TCP拥塞控制原理与工程实现

(第一版)

前言

在某段时间内，若对网络资源的需求超过了该资源的承载能力，网络的性能就会变坏，这种情况叫做拥塞。当拥塞发生或即将发生时，如何不能及时抑制往网络上发送的报文数量，网络拥塞将不断加重，而网络的吞吐量会随着拥塞程度的加重而不断降低，直至为0。为增强应对网络拥塞的处理能力，提高系统稳定性，TCP协议引入了拥塞控制机制。作为TCP协议的重要组成部分，典型的TCP拥塞控制过程包括四个阶段，即慢启动、拥塞避免、快速重传和快速恢复，然而具体到其深层理论及实现细节上，拥塞控制则显得颇为复杂，具有不少研究价值。

出于工作需要，作者曾经在对Linux内核2.6.32版本和3.17版本的TCP协议栈源码、RFC文档以及中外学者发表的相关论文进行学习研究的基础上，设计实现了一套具备拥塞控制能力的自主TCP协议，并成功应用于生产环境。有感于目前TCP协议相关的资料要么太“学院派”，要么局限于源码级别的分析，作者根据工程实践经验，以及对TCP拥塞控制理论的理解和个人的“研究成果”，结合已有的TCP拥塞控制资料，最终整理成本文。

本文由四章构成，由浅入深逐步分析介绍TCP拥塞控制原理及其实现过程。第一章，对TCP拥塞控制的基本理论进行介绍，从大角度概括如何实现TCP拥塞控制；第二章，在第一章的基础上更进一步，介绍TCP拥塞控制的设计思想；第三章，对TCP拥塞控制每一个阶段的拥塞窗口调整策略、各个参数的变化情况以及发包行为进行详细的分析研究；第四章，对TCP拥塞控制四个主要阶段之外的一些重要组成部分，比如管道控制、误判恢复、拥塞控制算法和选择性确认，进行分析介绍，并展望拥塞控制的研究和改进方向。

与介绍TCP拥塞控制的一般性资料相比，本文尽量将拥塞控制理论及其实现过程讲解得更为详细深入，以期读者能够通过这篇报告，可以对TCP拥塞控制有一个全面深刻的理解。文中不仅对TCP拥塞控制理论及其实现过程进行基本介绍，还从数学证明和实例推导等角度进行分析研究，是一篇研究与介绍并重的报告。本文主要是个人对TCP拥塞控制理论的理解，文中多处内容是在缺少资料的条件下推导出来的，谬误在所难免，力所不及之处，敬请斧正。

James Wei
weijianlhp@163.com
2015年09月

目录

1. TCP拥塞控制基本理论

本章主要对TCP拥塞控制的基本理论进行介绍，使读者能够了解拥塞控制的基本组成和基本工作过程。更加深入的拥塞控制设计思想及拥塞控制实现过程将在接下来的两章介绍。

1.1 拥塞窗口cwnd

TCP拥塞控制的目的，是为了防止“过多的”数据包不合时宜地进入网络中，引起吞吐率下降，但又不能因对数据发送速率限制得过于严格而降低TCP吞吐率。总之，尽量将吞吐率维持在一个较高的水平，而又不至于导致网络陷入拥塞状态，并及时从拥塞状态退出，是拥塞控制的最终目的。为了实现这一点，Linux系统的TCP拥塞控制机制引入了一个变量——拥塞窗口(以下用cwnd表示)。每一个TCP会话建立以后，都保持一个拥塞窗口。拥塞窗口在TCP连接进入ESTABLISHED状态后，会根据收到报文的信息[1]和重传定时器超时事件作相应的调整。拥塞窗口可以直观地理解为TCP连接对网络拥塞状况的估计，当网络状况良好时，增大拥塞窗口，使得每一轮数据发送时，允许将更多的数据段发送到网络上；当探测到丢包时，认为网络即将发生拥塞，减小拥塞窗口，防止大量注入数据而导致网络状况持续恶化。TCP拥塞控制，很大程度上可以认为是根据接收到的信息，对拥塞窗口大小进行的控制。可以说，维护一个适当的拥塞窗口是TCP拥塞控制的核心。

1.2 TCP拥塞控制的根本原则

那么，TCP会话如何依靠拥塞窗口来实现拥塞控制呢?

Linux操作系统中的TCP拥塞控制，是基于这么一个原则，即

正在网络中传输的数据段个数，不得大于拥塞窗口。

用公式表示，就是

$$in\_flight \le cwnd$$

in_flight表示正在网络上传输的数据段个数[2]，cwnd表示拥塞窗口大小。

虽然看不到Windows操作系统的源码，但通过分析Windows系统上TCP报文的传输过程反向推导其拥塞控制实现过程，可以发现，两类操作系统的TCP拥塞控制实现过程是差不多的，每一轮发包时，都遵循in_flight ≤ cwnd的原则。

---

[1] 这个“信息”一般是报文的确认号。

[2] 有些资料用“pipe”表示正在网络上传输的TCP数据段个数，所以在下文中有时会将in_flight表示为pipe，并翻译为“管道”，但意思是一样的。

“正在网络中传输的数据段个数”表示当前时刻，已经被发送出去的，既没有被对端接收，也没有在中间线路佚失或被中间设备丢掉的数据段个数。in_flight在很多情况下是一个估计量，并不精确，因为发送端无法准确统计丢失的报文个数。图1-1演示了in_flight的计算过程。

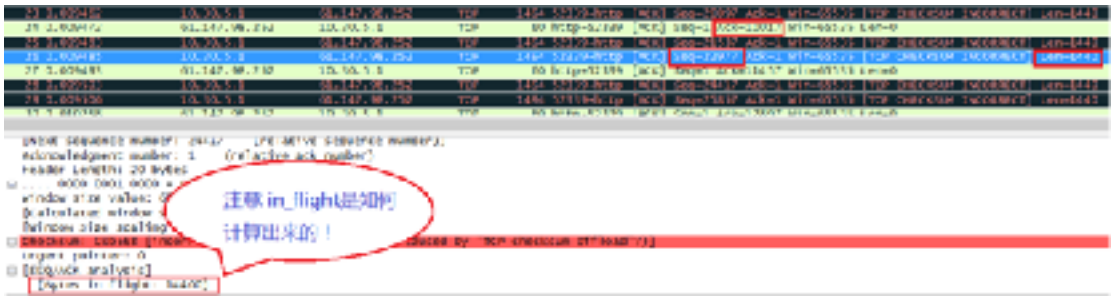


图1-1 in_flight的计算过程。图中，从10.30.5.1往61.147.98.252传输TCP数据，61.147.98.252先是回复一个ack,确认号为10017，表示“序列号为10017之前的数据都已收到”,接着10.30.5.1发送了两个数据报，当发送了序列号为22977，长度为1440字节的数据段后，bytes_in_flight = 22977 + 1440 – 10017 = 14400 bytes。如果每个报文的数据段长度fragment_length都是1440，并且没有报文丢失的话，则in_flight = bytes_in_flight / fragments_length = 10。

下面需要对这个原则特别说明三点：

(i) 从这个原则可以看出，Linux内核中拥塞窗口的大小是以TCP数据段个数为单位的，而早期RFC文档中对拥塞窗口的描述是以字节为单位的，导致很多人按照RFC文档的说法将拥塞窗口的单位理解为字节。事实上，以字节或者数据段作为拥塞窗口的单位都可以实现拥塞控制，但在对拥塞控制流程进行分析的过程中发现，用数据段作为拥塞窗口的单位实现起来比较简单一些。

(ii) 注意“数据段”这三个字。也就是说，TCP拥塞控制只对带数据的TCP报文产生影响，**而syn、fin、纯ack等不带数据的报文是不受TCP拥塞控制限制的，该发送的时候就发送**。比如，如果对端也给本端发送TCP数据段，当本端要回复不带数据的确认报文时，是不受这个TCP拥塞控制原则限制的。

有了这个原则，TCP拥塞控制问题就显得很直观了：一个TCP连接建立以后，通过调节拥塞窗口cwnd的大小，来控制正在网络上传输的数据段个数in_flight的大小，尽量避免大量数据涌入网络中而导致网络拥塞恶化，达到拥塞控制的目的。

具体来说，这个原则是这么发挥作用的：

当准备发送数据段时，先看看这时的TCP连接是否符合in_flight < cwnd (注意，不是in_flight ≤ cwnd)，如果符合，就把数据段发送出去，in_flight自增1，再对下一个待发送的报文进行同样的处理；如果不符合，则停止发包，未发送的数据段待下一轮发送时再做处理。用伪代码表示，就是

```
while (发送队列不为空) {
    if ( !(in_flight < cwnd)) {
        break;
    } else {
        tcp_transmit_skb();
        in_flight++;
     }
}
```

实际上，在发送数据段的过程中，是否可以发送该数据段还得看对端接收窗口“答不答应”，如果对端接收窗口太小的话，也是不能发送的，这一点就不再解释了。

1.3 “飞翔的报文”in_flight

从1.2节可以看出，为了维护“正在网络中传输的数据段个数，不得大于拥塞窗口”这个原则，TCP会话必须保证在发送数据段之前，获取当前正在传输的数据段个数信息。为此，Linux系统需要为每个TCP连接维护一个变量in_flight，表示当前正在网络上传输的数据段个数。具体到计算机语言实现上，in_flight由四个变量控制，即packets_out、sacked_out、lost_out和retrans_out，分别表示当前已发送出去但未被确认的数据段个数、通过重复确认而离开网络的数据段个数、在网络上丢失的数据段个数和被重传出去的数据段个数。

为了避免疑惑，对sacked_out和packets_out补充说明如下：

(i) sacked_out的本意是表示本端收到选择确认报文时，通过其携带的选择确认信息而了解到的“已经离开网络的数据段个数”。当不支持选择确认时，表示通过重复确认报文而了解到的“已经离开网络的数据段个数”。举个例子，当发送端收到确认号为n的确认报文后，发送了n ~ n+10号数据段，接着又收到一个确认号为n的报文，这说明，n+1 ~ n+10号报文之间肯定有某个报文到达接收端了，亦即至少有个报文离开网络了，于是将sacked_out自增1。这么做是因为，当接收端接收到的数据段序号比自己想获得的序号大，比如，接收端想获得序号为n的数据段却收到一个序号为n+2的数据段，就会发送确认号为n的报文给发送端，在发送端看来，这就是一个重复确认。当TCP连接不支持选择确认时，我们可以根据这个TCP特性，用选择确认信息来控制sacked_out。

sacked_out的具体调整方法稍微有些复杂，将在第四章进行详细介绍，在此之前，先暂且认为“每收到一个重复确认，sacked_out加1”。

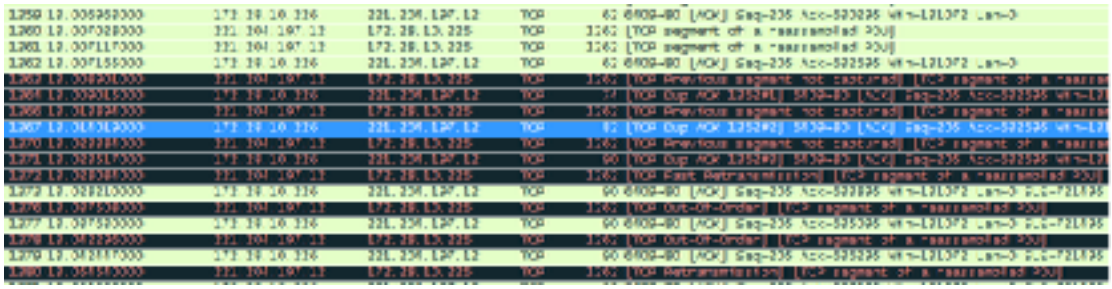


图1-2 从数据接收端看重复确认的生成过程。图中，接收端172.128.10.226想获得seq=632695的报文，却获得编号为1363、1366、1371等乱序报文，于是，每获得这样一个乱序报文，就把它放在乱序队列里面保存，并给发送端221.104.197.12发送一个重复确认。所以说，一个重复确认暗示一个数据包已经离开网络了。

