目录

1 实验内容说明

1.1 实验题目

实验 3：基于 UDP 服务设计可靠传输协议并编程实现

1.2 实验说明

实验 3-3：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

2 协议设计

基于上次滑动窗口，实现 NEW RENO 拥塞控制算法。

2.1 数据包格式

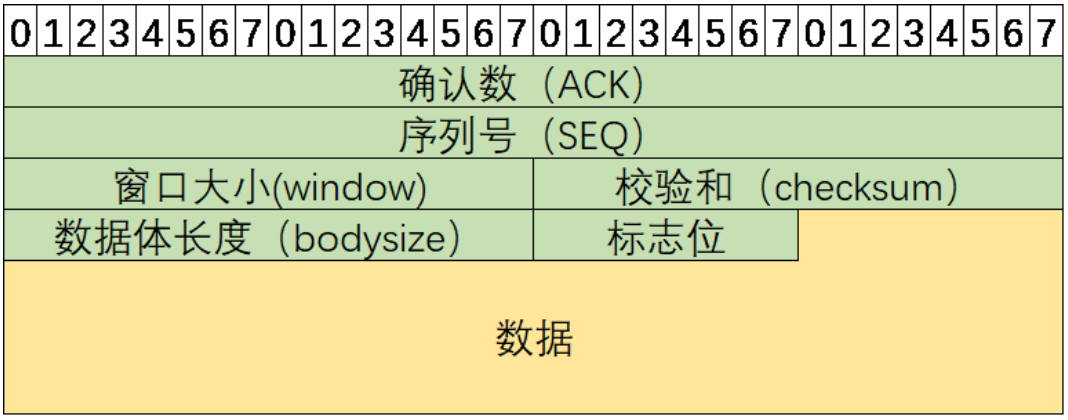

图 1: 数据包格式

标志位 FLAG 分别设置了：ACK，SYN，FIN，SEQ 四项，window 为接收方缓冲区大小。协议中，每个序列号对应一个字节。

2.2 RENO 状态机

拥塞控制的状态机分为三个状态: 慢启动、拥塞控制、快速恢复。对接收到新 ACK、重复 ACK、超时这三个事件分别做不同处理，状态机细节如图2所示。

2.3 NEW RENO 的改进

在 RENO 状态机的基础上，对于部分确认不退出快速恢复状态，对于恢复确认才退出快速启动状态。而部分确认是指只确认了部分失序数据包的 ACK，恢复 ACK 则是指确认了所有失序数据包的 ACK。

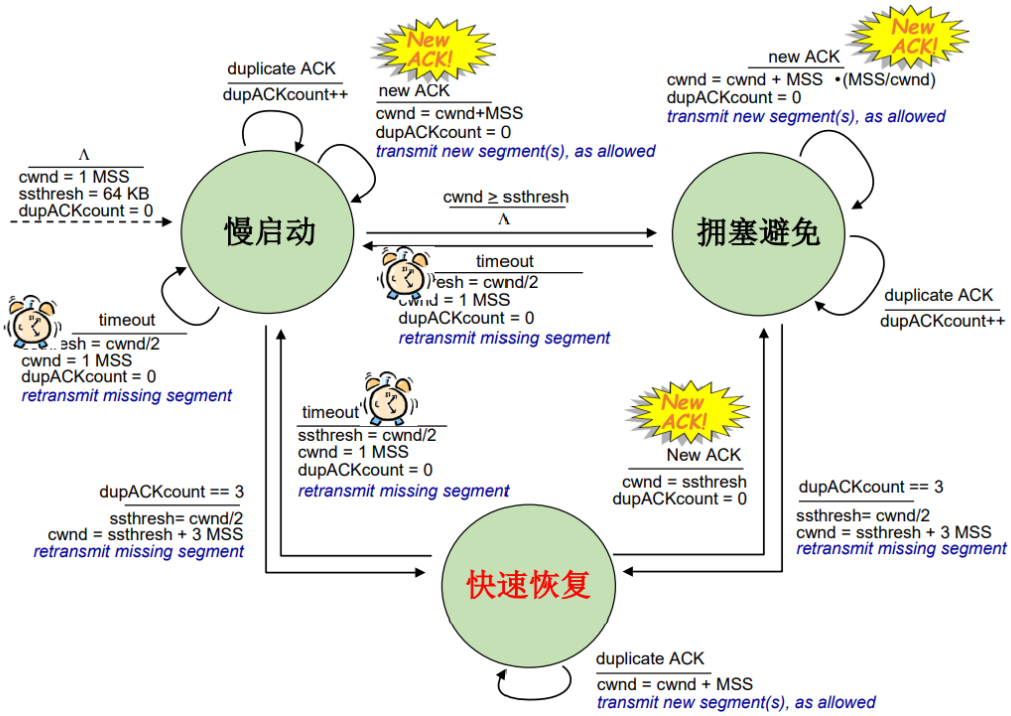

图 2: 拥塞状态控制机（课件原图）

3 拥塞控制的实现

在实现中，设置了状态变量表示拥塞控制状态机状态。

</> CODE 1: 拥塞控制状态

```
enum class CongestionState
{
    start,              //慢启动
    avoid,              //拥塞避免
    fastRecovery,       //快速恢复
};
```

