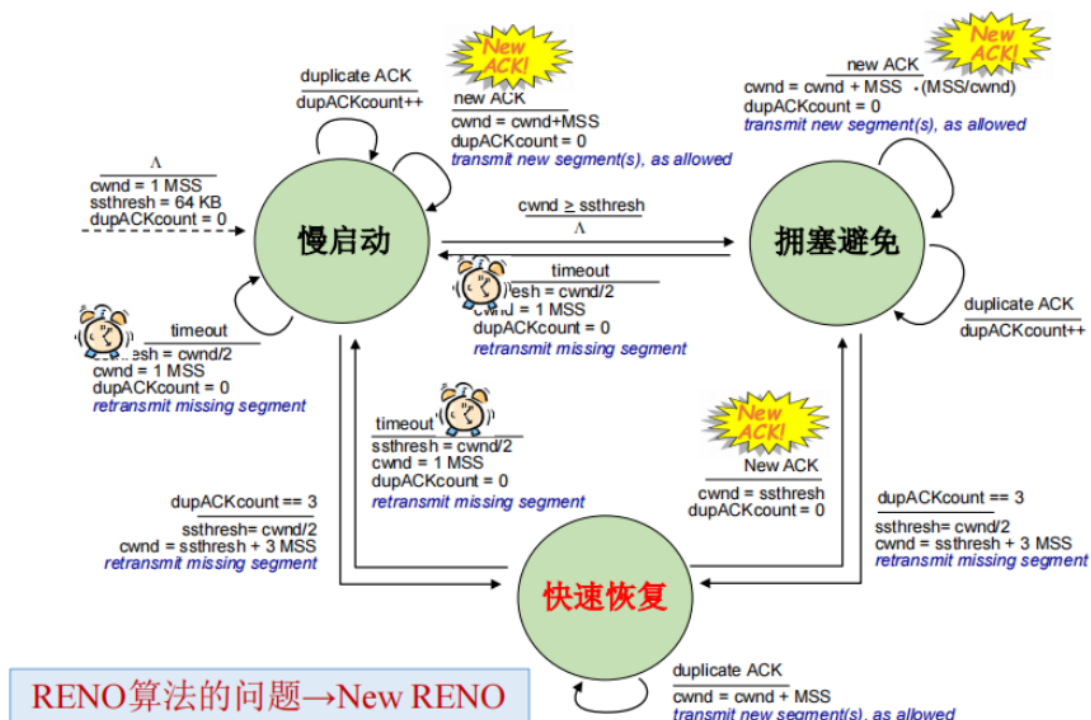
结合有限状态机，讲解RENO算法的原理：

RENO算法定义了三个阶段：

1. 慢启动阶段：初始拥塞窗口的大小  $cwnd=1$ ，每个RTT， $cwnd$ 翻倍（指数增长），但收到重复ACK消息时  $cwnd$  不变。
2. 拥塞避免阶段：每个RTT， $cwnd$ 增1（线性增长），但收到重复ACK消息时  $cwnd$  不变。
3. 快速恢复阶段：每个RTT， $cwnd$ 翻倍（指数增长）。

这三个阶段之间的关系如下：

- 连接建立时，进入慢启动阶段（可以理解为刚开始只发送一个报文，试探一下网络的情况）。
- 拥塞窗口达到阈值时，慢启动阶段结束，进入拥塞避免阶段。
- 收到三次重复ACK消息时，进入快速恢复阶段。
- 在快速恢复阶段，收到新的ACK消息时，进入拥塞避免阶段。
- 超时的情况下，进入慢启动阶段。



其中，在快速恢复阶段，收到新的ACK消息时，New RENO算法和RENO算法的不同之处在于，它会检查这个New ACK是否为已经发送的最后一个数据包的序列号，也就是说是否所有发送的数据包都被确认，如果不是，将会继续保持快速恢复，而不会进入拥塞避免。这样程序就不会频繁地在快速恢复和拥塞避免两个阶段反复横跳，造成阈值多次折半的问题。

另外，需要注意的一点是，流量控制和拥塞控制都是基于窗口的机制，二者在窗口大小的计算上是独立的。流量控制的窗口大小取决于接收端的缓冲区，只是我们上面为了简单将其设置为固定大小；拥塞控制的窗口大小是基于网络的情况。而实际发送窗口取决于接收通告窗口和拥塞控制窗口中的较小值。