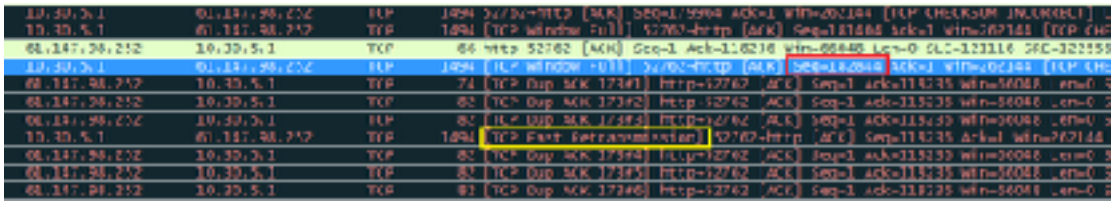


图1-3 从数据发送端看重复确认与快速重传报文。发送端10.30.5.1发送了序列号为182844的数据段以后，连续收到了接收端61.147.98.252发送的一系列确认号为118236的重复确认(Duplicate Acknowledgement)。这些重复确认是这么产生的：接收端收到了一系列序列号大于118236的报文，即乱序报文，每收到一个乱序报文(也就是说，该报文已离开网络)，接收端就发送一个重复确认。这期间，发送端每收到一个重复确认，sacked_out就自增1，即认为，有一个乱序报文离开网络了。

(ii) 注意packets_out和retrans_out的区别：packets_out等于待发送数据段与已被确认数据段之间的数据段个数。当发送一个新的、先前未被发送过的报文时，packets_out 自增1，而某个报文被重传时，retrans_out自增1，一个发送出去的报文只可以在packets_out中被统计一次。

in_flight的计算方法是:

$$in\_flight = packets\_out + retrans\_out - (lost\_out + sacked\_out)$$

当发送一个报文、接收一个报文或者重传定时器超时发生时，通过改变packets_out、sacket_out、retrans_out和lost_out的值，以维护in_flight一直处于可知的状态，与cwnd一起，实现拥塞控制。

packets_out、sacked_out、lost_out和retrans_out的具体统计过程将在第四章介绍，在此之前，凡是用到in_flight的地方，暂且认为它是已知的。

1.4 TCP拥塞控制有限状态机

TCP拥塞控制的核心在于维护一个适当的拥塞窗口cwnd，使得可以根据in_flight ≤ cwnd的原则保证输入网络中的数据段数量不至于过多，尽量避免网络拥塞。理论上，维护拥塞窗口的过程是靠四个TCP拥塞控制阶段实现的，即

慢启动、拥塞避免、快速重传和快速恢复。那么，具体到代码层面上，TCP拥塞控制该如何实现呢?

Linux内核对TCP拥塞控制算法的实现是靠TCP拥塞控制有限状态机实现的。

TCP拥塞控制有限状态机(往下简称为拥塞状态机)实际上是用一个变量state，表示当前的TCP连接处在什么样的拥塞状态，当收到对端发来的确认报文或者重传超时发生时，会根据当前所处的拥塞状态和确认报文所携带的确认号信息、时间戳信息等更新拥塞状态，同时更新拥塞窗口大小和慢启动阈值。

在Linux内核的TCP层实现中，TCP拥塞状态机由五种状态组成，即open、cwr、disorder、recovery和loss，其中cwr状态只有在收到显式拥塞通知并且TCP支持显式拥塞通知时才会存在。为简化流程，在分析TCP拥塞控制状态时将其忽略掉。于是本文所表述的拥塞状态机可以认为有四种状态。

(i) Open状态：相当于慢启动和拥塞避免阶段所处的状态，这时候的网络畅通，发送端收到的ACK报文都确认数据包的到达[1]。

(ii) Disorder状态：当在open状态收到重复确认时，将进入disorder状态。disorder状态相当于慢启动或者拥塞避免与快速重传之间的一种过渡状态。

(iii) Recovery状态：当收到足够多的重复确认时(一般是3个重复确认)，将会从disorder状态进入recovery状态，这时将发生快速重传。

(iv) Loss状态：当重传超时发生时，将会进入loss状态。

注意，在第(iii)点不说“收到3个重复确认”是因为Linux内核的确不是按照“每收到3个重复确认就发生快速重传”的原则进行处理的，这与一般所认识的“每收到3个重复确认即发生快速重传”稍有差别。具体收到收到个重复确认就发生快速重传则视具体情况而定，这一点将在第四章进行介绍。

TCP拥塞控制状态机的工作过程将在第3章进行介绍。

1.5 支持拥塞控制条件下的TCP协议工作流程

当TCP协议支持拥塞控制以后，TCP协议的工作流程可以概括如下:

[1] 有时候会收到窗口更新报文，这种报文一般只更新TCP头部的window字段，不带数据，确认号也不变，但不应被视为重复确认。

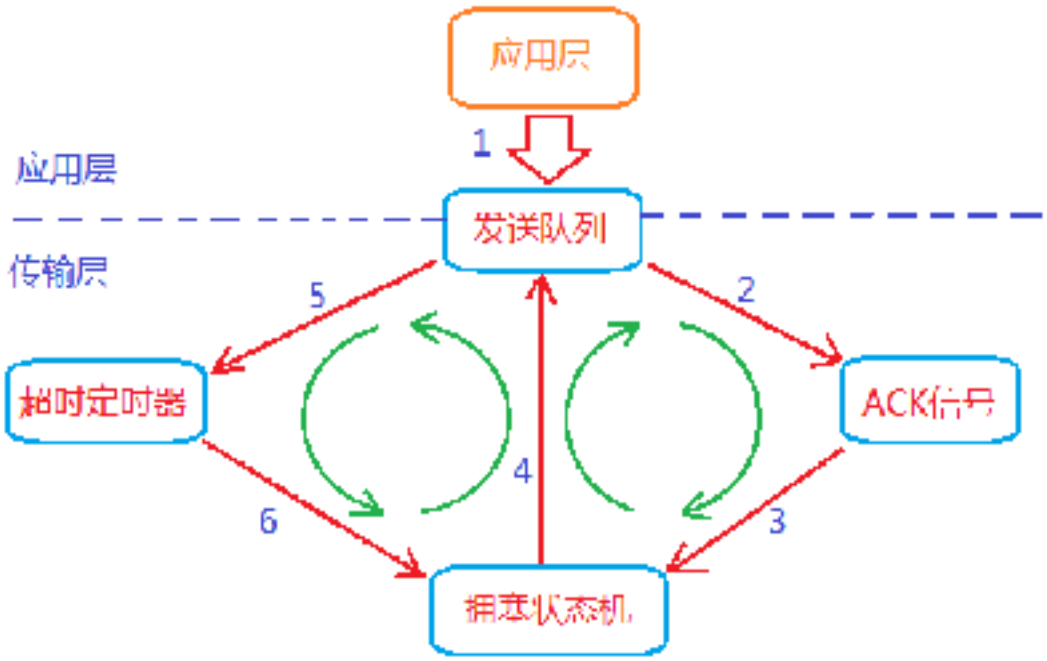


图 1-4 支持拥塞控制的TCP协议工作流程图，图中的流程环2-3-4通过不断重设超时时间来抑制流程环4-5-6的运行，流程环4-5-6只有在流程3断开的情况下才会执行

(1) 应用层负责往发送队列中添加所要发送的数据。如果在它添加数据之前，发送队列中没有数据，就触发步骤(2)和(5)，使TCP协议开始工作。

(2) 发送队列中的数据按照in_flight ≤ cwnd的原则发送数据。如果这一步是由步骤(1)触发的，就发送新数据(新数据指的是已加入发送队列，但还没有被发送过的数据)，如果这一步是由步骤(4)引起的，则该发送哪些数据由步骤(4)说了算，可能是重传数据段，也可能是发送新数据。接着转到步骤(3)。

(3) 在步骤(2)中发送数据后，对端收到数据，回复确认信息。当发送端收到确认信息后，先把发送队列中已被确认的数据段释放，接下来将确认信息交给拥塞状态机处理，转到步骤(4)。

(4) 拥塞状态机处理确认号信息或者超时信息以后，从发送队列中重传某些数据段或者发送新数据段，转到步骤(2)。并调整重传定时器，转到步骤(5)。

(5) 设定重传定时器，如果在超时时间之内没有被调整，则定时器超时，转到步骤(6)。

(6) 当超时发生时，触发TCP拥塞状态机的重传定时器超时处理流程，转到步骤(4)。

可见，在支持拥塞控制的条件下，当应用层将数据交给TCP协议后，TCP没有将所有待发送数据一次性全部发送出去，而是先发送一部分数据，等收到数据确认信息后，调整拥塞窗口，再发送数据……，如此循环下去，直到所有

待发送数据都被接收端接收。

2. TCP拥塞控制的设计思想

上一章已经介绍了拥塞控制的基本原理及其实现方式，本章在上一章的基础上，将介绍拥塞控制为什么这么设计，以及各阶段的行为。

2.1 拥塞控制的必要性

假如TCP协议没有拥塞控制机制，则其发包模式就很简单：一次性把所有待发送的数据包发送出去(如果对端接收窗口足够大)。这种做法在负载较小，网络畅通时没有问题，然而一旦发生网络拥堵，麻烦就来了，不断涌入的报文会加剧拥堵，使得网络吞吐率急剧降低，直至死锁，如图2-1中的绿线所示。

理想的拥塞控制，是将吞吐量维持在一个临界点极高值。然而，由于网络状态的不稳定性，以及数据发送端本身无法准确知道当前的网络状态，所以，实际的拥塞控制只能尽量将拥塞窗口维持在一个足够高的水平。

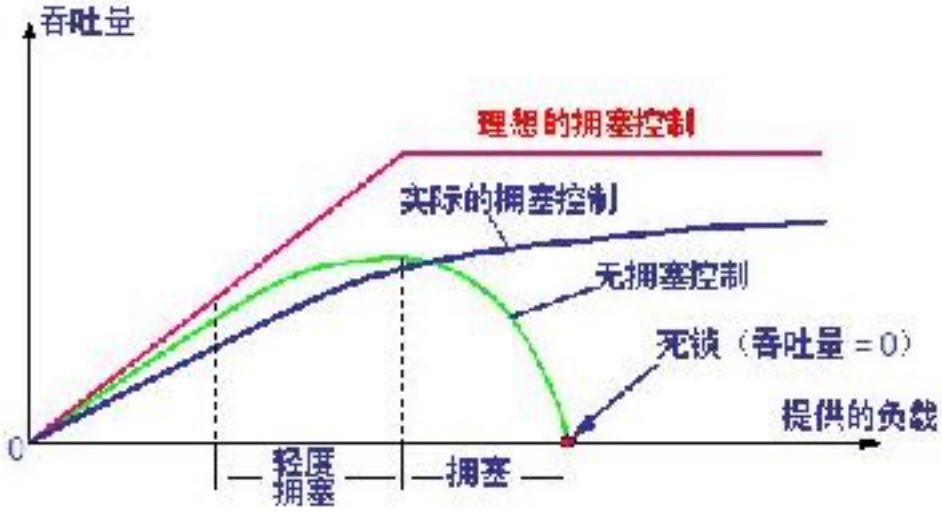


图2-1 吞吐量与负载的关系。吞吐量随着负载的增大而逐渐收敛于一个“上限”，而不是无限增

大。拥塞控制算法的研究意义在于，对于不同的拥塞控制算法，这个“上限”可能不同。

2.2 拥塞窗口的走向及调整时机

图2-2左半部分大致描绘出Reno算法拥塞窗口在慢启动和拥塞避免阶段的走向，即，在慢启动阶段，拥塞窗口乘法增长，增长得很快；在拥塞避免阶段，拥塞窗口加法增长，增长得比较慢。