3.1 NEW ACK

当接受到新 ACK 时，判断状态机当前状态，分别采取不同行动。

</> CODE 2: 处理 NEW ACK

```
void RSend::updateCongestionWindow(uint32_t ack)
{
    switch (cState) {
    case CongestionState::start: //当前是慢启动
        cwnd += MSS;             //指数增加
        if (cwnd > ssthresh) {   //如果达到阈值
```

```
            //状态更新为拥塞避免
            setCongestionState(CongestionState::avoid);
        }
        break;
    case CongestionState::avoid: //当前是慢启动
        cwnd += MSS *MSS / cwnd; //线性增加
        break;
    case CongestionState::fastRecovery:
        if (ack > lastSeq) {    //是部分应答
            //do nothing 不改变状态
        }
        else {                  //是恢复应答
            //退出快速恢复，回到拥塞避免
            cwnd = ssthresh;
            dupACK = 0;
            setCongestionState(CongestionState::avoid);
        }
        break;
    }
}
```

3.2 重复 ACK

对于重复 ACK，当达到 3 次重复后，进入快速恢复状态。

</> CODE 3: 处理重复 ACK

```
if (lastAck == ack) { //if duplicate ACK
    dupACK++;
    if (cState == CongestionState::fastRecovery) {
        cwnd = cwnd + MSS; //当前是快速恢复状态，拥塞窗口指数增加
    }else if (dupACK>=3) {
        //enter fastRecovery
        uint32_t sendnum = sendSeq(lastAck, MSS); //进行快速重传
        ssthresh = cwnd / 2;
        cwnd = ssthresh + 3 * MSS;
        setCongestionState(CongestionState::fastRecovery);
        //记录进入快速启动时，发送的最高序列号
        //用以区分NEW RENO中部分确认和恢复确认
        lastSeq = nextSeq;
    }
} else {
    dupACK = 0;
    lastAck = ack;
}
```

3.3 超时

当超时发送时，一律将当前状态设为慢启动状态。

</> CODE 4: 处理重复 ACK

```
if (/*需要超时重传的判断，略*/) { //如果需要重传
    if (evt.nTimeout > maxNTimeout) { //超过最大超时次数
        //错误处理，略
    }
    //重传处理，略

    ssthresh = cwnd / 2;
    cwnd = MSS;
    dupACK = 0;
    setCongestionState(CongestionState::start);

    //重新设置定时器，略
}
```

4 自适应 RTT 实现

为了获得 RTT，实现了 RTT 采样与 RTT 自适应更新。

首先，给原定时器节点增加成员，用以实现 RTT 采样。

</> CODE 5: RTT 采样数据结构

```
struct Timevt {
    //定时器预定发生时刻
    chrono::time_point<chrono::system_clock> tp;
    //数据包发送时刻
    chrono::time_point<chrono::system_clock> sendTimestamp =
        chrono::system_clock::now();
    bool acked = false;//是否已经收到该数据包的回复
    //其他成员
};

//定时器保存在一个以时刻排序的队列中
vector<Timevt> timevts;
```

其次，接收端回复的数据包会告知确认的是哪一个数据包，即 SACK。最后，根据数据包的对应关系，当接收到回复后，对 RTT 进行采样。

</> CODE 6: RTT 采样

```
for (auto& evt : timevts) {
    //无关代码
```

```
    if (evt.dataSeq == seq) { //找到回复数据包的对应数据包
        evt.acked = true;
        auto now = system_clock::now();
        updateRTT(evt.sendTimestamp, now);
        //无关代码
    }
}
```

RTT 和 RTO 使用如下公式更新:

$$EstimatedRTT = (1 -)EstimatedRTT + SampleRTT$$

$$RTO = * EstimatedRTT$$

5 效果演示

5.1 文件发送

发送端在命令行参数中输入接受方 IP,UDP 端口, 发送文件名称。发送后显示统计

```
.\Sender.exe -i 127.0.0.1 -p 12300 -f 2.jpg
...
send file spent 4.092461 s.
1857359 bytes in total. speed = 453848 bytes ps
success to send the file: 1.jpg.
```

接收端同样在命令行参数中指明 IP 和 UDP 端口。最终成功发送文件。

```
.\Recipient.exe -i 127.0.0.1 -p 12300
...
succeed to close connect.
recv file successfully, file:C:/Users/A/Desktop/net/recvfile/1.jpg
```

5.2 日志记录

```
...
[LOG]: <Congestion> NEW_ACK when avoid cwnd=10037 ssthresh=6782, upda
[LOG]: GET 224 bytes from buffer .
[LOG]: SEND SEQ [1851828,1852052).
[LOG]: TIMER [1851828,1852052) in 6430 ms, 0 times,flag=16.
[LOG]: STATE nextSeq=1852052,sendBase=1841911,rwnd=32768,cwnd=10141,t
[LOG]: RECV ACK ack=1842935, nextSeq=1852052, rwnd=32768, buffer[1841
[LOG]: RTT rtt=1257,rto=6285.
[LOG]: POP 1024 bytes from buffer. buffer:[1842935,1857359).
...
```

发送端日志详细记录了 buffer 变化，超时重传，接受数据包, 拥塞状态更新，拥塞窗口，当前各项参数等状态信息。

接收端同样有详细日志。

```
...
[LOG]: recv-->recv | window=32768, nextACK=5869574,
buffer=[5869574,5869574).
[LOG]: RECV [5869574,5873670), flags=16.
[LOG]: PUSH 4096 bytes into buffer, buffer:[5869574,5873670).
[LOG]: <USER> recv 4096 bytes.
[LOG]: SEND ACK ,ack=5873670, window=32768.
...
```

5.3 关于性能的思考

目前程序的数据通过路由器后，可以保证可靠传输，但速度较慢。应该有两方面原因。

第一、路由器性能。在设置路由器无丢包、无延迟时使用路由器，也比不使用路由器要慢一个数量级。

第二、发送数据包是变长的，上限为 MSS，但是在实现中没有考虑下限，导致传输过程中出现了大量几百甚至几十字节左右的小数据包，拖慢了性能。