## 程序实现

New RENO会用到的全局变量如下，包括通告窗口 $rwnd$ （这里取固定值10）、拥塞窗口 $cwnd$ 、阈值 $ssthresh$ 、reno状态机的状态 $renoState$ 等。再回顾以下我们的协议头部字段， $window$ 字段显示当前可发送窗口大小，取决于接收通告窗口和拥塞控制窗口中的较小值。

```
//GBN需要使用的全局变量
int resendCount = 0;           // 重传次数
int base = 2;                  // base之前序列号累计确认
bool resend = false;           // 是否超时重传
bool restart = false;          // 是否重新开始计时
bool wait = false;             // 是否因窗口不够而需等待
long long lenCopy = 0;         // 文件数据偏移量的拷贝，供重传时使用

//拥塞控制
const int rwnd = 10;           // 接收通告窗口大小，固定值
double cwnd = 1;               // 拥塞窗口大小
int ssthresh = 8;              // 阈值
unsigned char lastAck = 0;      // 上一个ACK序列号
int dupAck = 0;                // 重复收到的ACK次数
int renoState = 0;             // RENO状态机的状态，0为慢启动，1为拥塞控制，2为快速恢复

struct Head {
    u_char type;                //数据包类型
```

```

u_char window;           //窗口大小
u_char seq;              //序列号
u_short checksum;        //校验和
u_short length;          //数据部分长度
};

```

## 计时线程

计时线程中，我们设置了restart标志位，只要在其他线程中正确设置这个标志位，计时线程就可以正常运行。我们还设置了resend标志位，这是负责通知主线程超时重传，超时后阈值减半，窗口大小为1，回到慢启动阶段。

```

//标志位restart、resend
//timer线程函数，管理计时器
DWORD WINAPI timer(LPVOID lparam) {
    clock_t start = clock();
    while (true) {
        //是否重新开始计时
        if (restart) {
            start = clock();
            restart = false;
        }
        //是否超时重传
        if (clock() - start > MAX_WAIT_TIME) {
            resend = true;
            start = clock();

            // 超时后阈值减半，窗口大小为1
            //回到慢启动阶段
            ssthresh = cwnd / 2;
            cwnd = 1;
            dupAck = 0;
            renoState = 0;
        }
    }
    return 0;
}

```

## 接收ACK消息进程

接收到正确的ACK消息后，还需要处理RENO状态机的状态转换。在最后的程序演示中，我们观察到了因为发生超时重传，导致发送端从拥塞避免状态转变到慢开始状态，拥塞窗口的变化；以及三次ACK累计确认导致快速恢复的情况。

```

//接收ACK线程
DWORD WINAPI recvThread(LPVOID IpParameter) {
    //线程socket sockSender
    SOCKET sockSender = *(SOCKET*)IpParameter;
    int len = sizeof(SOCKADDR);

    //接收数据包recv
    Head recv;
    while (true) {
        //接收到数据包
        memset(&recv, 0, sizeof(recv));
    }
}

```

```

        if (recvfrom(sockSender, (char*)&recv, sizeof(recv), 0,
(SOCKADDR*)&ServerAddr, &len) > 0) {
            // 验证校验和、类型字段
            if (recv.type == ACK && !Checksum((u_short*)&recv, sizeof(recv))) {
                lenCopy += (recv.seq - base) * MAX_SEND_SIZE;
                base = recv.seq;
                //清除标志位
                restart = true;
                resendCount = 0;
                wait = false;

                //RENO算法状态处理
                // 慢启动
                if (renoState == 0) {
                    // 判断是否为new ACK，是的话更新窗口空间
                    if (recv.seq == lastAck) {
                        dupAck++;
                    }
                    else {
                        dupAck = 0;
                        lastAck = recv.seq;
                        cwnd++;
                    }
                    // 窗口大小超过阈值，进入拥塞避免阶段
                    if (cwnd >= ssthresh)
                        renoState = 1;
                    // 重复ACK超过3次，进入快速恢复阶段
                    if (dupAck == 3) {
                        cout << "快速恢复..." << endl;
                        resend = 1;
                        ssthresh = cwnd / 2;
                        cwnd = ssthresh + 3;
                        renoState = 2;
                    }
                }
            }
            // 拥塞避免
            else if (renoState == 1) {
                if (recv.seq == lastAck) {
                    dupAck++;
                }
                else {
                    dupAck = 0;
                    lastAck = recv.seq;
                    cwnd += 1.0 / cwnd;
                }
                // 重复ACK超过3次，进入快速恢复阶段
                if (dupAck == 3) {
                    cout << "快速恢复..." << endl;
                    resend = 1;
                    ssthresh = cwnd / 2;
                    cwnd = ssthresh + 3;
                    renoState = 2;
                }
            }
            // 快速恢复
            else {
                if (recv.seq == lastAck) {
                    cwnd++;