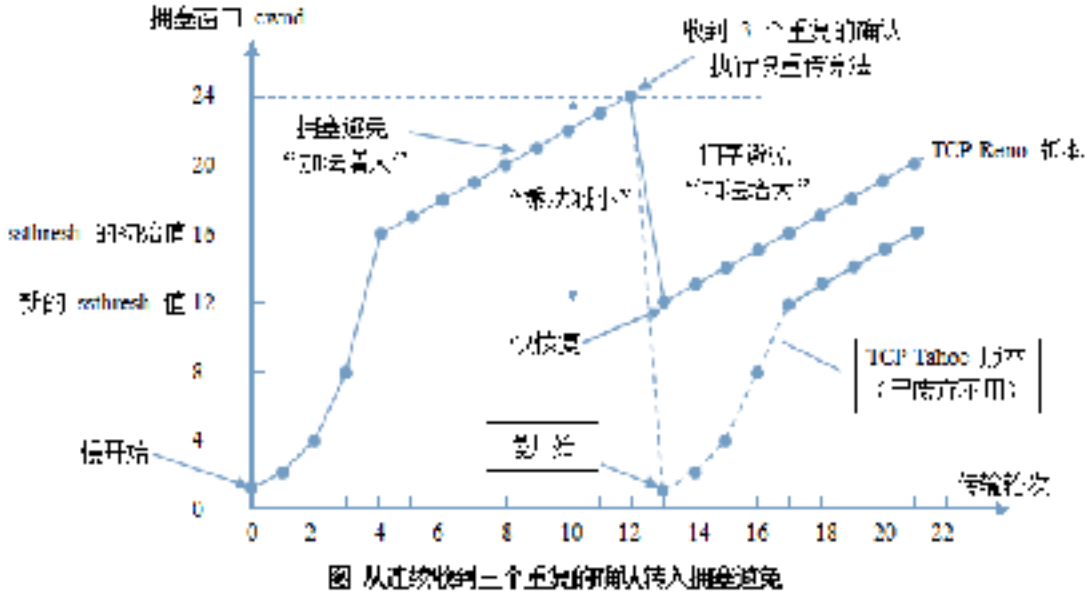


图2-2 拥塞窗口的走向图。这幅图经常出现于各类介绍TCP拥塞控制的资料上，在图中，0-12传输轮次的描绘大致是正确的，但第12传输轮次以后的描绘存在很多问题。注意几点：(1)这幅图以传输轮次为横坐标，但具体实现起来，拥塞窗口的调整是在收到ACK确认报文后进行的，而不是在每个传输轮次结束时进行；(2)这幅图中拥塞避免阶段明显采用Reno算法，而目前，很多拥塞控制算法都修改了拥塞避免阶段的窗口调整策略；(3) 图中所标记的快速重传和快速恢复阶段是错误的，事实上，快速重传和快速恢复阶段才是拥塞控制最复杂的阶段。

然而，由于在TCP连接整个过程中，一次往返时间RTT(Round Time Trip)一直处在动态改变的状态，而且，连续发出的一系列数据段，不可能在同一时刻收到对它们的确认，所以，不可能按照传输轮次的起始或者结束来调整拥塞窗口。实际上，拥塞窗口的调整时机是这样的：

拥塞窗口只有在收到确认报文，或者重传定时器超时时进行调整，其余时间段内，拥塞窗口保持不变。

这里的“确认报文”指的是TCP头部的ack标记被置上的报文。从这句话可以看出，拥塞窗口的调整是瞬时进行的，并且只有在收到对端发来的TCP报文，或者重传定时器超时时，才改变拥塞窗口。

这种基于确认报文的拥塞窗口调整策略，如果以传输轮次为横坐标，则拥塞窗口在慢启动和拥塞避免阶段的走向的确符合图2-2，这一点将在下一章进行分析介绍。

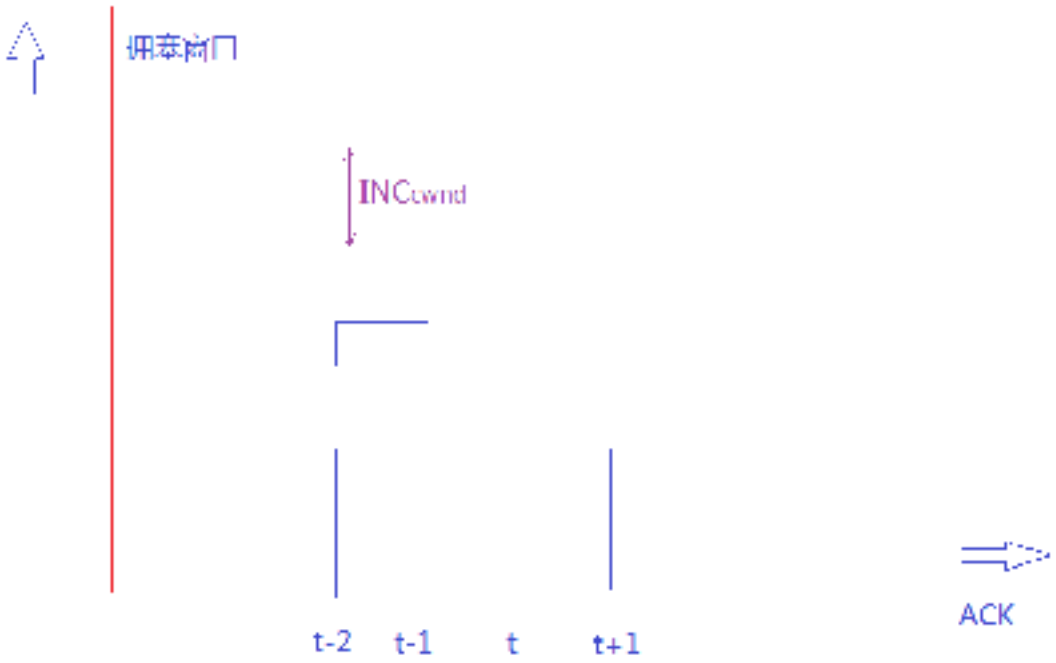


图2-3 慢启动阶段的拥塞窗口实际变化情况。注意，横坐标是收到的ACK的编号，而不是时间。在慢启动阶段，每收到一个ACK，拥塞窗口就改变一次。拥塞窗口的增长值是这么算的：第t-1号ACK的确认号是ACK_{t-1}，第t号ACK的确认号是ACK_t，则收到第t号ACK后，拥塞窗口的增值为INC_{cwnd}=fragment(ACK_t - ACK_{t-1})，fragment(X-Y)表示处在X和Y之间的数据段个数。

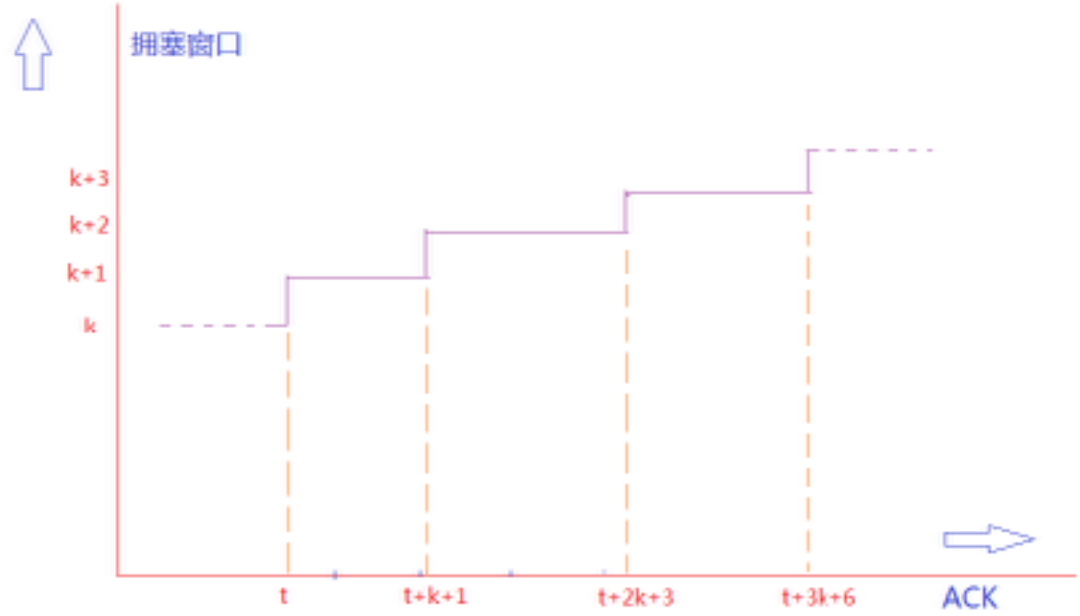


图2-4 Reno算法拥塞避免阶段的拥塞窗口实际变化情况。在拥塞避免阶段，每收到等于当前拥塞窗口的确认ACK个数，则拥塞窗口增1。比如，在收到第t号ACK时，拥塞窗口变为k+1，则得再收到k+1个ACK后，拥塞窗口才变为k+2。

2.2 慢启动与拥塞避免阶段

如上一节所说的，发送端无法准确知道当前的网络状态，只能大概从收到

的ACK确认号和重传超时信息估计当前的拥塞状态。那么，如何在不了解网络状态的情况下，如何使拥塞窗口尽可能大，而又不导致网络拥塞呢？对此，TCP采用的是探测法：

当收到的确认号一直在确认数据，增大拥塞窗口；当感觉到发生拥塞时，把拥塞窗口缩回去。

这也是TCP拥塞控制流程中拥塞窗口调整策略的总体设计思想。

那么，当收到的确认号一直在更新时，如何控制增大拥塞窗口的步数呢？步子迈得太小，则走得慢；太大，容易一不小心就栽进前方的陷阱了。我们很容易想到，当拥塞窗口非常小时，拥塞窗口长得快一些，风险较小，而拥塞窗口比较大时，就得涨得慢一些，不然就容易一下子越过“拥塞临界点”，出问题。那么，如何区分“非常小”与“比较大”的尺度呢？为此，拥塞控制引入了一个变量，即**慢启动阈值**[1]。当拥塞窗口小于慢启动阈值时，处于“慢启动”区，拥塞窗口可以“放心地”涨得快一些，拥塞窗口大于慢启动阈值时，处于“拥塞避免”区，拥塞窗口的调整就“如履薄冰”，涨得慢一些。

那如何估计慢启动阈值呢？TCP的做法是

当探测到可能要发生拥塞时，向下调整慢启动阈值。

数据发送端探测到拥塞只有两种途径，一是重复确认：足够数量的重复确认，可能意味着丢包，而丢包经常被视为拥塞的先兆；二是重传定时器超时。也就是说，慢启动阈值只有在认为发生拥塞了，才进行调整，其余情况下是不会调整慢启动阈值的。在拥塞控制的实现过程中，慢启动阈值只有在进入Recovery和Loss状态瞬间才会发生改变，就是这个思想的体现。

当探测到拥塞时，说明当前时刻的拥塞窗口过大，就调整慢启动阈值一次。不同的拥塞控制算法有不同的慢启动阈值调整策略。比如，Reno算法将慢启动阈值调整为当前拥塞窗口的一半，而Cubic算法调整为当前拥塞窗口的70%。

可见，慢启动与拥塞避免的差别主要在于慢启动阈值，但都表示没有发生拥塞，网络畅通的状态，具体到代码实现上，是差不多的，故Linux内核在实现拥塞控制时，统一用一个open状态来表示慢启动和拥塞避免阶段。

当TCP会话初次建立连接时，还没有发生拥塞，这时慢启动阈值没法估计，故一般可以设置为一个极大值0xffffffff。

[1] 慢启动阈值还有另外一个作用，当退出快速恢复状态时，拥塞窗口等于慢启动阈值，下一章会介绍。

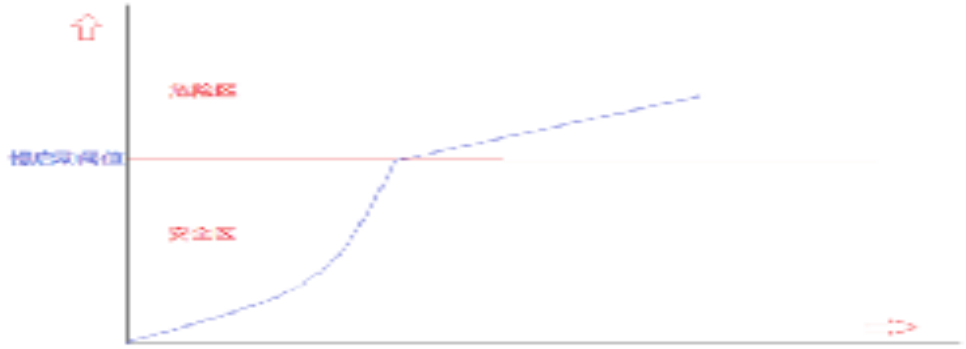


图2-5 慢启动阈值用来分割拥塞窗口增长的“安全区”和“危险区”，两个区域的拥塞窗口增长模式明显不同。

在open状态，拥塞窗口的调整策略可以概括为：

TCP用探测法来控制拥塞窗口，当拥塞窗口小于慢启动阈值时，处于慢启动阶段，拥塞窗口增长得快一些；当拥塞窗口大于慢启动阈值时，处于拥塞避免阶段，增长得慢一些。一旦察觉到发生拥塞，就调整慢启动阈值。

注意这里的表述方式，这是所有拥塞控制算法的设计思想，当具体到实现上，在拥塞避免阶段如何调整拥塞窗口，发生拥塞时如何估计慢启动阈值，不同的拥塞控制算法就有所差异了。比如，Reno算法与Cubic算法在慢启动阶段，都是每当有n个数据被确认，拥塞窗口就自增n(但不能超过慢启动阈值)，而在拥塞避免和慢启动阈值的计算上，就有所不同了[1]。

2.3 disorder状态分析

在Linux内核的TCP实现中，有一个disorder状态，disorder状态实际是慢启动或者拥塞避免阶段往快速重传阶段过渡的一个过渡状态，因为只有收到3个重复确认才开始快速重传。从收到第一个重复确认到收到第三个重复确认报文或者数据确认报文的这段时间，处于disorder状态。

设置disorder状态的必要性在在于，当收到重复确认比较少时，我们还没法判断当前是否发生丢包情况，因为对端收到乱序报文也会发送选择确认，只能持观望的态度。如果连续收到的重复确认足够多，比如，3个重复确认，就进入recovery状态，而如果在收到一两个重复确认后，再收到数据确认报文，则回到open状态(这种情况发生的可能性也很高)。这期间的拥塞窗口保持不变，与open(慢启动或者拥塞避免)和recovery状态的行为是不同的，故有必要设置一个disorder状态。

disorder状态的各参数变化情况及发包行为将在下一章介绍。

---

[1] 不同的TCP拥塞控制算法，其差别主要在拥塞避免阶段和慢启动阈值的调整策略，在慢启动、快速恢复和Loss (重传超时) 等阶段，其行为基本是一样的。这一点将在第四章进行介绍。

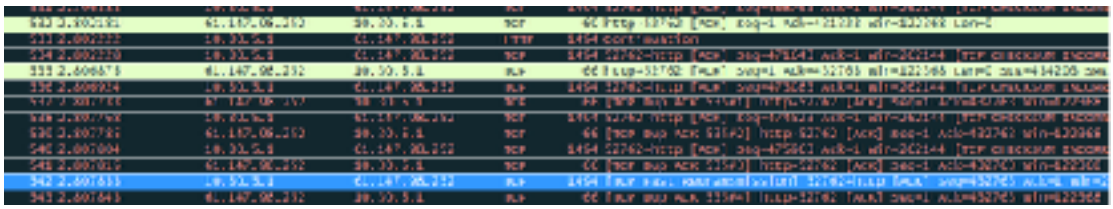


图2-6 重复确认报文的识别。重复确认报文需要同时满足四个条件：(1)序列号与前一个报文相等 (2)确认号与前一个报文相同 (3) 不带数据 (4)window字段不能比前一个报文的window字段大(满足1~3而不满足本条件4的叫window update报文)。图中的537、539、541号报文都是重复确认，但535号报文不是重复确认报文。从收到537号报文到541号报文的这段时间，10.30.5.1处于disorder状态。

2.4 快速重传与快速恢复

快速重传与快速恢复阶段是拥塞控制流程中非常重要也是最难理解的阶段，然而很多资料对本阶段只进行简单粗略的介绍，使人难以对快速恢复有一个全面的理解。本节将对快速重传与快速恢复阶段进行初步介绍，而在下一章对快速恢复阶段进行深入的分析介绍。

快速重传[1]实际上就是当收到对某个报文的3次连续重复确认时，立即重传该报文的过程。如图2-6所示，序号为542的报文就是一个快速重传报文。准确来说，快速重传是一个瞬间动作，而不是一个阶段，所以，在本篇中应该把“快速重传与快速恢复”阶段理解为一个阶段，为表述方便，在下文中将其表述为Recovery阶段。在Linux内核的TCP实现中，快速重传与快速恢复阶段用Recovery状态来表示。当连续收到3个重复确认时，认为重复确认序号所指示的那个报文已经丢失了，于是快速重传这个“丢失”的报文，并进入Recovery状态，开始快速恢复阶段。

那么，什么是快速恢复阶段呢?

快速恢复阶段指的是，从快速重传开始，到网络上没有丢失的报文，可以回到open状态的这段时间。

注意这句话的理解，也就是说，什么时候认为丢失的报文都被确认了，快速恢复就什么时候结束。最开始的Reno算法认为，引发重复确认的报文被确认了，就可以从Recovery状态退出，快速恢复结束。比如说，发送端A连续发送了10~50号数据包，然后，连续收到3个ACK=20的重复确认，则快速重传序列号为20的报文，进入快速恢复状态。接着，收到ACK=25的序列号，说明引发重复确认的20号报文已经被接收，Reno算法认为快速恢复已经结束，则回到open

[1] 叫做“快速”重传是因为它不等重传定时器超时就开始重传，显得非常“快速”。

状态，继续发送新数据。如果按照这种做法绘制Recovery阶段的拥塞窗口走向图，则快速恢复阶段的确像图2-2那样，只占一个传输轮次的时间。

但这样有一个问题，即25~50号报文中也可能有某几个报文丢失了(这种情况叫做“**多包丢失**”)，按照上一段的做法，会继续引发重复确认，导致多次进入Recovery状态，而每次退出Recovery状态时，拥塞窗口会减半，很影响性能。为此，最新的拥塞控制算法采取的做法是

快速重传之前发送的所有数据包都被确认后，才退出快速恢复状态。

注意这句话的理解。在上面的例子中，20~50号报文就是“快速重传之前所发送的所有数据包”，也就是说，直到第50号报文被确认后，才退出快速恢复状态。

这种做法的思想也很好理解：

在拥塞条件下，如果发送的某个数据包丢失了，那么后面发送的数据包也很可能丢失。

在快速恢复阶段，主要的工作，是保证丢失的的报文能尽快到达客户端，以尽早退出快速恢复状态。

快速重传与快速恢复的设计思想：

当收到足够多的重复确认时，认为这个报文丢失了，网络发生拥塞。于是调整慢启动阈值，开始快速重传，并进入快速恢复阶段。直到快速重传之前发送的所有数据包都被对方接收，再退出快速恢复状态，回到open状态[1]。在快速恢复阶段，应想办法尽快将丢失的数据包重传出去，争取早点退出快速恢复状态。

那么，如何实现早点退出快速恢复状态呢？这几乎只有一种方法：支持**选择确认**。选择确认将在第四章进行介绍。

在快速恢复阶段，拥塞窗口是如何调整，发包行为是如何进行的呢？所有的操作系统、拥塞控制算法的思想是差不多的：

在进入快速恢复阶段之前，向下调整慢启动阈值，当退出快速恢复阶段时，拥塞窗口等于调整后的慢启动阈值。

而在快速恢复阶段，拥塞窗口的调整策略和发包行为比较复杂，将在下一章进行介绍。

---

[1] 当发生快速重传时，慢启动阈值会被调整，而退出快速恢复时，拥塞窗口被赋值为慢启动阈值，故实际上是进入拥塞避免状态。

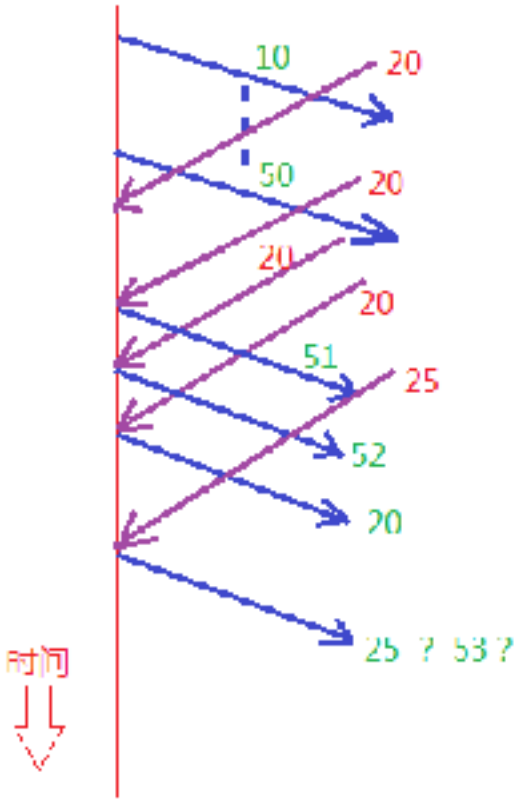


图2-7 disorder与recovery状态的收发包行为。当收到3个确认号为20的重复确认，则快速重传序列号20的报文。当收到ack=25的报文时，最初的Reno算法会退出recovery状态，并发送53号报文。而最新的所有拥塞控制算法都会重传25号报文，直到52号报文被确认后，再退出快速恢复状态。

2.5 Loss状态

重传定时器超时以后所处的状态就是Loss状态。

重传定时器用于在长时间没有收到数据确认报文的情况下保证TCP协议的正常工作。一般情况下，只要在一定时间段内收到数据确认报文，则会关闭或者延迟重传定时器的超时时间，重传定时器是不会运行。而如果长时间没有收到对数据的确认，就会引发定时器超时。

重传定时器超时事件发生时，一般都说明网络状态已经变得极差。所以，一旦发生定时器超时，拥塞窗口变得很小(一般为1)，重新开始慢启动算法。Linux内核对应于这一阶段的状态是Loss状态。当重传定时器超时时，拥塞窗口变为1，同时调整慢启动阈值，认为之前发送出去的所有报文都已丢失，重新开始慢启动算法。

Loss状态的慢启动与open状态的慢启动阶段区别在于，前者采用慢启动算法来重传丢失的报文，直到丢失的数据都被确认后，才发送新数据，而后者没有丢包的情况，只传新数据。

与快速恢复阶段相似，Loss状态的退出时机是这么设定的：

重传定时器超时之前所发送的所有数据包都被确认后，才退出快速恢复状态。

除了之前发送的所有数据包都被标记为丢失，需要重传之外，loss状态和open状态的发包行为还是很相同的。

3. TCP拥塞控制全程分析

作为全文的核心章节，本章将重点介绍拥塞控制各个阶段的拥塞窗口、慢启动阈值的调整情况，并分析其发包行为，并给出Linux内核中的代码级实现。

3.1模型与约定

TCP拥塞控制的核心在于如何实现慢启动、拥塞避免、disorder、快速恢复和定时器超时等阶段来控制拥塞窗口。为此，Linux内核引入了TCP拥塞控制有限状态机来实现这四个阶段。在不考虑显式拥塞通知的情况下，有限状态机有四种状态，即 open、disorder、recovery和loss (见图2-1)。在TCP连接建立以后，拥塞状态能且只能处于这四种状态中的其中一种。下图列出了拥塞状态机所有可能的状态迁移。