```

```

    }
    //新RENO算法
    // 接收到new ACK，需要判断是否发送的消息都被接收
    // 如果不是，保持快速恢复；如果是，进入拥塞避免阶段
    else {
        if (recv.seq == Seq) {
            renoState = 1;
            lastAck = recv.seq;
        }
    }
}
}
}
}
return 0;
}

```

## 发送消息函数

在发送消息的函数中，window应该代表实际发送窗口的大小，而这取决于接收通告窗口和拥塞控制窗口中的较小值。实验时将rwnd设置为固定窗口大小，也可以将其设置为可变的，由接收端发送来的消息确定，可以用消息头中的window字段加上控制位来表示rwnd。

```

void sendData(char* Buffer, int FileSize) {
    // 实际发送窗口取决于接收通告窗口和拥塞控制窗口中的较小值
    int window = 0;
    HANDLE Thread1 = CreateThread(NULL, NULL, recvThread, LPVOID(&Client), 0,
    NULL);
    HANDLE Thread2 = CreateThread(NULL, NULL, timer, LPVOID(NULL), 0, NULL);
    int sentLen = 0; //已经发送的长度
    u_int size = MAX_SEND_SIZE; //发送数据包size
    bool last = 0; //判断是否是最后一个数据包
    base = Seq;

    while (1) {
        //更新窗口
        if (rwnd < cwnd)
            window = rwnd;
        else
            window = cwnd;

        //需要超时重传的情况，Seq回到base的位置
        if (resend == true) {
            sentLen = lenCopy;
            Seq = base;
            resend = false;
        }

        //当前window还可以发送
        if (((base + window) < 256 && (int)Seq < base + window) ||
            ((base + window) >= 256 && ((int)Seq >= base || (int)Seq < (base +
window) % 256))) {
            if (sentLen + MAX_SEND_SIZE > FileSize) {
                size = FileSize - sentLen;
                last = 1;
            }
            //发送数据包函数

```

```

        sendPackage(Buffer + sentLen, size, Seq, window, last);
        Seq++;
        sentLen += size;
        //针对最后一个数据包的处理，退出循环
        if (sentLen == FileSize) {
            closeHandle(Thread1);
            closeHandle(Thread2);
            break;
        }
    }
    // 若下一个序列号不在窗口内，等待ACK
    else {
        cout << "Waiting for window!" << endl;
        wait = true;
        while (wait & !resend) {
            //sleep(100);
        }
    }
}
}
}

```

## 程序演示

### 无丢包情况

在没有丢包的情况下，程序的接收发送速度很快，window大小取决于接收通告窗口和拥塞控制窗口中的较小值，保持为10不变。但是cwnd窗口的一直在增长，拥塞避免阶段1个RTT时间，cwnd大小增1。

```

[SEND]Client: Seq = 170 Window = 10 cwnd=19.6698 Length = 10000 CheckSum = 65457
[SEND]Client: Seq = 171 Window = 10 cwnd=19.7207 Length = 10000 CheckSum = 60451
[SEND]Client: Seq = 172 Window = 10 cwnd=19.7714 Length = 10000 CheckSum = 16089
[SEND]Client: Seq = 173 Window = 10 cwnd=19.822 Length = 10000 CheckSum = 38818
[SEND]Client: Seq = 174 Window = 10 cwnd=19.8724 Length = 10000 CheckSum = 6797
[SEND]Client: Seq = 175 Window = 10 cwnd=19.9227 Length = 10000 CheckSum = 56444
[SEND]Client: Seq = 176 Window = 10 cwnd=19.9729 Length = 10000 CheckSum = 46011
[SEND]Client: Seq = 177 Window = 10 cwnd=20.023 Length = 10000 CheckSum = 12612
[SEND]Client: Seq = 178 Window = 10 cwnd=20.0729 Length = 10000 CheckSum = 16876
[SEND]Client: Seq = 179 Window = 10 cwnd=20.1228 Length = 10000 CheckSum = 38424
[SEND]Client: Seq = 180 Window = 10 cwnd=20.1725 Length = 10000 CheckSum = 63434
[SEND]Client: Seq = 181 Window = 10 cwnd=20.222 Length = 10000 CheckSum = 18088
[SEND]Client: Seq = 182 Window = 10 cwnd=20.2715 Length = 10000 CheckSum = 40305
[SEND]Client: Seq = 183 Window = 10 cwnd=20.3208 Length = 10000 CheckSum = 1310
[SEND]Client: Seq = 184 Window = 10 cwnd=20.37 Length = 10000 CheckSum = 28651
[SEND]Client: Seq = 185 Window = 10 cwnd=20.4191 Length = 10000 CheckSum = 161
[SEND]Client: Seq = 186 Window = 10 cwnd=20.4681 Length = 10000 CheckSum = 30490
[SEND]Client: Seq = 187 Window = 10 cwnd=20.5169 Length = 7353 CheckSum = 22788
send over!
发送数据:1857353 Bytes
发送时间0.271s
吞吐量6853.7KB/s
Save Hand...
[FIN,ACK]Client: Seq=188 CheckSum=13120
挥手成功
program will close after 3 seconds
请按任意键继续. . .