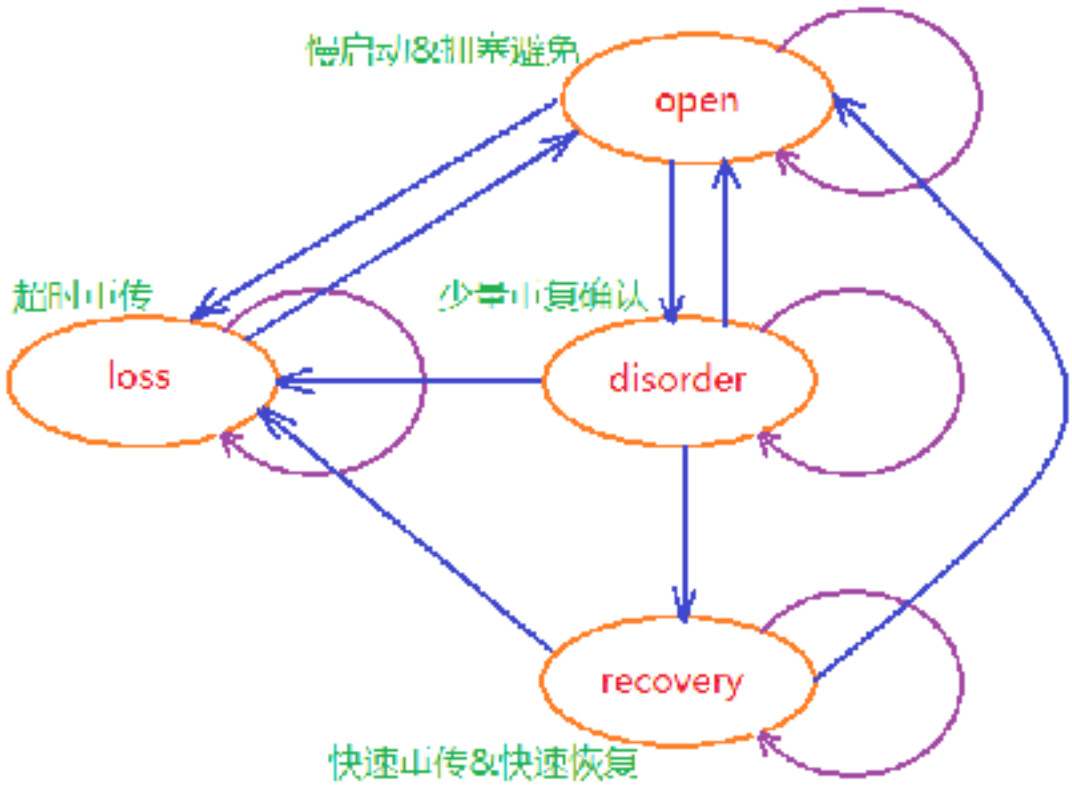


图 3-1 TCP拥塞状态机及状态迁移图，图中每一个箭头表示一种可能的状态迁移，绿色字体表示该状态所对应的阶段。

拥塞控制状态机靠信号驱动，并且只有两种信号对状态机有影响——接收端回复的确认号(ACK)信息和重传定时器超时信息(见图1-1)。当数据发送端收到不同类型的确认信息或重传超时发生时，TCP可能会发生状态迁移，并伴随着拥塞窗口、慢启动阈值及其它状态变量的改变，从而实现拥塞控制。

本章主要以Linux内核为基础，介绍并分析TCP拥塞控制的实现过程。在介绍具体流程之前，我们先介绍拥塞控制相关的基本模型，并定义一些下文需要用到的符号：

(i) Linux内核的TCP发送与重传队列是一个环形的双向链表，并且发送队列、重传队列等是在一个链表上实现的。当应用层有数据要发送时，将数据对应的skb结构插入至链表的尾部，数据的发送时机及发送方式交由TCP协议处理。当数据被接收到的确认号确认时，从链表中移除相应的skb结构并释放相应内存。

在下文中，snd_una表示已发送但未被确认的数据段起始序列号，snd_next表示待发送数据段的起始序列号(见图2-2)。当发送一个报文以后，snd_next会更新；当接收的确认信息确认部分数据已经到达接收端以后，snd_una会更新。

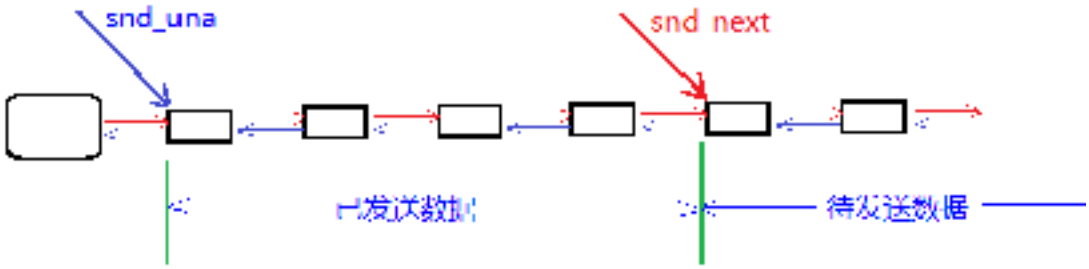


图 3-2 snd_una及snd_next在发送队列中所处的位置，本图画得不标准，发送队列应为环状

(ii) 下文所提到的所有“状态”都是针对数据发送端而言的，用state表示发送端的当前拥塞状态，ack表示发送端接收到的确认报文中的确认号。

(iii) 当发送端收到接收端回复的报文时，先根据其序列号、确认号、报文头部的window字段及报文是否携带数据等信息，可以判断出收到什么类型的报文(正常确认、数据包、重复确认、窗口更新……)。

(iv) 发送端收到接收端回复的ack报文时，如果ack < snd_una或者ack > snd_next，将被视为非法的报文并被丢弃，不会对拥塞状态产生影响。因此，下文分析接收到的确认号信息时，不再考虑ack<snd_una和ack>snd_next的情况。

(v) 本章主要介绍拥塞状态迁移过程和拥塞窗口调整策略，并分析发包行为。在以下各步骤中，先假定正在传输的报文段个数in_flight是已知的。in_flight的具体计算过程将在下一章进行介绍。

3.2 拥塞控制的初始化

在Linux内核的TCP协议实现中，拥塞窗口、慢启动阈值等拥塞控制相关变量的初始化，是在TCP连接状态刚进入established 状态时完成的 (见图2-3)[1]。当TCP连接状态从syn_sent或者syn_recv状态进入established状态时，会初始化拥塞窗口cwnd、慢启动阈值ssthresh及其它相关控制信息。

[1] 初始化位置对拥塞控制的影响不大，在实现自主TCP协议时，拥塞控制相关变量是在会话建立时初始化的。

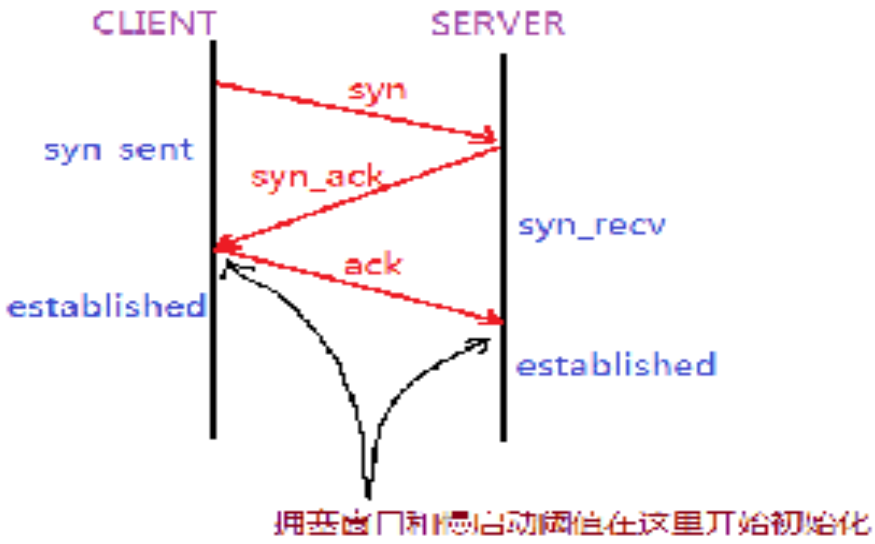


图 3-3 拥塞窗口与慢启动阈值的初始化位置

当第一次建立TCP会话时，Linux内核2.6版本的拥塞窗口初始值是根据RFC3390规范设定的[1]，其初始值与最大段长度MSS大小密切相关，MSS越大,拥塞窗口越小，具体可以参考下表:

表 3-1 拥塞窗口与最大段长度的关系

最大段长度(mss)	mss≤1095	1095 < mss ≤ 1460	mss>1460
拥塞窗口(cwnd)	4	3	2

相比于拥塞窗口设置为一个较小的值，慢启动阈值是用于估计拥塞危险区的，而刚刚建立连接时，网络情况难以估计，故一般将ssthresh设置得够大，Linux内核将ssthresh设置为0x7fffffff，等到察觉到拥塞发生时再调整ssthresh[2]。

值得注意的是，在Linux的TCP实现中，当TCP连接进入四次挥手中的TIME_WAIT或者LAST_ACK状态，准备关闭连接时，会选择性地将当前的拥塞窗口与慢启动阈值保存起来，以作为下次同一条线路的TCP会话建立时的拥塞窗口与慢启动阈值初始值。这么做，可以使拥塞窗口获得比较大的初始值，在一定程度上有利于改善TCP连接的性能。

---

[1] 这一点要求没那么严格，最新的Linux内核做法是这样的：如果在三次握手时，syn包被重传过，则拥塞窗口为3，其它正常情况，拥塞窗口初始值为10。在自主TCP协议中，拥塞窗口初始化为10。

[2] Cubic拥塞控制算法附带了一个hystart算法，该算法可以从收到的ACK信号分布规律和收到ACK信号的时间间隔推测出是否将要发生拥塞了，从而适时调整慢启动阈值，平稳地从慢启动状态过渡到拥塞避免状态。Hystart算法不是拥塞控制所必须的，在此就不再介绍了。

3.3慢启动与拥塞避免阶段

Linux内核用open状态来表示TCP连接处在慢启动或者拥塞避免阶段，两者的差别在于拥塞窗口与慢启动阈值的大小关系，但都表示网络畅通的状态。

这一阶段的拥塞窗口调整策略可以概括为:

在慢启动和拥塞避免阶段，如果有增大拥塞窗口的必要，则按照慢启动或者拥塞避免算法调整拥塞窗口。

每一轮数据发送[1]时，实际可以发送的数据包个数受到三个条件的限制，即，发送队列的数据包个数、接收端的接收窗口大小和发送端的拥塞窗口大小。所以，并不是每一轮数据发送时，实际发送的数据包都卡在“拥塞窗口”这一环。当某一轮数据的发送不被拥塞窗口限制时，就算接着收到确认数据的ack，也不会调整拥塞窗口。也就是说，没有增大拥塞窗口的必要时，不调整拥塞窗口。

然而，在绝大部分情况下，实际可以发送的数据包个数都是因为受限于拥塞窗口(pipe ≤ cwnd)。每一轮次的开始和结束，以pipe == cwnd的情况居多，所以，在下文总结各个阶段的发包行为时，都是以pipe == cwnd 为前提的。

慢启动阈值ssthresh只有在察觉到发生拥塞时才会发生变化，由于open状态没有发生拥塞，期间ssthresh的值并不改变。

3.3.1 慢启动阶段的拥塞窗口调整策略

在慢启动阶段，拥塞窗口的调整策略可以概括为

有多少个数据包被确认，则拥塞窗口增加多少。

下面我们从这个原则推导慢启动阶段的发包行为。

每一轮次开始前, 正在传输的报文pipe和拥塞窗口cwnd分别为

$$pipe = k1,\ cwnd = k2\ (满足 k1 \le k2)$$

然后，收到了一个ack,该ack确认了n个数据包，则有

$$pipe = k1 - n$$

按照“在慢启动阶段，有多少个数包确认，则拥塞窗口增加多少”的原则，则

$$cwnd = k2 + n$$

于是，本轮可以发送的数据包个数

$$delta = cwnd - pipe = (k2 + n) - (k1 - n) = 2n + (k2 - k1) \ge 2n$$

由于发送端经常有大量的数据要发送，k2 == k1，则经常有

$$delta = 2n$$

---

[1] 发送端“收到一个ack,调整拥塞窗口，发送一波数据包”，这个过程就构成“一轮”

故慢启动的发包行为可以概括为

收到的ACK确认了n个数据包，则可以发送2n个数据包。

下图演示的就是一个典型的慢启动阶段。这种收发包行为在TCP连接刚开始建立时，经常看得到。

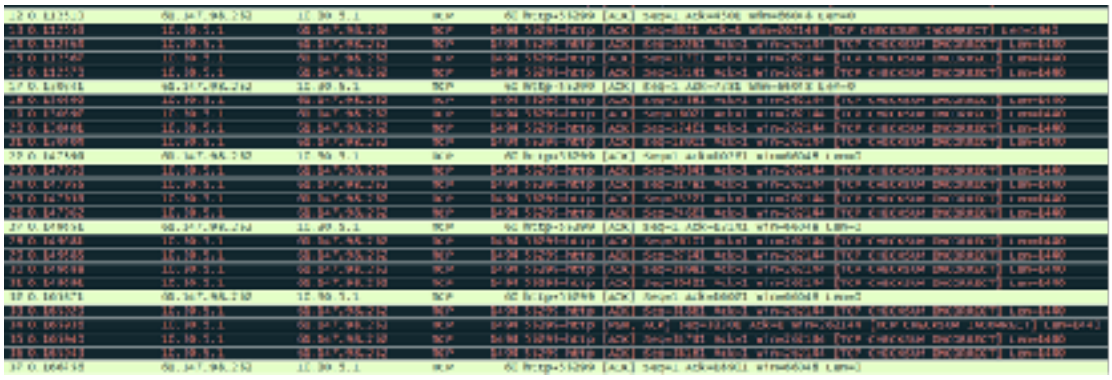


图3-4 Windows操作系统中典型的慢启动阶段。在慢启动阶段，有多少个数据段被确认，则拥塞窗口增加多少。这在发包行为上表现为，每个ACK确认n个数据包，则发送端可以发送2n个新数据包。图中每个ACK确认了2个数据包，故发送端发送4个数据包。

一般资料将慢启动阶段的拥塞窗口调整策略概括为“在慢启动阶段，每经过一个传输轮次，拥塞窗口翻倍”，这与“在慢启动阶段，有多少个数据包被确认，则拥塞窗口增加多少”说的其实是一个意思。前者的表述更适于作为教材，而后者的表述更适于计算机语言实现。

由于慢启动阶段的拥塞窗口增长速度已经够快，而且慢启动阶段在TCP会话整个生命周期中所占的时间比例并不多[1]，故几乎所有拥塞控制算法都采用这种拥塞窗口调整策略。

3.3.2 拥塞避免阶段的拥塞窗口调整策略

拥塞避免阶段的拥塞窗口调整策略可以概括为

每收到一定个数的数据确认ACK，拥塞窗口加1。

这里使用“一定个数”，是因为对于不同的拥塞控制算法来说，这个“一定个数”的值是不同的，一般来说，它与当前的拥塞窗口有关

$$t = f(cwnd)$$

比如，对于Reno算法来说，f(cwnd) = cwnd。但不管怎样，t不是一个很小的值，这使得拥塞窗口在拥塞避免阶段表现为

拥塞窗口在维持一段时间内的恒定值后，自增1。

[1] 只要不发生重传定时器超时，TCP会话只在连接刚建立时处于慢启动阶段。

下面分析拥塞避免阶段的发包行为。

在收到ack之前，正在传输的报文个数pipe和拥塞窗口cwnd分别为

$$pipe = k1, cwnd = k2$$

收到ack之后，ack确认了n个数据，则

$$pipe = k1 - n$$

在拥塞避免阶段，调整后的拥塞窗口为

$$cwnd = k2 \text{ 或者} cwnd = k2 + 1$$

取k1 == k2， 则可以发送的数据包

当cwnd = k2时

$$delta = cwnd - in\_flight = k2 - (k1 - n) = n + (k2 - k1) = n,$$

当cwnd = k2 + 1时

$$delta = (k2 + 1) - (k1 - n) = n + 1$$

由于cwnd = k2的情况占大多数，则拥塞避免阶段的发包行为概括为

收到的ACK确认了n个数据包，则发送n个数据包，偶尔发送n+1个数据包。

图3-5演示了拥塞避免阶段的收发包行为，这种报文收发模式在TCP连接中经常看到。

由于拥塞避免阶段在TCP连接的整个生命周期中占了很大的一部分时间，对TCP协议的性能有着重要的影响，故所有拥塞控制算法都会修改拥塞避免阶段的窗口调整策略。“在拥塞避免阶段，每经过一个RTT，拥塞窗口加1”的情况只会在Reno算法中看到。不管怎样，所有算法的拥塞避免阶段，拥塞窗口都涨得比较慢，但慢的程度就各不相同了。

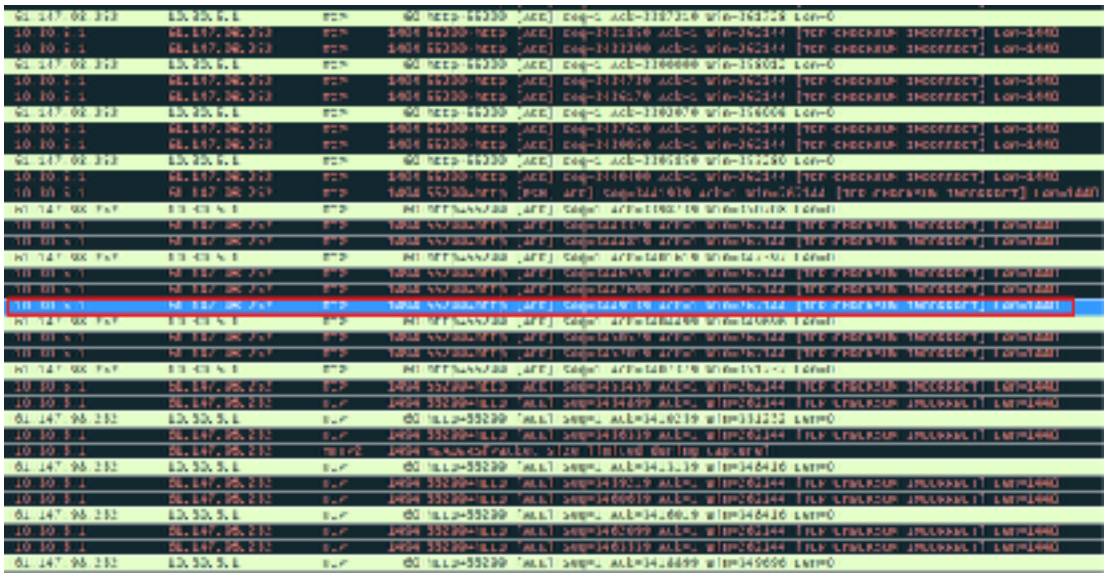


图3-5 典型的拥塞避免阶段。有多少个数据包被确认，就发送多少个新的数据包，偶尔突然多

发了一个数据包(图中标红线的那个数据包)，那是因为收到ACK = 3401619的确认报文时，拥塞窗口自增1，然后，又恢复原来的收发包模式。

3.3.3 open状态的代码级实现

处在open状态的TCP发送端，按照in_flight≤cwnd的原则从发送队列读取数据并发送出去以后，根据所接收的信号[1]不同，可能有多种不同的结果，具体如表3-2所示。

表 3-2 open状态迁移表

信号	下一个状态	备注
snd_una < ack ≤ snd_next	open	见3.3.3条
ack = snd_una	disorder	见2.4.1条
time_out	loss	见2.6.1条

下面将介绍snd_una ≤ ack ≤ snd_next时的处理流程。

处在open状态的TCP连接，当收到一个ack信号，并且满足snd_una < ack ≤ snd_next ack，说明发送出去的数据已经顺利到达了对端，即ack确认了新数据的到达。这时，将调用慢启动或者拥塞避免算法更新拥塞窗口，并且将拥塞状态将保留在open状态。

首先，Linux内核先判断是否有必要调整拥塞窗口，避免不必要的拥塞窗口调整。是否有必要调整拥塞窗口的原则是：看看目前的拥塞窗口是否够用。对此，不同的内核版本有不同的处理方式。2.6版本的Linux内核先计算当前拥塞窗口大小与收到本ack之前的正在传输的报文段个数prior_in_flight[2] 的差值δ，如果δ 小于或者一个计量值reordering，则进行拥塞窗口调整，否则不需调整：

```
δ = cwnd – prior_in_flight;
if (δ < reordering) {
    Adjust_congestion_window( );
} else {
    return;
}
```

reordering这个变量有特殊的含义，它的含义就是一般所理解的“每收到3个

[1] 将接收到ack事件和重传定时器被触发事件统称为“信号”，下同。

[2] prior_in_flight保存in_flight调整之前的in_flight值，每次收到ack后会调整in_flight的值。

重复确认号时就开始快速重传”中的“3”，reordering将在第四章进一步讨论。

而最新的Linux内核3.17版本的做法更可取。它的做法是，每进行一轮数据发送时，如果因为拥塞窗口的限制导致没有将发送队列的所有数据段发送出去，则将变量is_cwnd_limited置位。当收到报文时，如果 cwnd ≤ snd_ssthresh，则满足cwnd < 2 * max_packets_out[1]的条件时才需要调整拥塞窗口；当处于拥塞避免阶段，则只有is_cwnd_limited已经被置位才进行拥塞窗口的调整：

```
if (( cwnd ≤ snd_ssthresh && cwnd < max_packets_out) | | is_cwnd_limited) {
    Adjust_congestion_window( );
} else {
    return;
}
```

Linux在open状态添加一个是否需要调整拥塞窗口的判断，是基于这样的考虑：数据发送率不仅受拥塞窗口限制，还受到待发送数据量的多少和发送窗口限制。当应用层需要发送的数据不多，每轮发包时TCP都能将发送队列中的所有数据段发送出去，那么就没有必要调整拥塞窗口。否则，在待发送数据很少时按照慢启动或者拥塞避免算法一直增大拥塞窗口，就会使得拥塞窗口变得非常大，这时候要是突然有大量应用层数据需要发送，拥塞窗口就失去了拥塞限制作用，可能会导致网络拥塞。通过添加一个是否需要调整拥塞窗口的判断，保证数据发送速率不会陡增，保证TCP数据段发送过程平稳进行。

当需要调整拥塞窗口时，会判断当前的拥塞窗口是否小于慢启动阈值，是则调用慢启动算法，否则采用拥塞避免算法，下面介绍内核3.17版本的open拥塞控制实现过程。

慢启动：将cwnd增大到cwnd+acked，如果此时cwnd ＞ ssthresh，再调整为ssthresh+1。

```
cwnd += acked;
if (cwnd > ssthresh) {
    cwnd = ssthresh +1 ;
}
```

拥塞避免：拥塞避免算法要求每收到T个ack后，cwnd自增1，为此， TCP会话维护一个变量snd_cwnd_cnt，初始值为0，在调用拥塞避免算法时，判断snd_cwnd_cnt 是否大于或等于T，是则拥塞窗口加1，并将snd_cwnd_cnt重设为0，否则snd_cwnd_cnt加1，cwnd不变：

```
if (snd_cwnd_cnt ≥ T ) {
```

[1] max_packets_out表示最近一轮数据发送时的packets_out

```
    cwnd +=1;
    snd_cwnd_cnt =0;
  } else {
    snd_cwnd_cnt++;
  }
```

在拥塞避免阶段，每一种拥塞控制算法的这个“T”，都有它自己的一套计算公式。比如，对于Reno算法来说，“T”就是当前的拥塞窗口大小。

3.4 disorder阶段

Disorder阶段表示发送端在open状态收到重复确认时所处的阶段。设置disorder状态的合理性在于，当收到重复确认时，确认号对应的报文可能在网络上丢失了，也可能还在网络上传输，而一些后于其发送的报文段却先到达接收端了。未获得足够的信息而过早地将报文判定为丢失并进入快速重传状态，会引起吞吐率的下降。为此，设定disorder状态，用于权衡两种可能的情况。当收到足够多的重复确认信息时才开始快速重传，而期间收到的ack不再是重复确认，而是确认了新数据时，将回到open状态。

3.4.1 disorder阶段的拥塞窗口调整策略

disorder阶段的拥塞窗口调整策略可以概括为

拥塞窗口在disorder阶段保持不变。

作为一种“观望”的状态，disorder阶段的拥塞窗口不增也不减。

下面分析其发包行为。

收到重复确认之前，

$$pipe = k1, cwnd = k2$$

收到重复确认[1]时，

$$pipe = k1 - 1, cwnd = k2$$

则可以发送的数据包个数

$$delta = cwnd - pipe = k2 - (k1 - 1) = 1 + k2 - k1$$

取k2 == k1，则

$$delta = 1$$

故disorder状态的发包行为可以概括为

每收到一个重复确认，就发送一个新的数据包。

事实上，当设计人员在设计disorder状态时，他们的想法是，每收到一个重复确认，就发送一个新的数据包，以触发对端回复更多的信息，这才有“拥塞窗

[1] 在介绍pipe的调整方式之前，先暂且认为“每收到一个重复确认，表示有一个报文离开网络”。

口在disorder阶段保持不变”这个结果。即，**发包行为的设计思想决定拥塞窗口的调整策略**，这在其它阶段也是一样的，不过我们在分析TCP拥塞控制的时候，反过来进行了。