```

### 设置丢包率为5%

当出现超时重传时，RENO回到慢开始阶段，sssthresh大小等于cwnd/2，cwnd变成1。从图中可以看到，经历了重传，sssthresh变成3，Window=cwnd=1。

```

Waiting for window!
[SEND]Client: Seq = 178 Window = 6 cwnd=6 sssthresh=2 Length = 10000 CheckSum = 17900
Waiting for window!
Waiting for window!
Waiting for window!
Waiting for window!
Waiting for window!
[SEND]Client: Seq = 173 Window = 1 cwnd=1 sssthresh=3 Length = 10000 CheckSum = 41122

```

当发送端连续接收到3个重复的ACK时，会进行快速恢复阶段，此时sssthresh变成cwnd/2，cwnd变成sssthresh+3。

由图中可以看出，sssthresh由7变成cwnd/2=6，cwnd由13.6261变成sssthresh+3=9。注意cwnd为double类型，sssthresh为int类型，因此会发生舍入。这是因为TCP中的reno算法增长以字节为单位，而我们这以数据包为单位，因此cwnd设置为了浮点型。

```
[SEND]Client: Seq = 165 Window = 10 cwnd=13.4781 ssthresh=7 Length = 10000 CheckSum = 55063
Waiting for window!
[SEND]Client: Seq = 166 Window = 10 cwnd=13.5523 ssthresh=7 Length = 10000 CheckSum = 14739
Waiting for window!
[SEND]Client: Seq = 167 Window = 10 cwnd=13.6261 ssthresh=7 Length = 10000 CheckSum = 60225
Waiting for window!
Waiting for window!
快速恢复...Waiting for window!
[SEND]Client: Seq = 158 Window = 9 cwnd=9 ssthresh=6 Length = 10000 CheckSum = 7141
[SEND]Client: Seq = 159 Window = 9 cwnd=9 ssthresh=6 Length = 10000 CheckSum = 34452
[SEND]Client: Seq = 160 Window = 9 cwnd=9 ssthresh=6 Length = 10000 CheckSum = 60097
[SEND]Client: Seq = 161 Window = 9 cwnd=9 ssthresh=6 Length = 10000 CheckSum = 45043
[SEND]Client: Seq = 162 Window = 9 cwnd=9 ssthresh=6 Length = 10000 CheckSum = 34829
[SEND]Client: Seq = 163 Window = 9 cwnd=9 ssthresh=6 Length = 10000 CheckSum = 30623
```

当出现丢包时，我们程序的吞吐率还是受到了影响，下降了不少，一方面是因为丢包，我们窗口的值一直保持5左右，缓冲区变小。另一方面则是重传时间设置的是固定值，无法根据网络拥塞程度动态变化。相比停等协议还是有很大的性能优势，但相比于GBN的优势则不明显，因为我们无法模拟出网络中复杂的情况，网络拥塞窗口的设置甚至阻碍了性能的提升。

```
[SEND]Client: Seq = 183 Window = 10 cwnd=11.4672 ssthresh=7 Length = 10000 CheckSum = 1310
Waiting for window!
[SEND]Client: Seq = 184 Window = 10 cwnd=11.5544 ssthresh=7 Length = 10000 CheckSum = 28651
Waiting for window!
[SEND]Client: Seq = 185 Window = 10 cwnd=11.641 ssthresh=7 Length = 10000 CheckSum = 161
Waiting for window!
[SEND]Client: Seq = 186 Window = 10 cwnd=11.7269 ssthresh=7 Length = 10000 CheckSum = 30490
[SEND]Client: Seq = 187 Window = 10 cwnd=11.8968 ssthresh=7 Length = 7353 CheckSum = 22788
send over!
发送数据:1857353 Bytes
发送时间3.072s
吞吐率604.607KB/s
Wave Hand...
[FIN,ACK]Client: Seq=188 CheckSum=13120
挥手成功
program will close after 3 seconds
请按任意键继续. . .
```