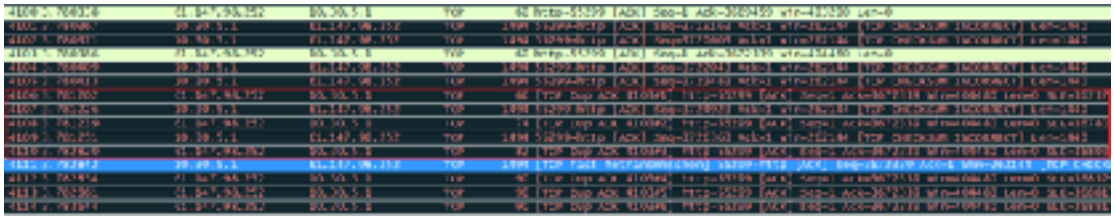


图3-6 典型的disorder阶段(一)。在图中，发送端(10.30.5.1)从收到4106号报文，至收到4110号报文为止的这段时间，处于disorder状态。注意期间发送端的发包行为：每收到一个重复确认，就发送一个新的数据包，直到收到第三个重复确认后，开始快速重传(进入recovery状态)

3.4.2 disorder阶段的代码级实现

表 2-3 disorder状态迁移表

信号	下一个状态	备注
snd_una < ack ≤ snd_next	open	见2.4.2条
ack=snd_una (duplicate ack)	disorder/ recovery	disorder见3.4.2.1款 recovery见2.5.1条
time_out	loss	见2.6.1条

3.4.2.1 disorder阶段收到重复确认

由于在open状态和在disorder状态期间收到重复确认(往下用duplicate ack表示)时的处理流程差不多，遂合并介绍。

当收到duplicate ack时，如果处在open或者disorder状态，先将sacked_out加1。然后，判断sacked_out ≥ reordering是否成立，当成立时将进入recovery状态并开始快速重传算法，本条主要介绍sacked_out < reordering的情况。

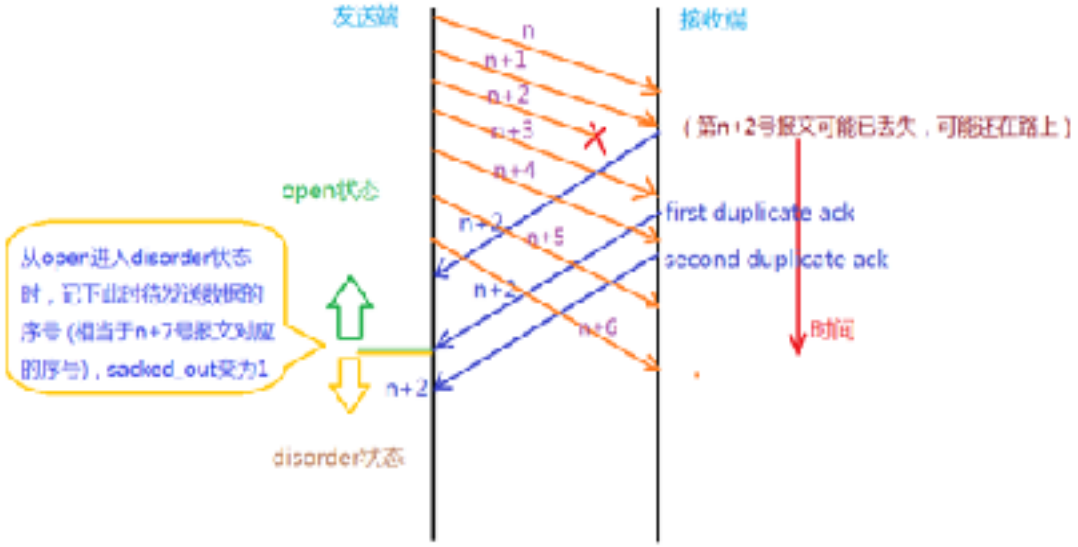


图3-7 open-disorder状态示意图

用伪代码表示这一过程：

```
if (duplicate_ack && (state == open | | state==disorder )) {
    sacked_out++;
  if  (sacked_out ≥ reordering) {
      do_recovery();
  } else {
        do_disorder();
  }
}
```

当判断为sacked_out < reordering时，即此时收到的duplicate ack还不够多，如果此时的状态不是disorder，将状态调整为disorder：

```
function do_disorder(tp) {
    if (tp.state == open) {
        tp.state = disorder;
    }
}
```

综上，当在open或者disorder状态收到重复确认，并且重复确认的个数不够多时，Linux内核不调节拥塞窗口和慢启动阈值(即，如果这时满足in_flight < cwnd，并且发送队列中还有待发送数据，则会继续发送新数据)，不会立即重传dupliate ack对应的报文段。

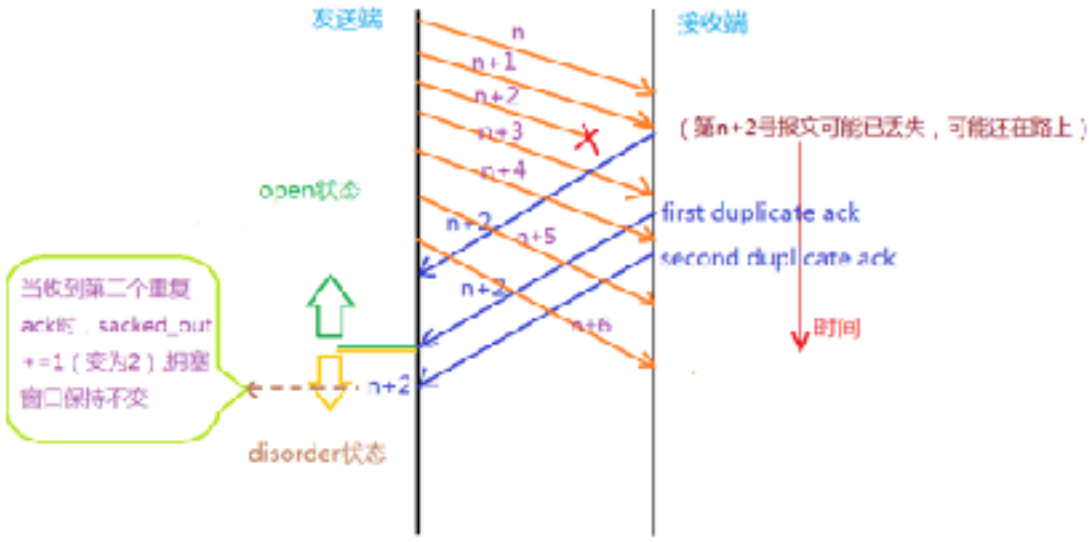


图 3-8 disorder-disorder状态示意图

3.4.2.2 disorder阶段收到数据确认报文

当处于disorder状态的TCP发送端收到的ack确认了数据时，说明先前引起重复确认的报文段已经到达了接收端。这时，将sacked_out重置为0，state设置为open，重新回到open状态：

```
function tcp_try_keep_open(tp, flag) {
    if ( flag & FLAG_SND_UNA_ADVANCED) /* 有数据被确认时,flag置FLAG_SND_UNA_ADVANCED位 */
    {
        tp.sacked_out = 0;
        tp.state = open;
    }
}
```

可见，从open到disorder，再从disorder回到open状态这一过程中，拥塞窗口和慢启动阈值都不会发生改变，相当于一个平台期。

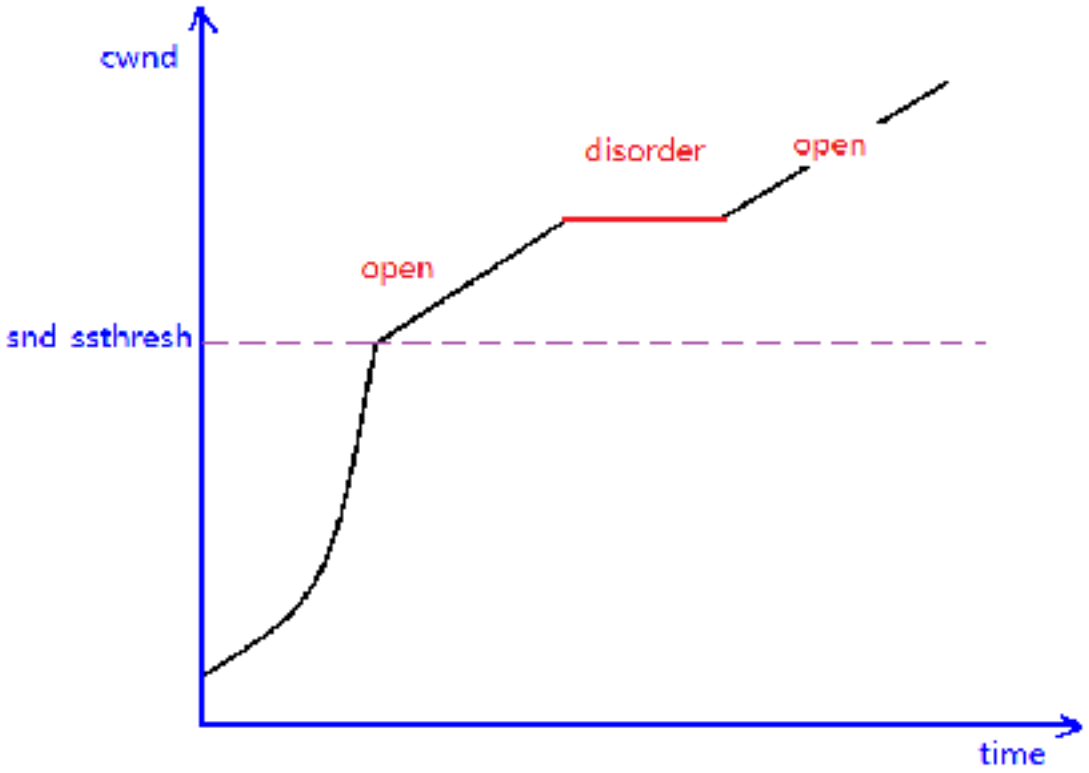


图 3-9 disorder状态的cwnd与snd_ssthresh

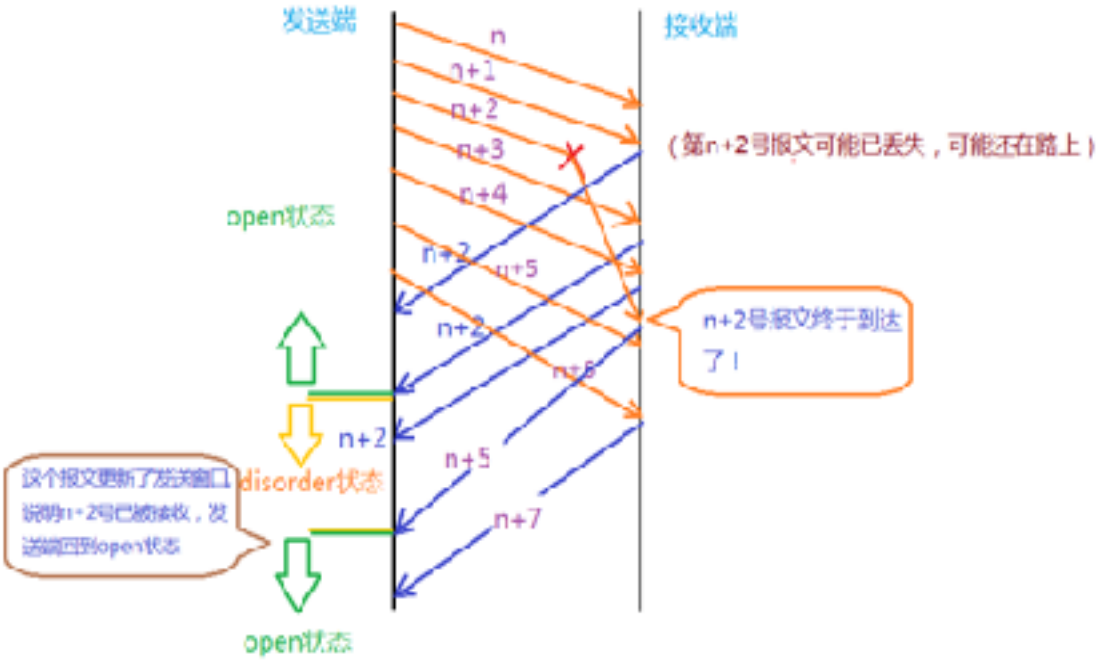


图 3-10 disorder-open状态示意图

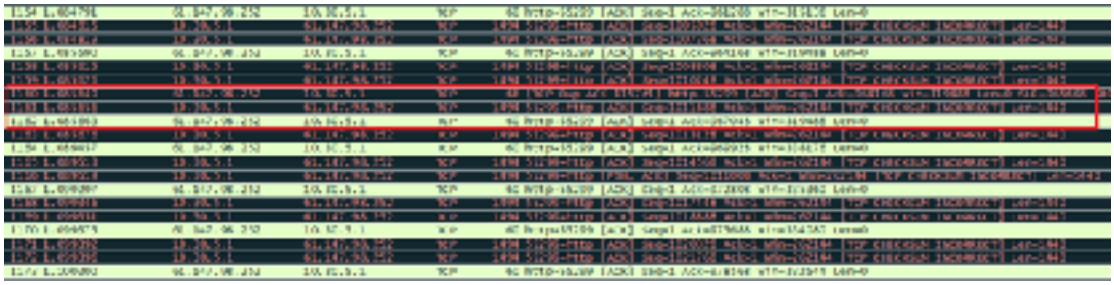


图 3-11 典型的disorder状态(二)。在图中，发送端收到一个重复确认后，再次收到的ACK(序号1162的那个报文)就确认新数据了。这时，它将回到open状态，所以接下来它的发包行为就与拥塞避免阶段的发包行为一致了。

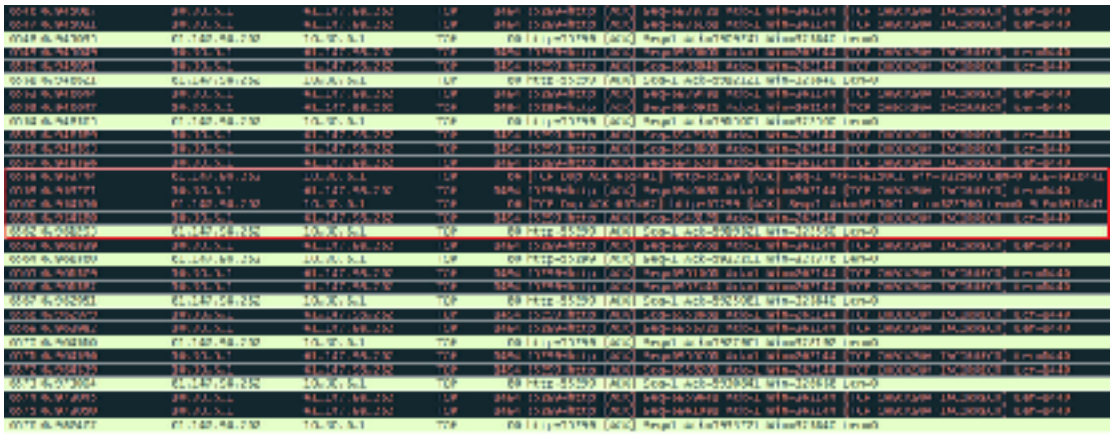


图 3-12 典型的disorder状态(三)。与图3-11相似，发送端收到两个重复确认，再收到的ACK就确认数据了，于是又回到拥塞避免状态。

3.5 快速重传与快速恢复阶段

当在disorder状态收到足够多的重复确认(一般是3个)时，将进入快速恢复阶段(即recovery状态)。由于快速恢复阶段可能发生拥塞了，故其慢启动阈值和拥塞窗口都会进行调整，下面将逐步介绍其调整策略及发包行为。

3.5.1 快速恢复阶段的拥塞窗口调整策略

先对快速恢复阶段的拥塞窗口和慢启动阈值调整策略做一个总结:

进入快速恢复阶段之前，先调整慢启动阈值，等到退出快速恢复阶段时，拥塞窗口等于慢启动阈值。在快速恢复期间，不同的快速恢复算法有不同的拥塞窗口调整策略。

先解释一下这个总结的前半句。当收到3个重复确认，开始快速重传之前，先把慢启动阈值调整一下

$$ssthresh = k * cwnd$$

对于不同的拥塞控制算法，这个k值并不相同。比如，Reno算法和Compound算法，k = 0.5；Cubic算法, k = 0.7; Bic 算法, k = 0.8 。k值总是小于1，

但并不是k值越大越好。

等到退出快速恢复状态时，拥塞窗口等于慢启动阈值

$$cwnd = ssthresh = k * cwnd \quad (k < 1)$$

可见，**一旦经历快速恢复阶段，拥塞窗口就会下降**。拥塞窗口的下降就会降低发往网络中的数据包个数，TCP协议也就是靠这种工作模式来尽量避免拥塞的。

下面将重点介绍快速恢复阶段的拥塞窗口调整策略。

在快速恢复阶段，拥塞窗口的调整策略可以分为两种：**突降式和比例减少式**。突降式指的是一旦进入快速恢复状态，就执行“cwnd = ssthresh = k * cwnd”操作，拥塞窗口突然降低为调整后的慢启动阈值。Windows操作系统所采用的拥塞控制算法就是采用这种策略。一般资料和教材，比如《计算机网络》中所说的“一旦收到三个重复确认，则慢启动阈值与拥塞窗口减为当前拥塞窗口的一半”，说的也是这种突降式拥塞窗口调整策略。

比例减少式拥塞窗口调整策略指的是，拥塞窗口缓慢过渡至ssthresh (即 k * cwnd)。下面先以Windows的Compound拥塞控制算法作为实例，介绍快速恢复阶段的发包行为，再分析平缓式拥塞窗口调整策略。

3.5.2 突降式拥塞窗口调整策略实例分析

通过分析Windows操作系统的快速恢复阶段，将Compound算法的快速恢复阶段总结如下：

当发生快速重传时，慢启动阈值和拥塞窗口变成当前拥塞窗口的一半。在快速恢复期间，都会按照in_flight ≤cwnd的原则发送新数据包。如果收到部分确认报文，其序列号对应的报文被标记为丢失并重传出去。直到发生快速重传前发送的所有数据包都被确认，退出快速恢复状态。

补充解释一下“部分确认(Partial Acknowledgement)”：当进入Recovery阶段时，所有未被确认的数据构成一个区间(在代码级实现中，这个区间就是send_next ~ high_seq)，如果收到的确认号落在这个区间，就是一个“部分确认”。部分确认表示之前发送的所有数据包有一部分被确认了。

下面将截取一段报文来分析推导Compound算法的快速恢复阶段，以加深对拥塞控制快速恢复阶段的理解。

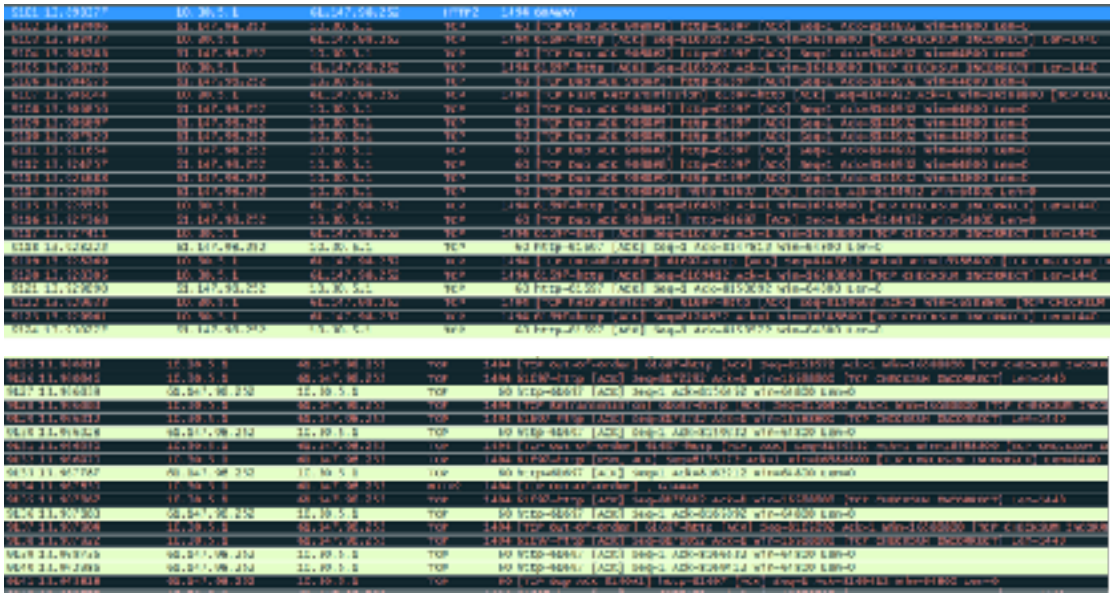


图3-13 Compound 拥塞控制算法的快速恢复阶段。Compound算法是Windows操作系统所采用的拥塞控制算法，它在快速恢复阶段采用突降式拥塞窗口调整策略。所以经常在Windows系统中见到这种类型的快速恢复阶段。

如图3-13所示，数据发送端为10.30.5.1，接收端为61.147.98.252，下面将逐条分析快速恢复阶段的收发包行为，解释它为什么这么发包，为分析方便，po表示packets_out，so表示sacked_out，ro表示retrans_out，lo表示lost_out。

(1) 发送端接收了1904号报文后，发送端发送了1905号报文，此时

$$po = (8165092 + 1440 - 8144932) / 1440 = 15$$

$$so = 2$$

$$ro = 0$$

$$lo = 0$$

则

$$cwnd = pipe = (po + ro - so - lo) = 13$$

(2) 收到1906号报文，则拥塞窗口和慢启动阈值变为当前拥塞窗口的一半：

$ssthresh = cwnd = cwnd/2 = 13/2 = 6$

seq = 8144932的报文被认为已丢失并被重传出去(1907号报文)，则此时

$$po = 15,\ so = 3,\ ro = 1,\ lo = 1$$

$$pipe = (po + ro) - (so + lo) = 12$$

此时 $cwnd - pipe = 6 - 12 < 0$，故不发新数据包。

(3) 收到1908号报文，

$so = 4,\ pipe = 11,\ cwnd - pipe = 6 - 12 < 0$, 不发新数据包

收到1909号报文，

so = 5, pipe = 10, cwnd – pipe < 0, 不发新数据包

……(和上面相似)

收到9113号报文

so = 9, in_flight = 6, cwnd – pipe = 6 – 6 = 0, 不发新数据包

收到9114号报文，

so = 10, in_flight = 5, delta = cwnd – in_flight = 6 – 5 = 1,故可以发送1个新数据包——9115号报文就是这么产生的。发送9115号新数据包后, packets_out 自增1，packets_out = 16, pipe = 6

收到9116号报文，

so = 11, pipe = (po + ro) – (so + lo) = (16 +1) – (11+1) = 5

delta = cwnd – in_flight = 6 – 5 = 1, 可以发送1个新数据包

发送9117号新数据包后，po = 17, in_flight = cwnd = 6,其余参数不变

(4) 收到9118号报文，该报文确认的数据包个数:

acked = (8147812 – 8144932) / 1440 = 2

于是 po = po – acked = 17 – 2 = 15

so = so – (acked – 1) = 11 – (2-1) = 10

lo = 0

ro = 0

然后，序列号为8147812的报文被标记为丢失，并被重传出去(就是9119号报文)，则

lo = 1, ro = 1

pipe= 15 + 1 – 10 – 1 = 5

delta = cwnd – pipe = 1，故发送9120号报文，此后po= 16, pipe = cwnd =6

(5) 9118~9138号报文的推导过程与(4)差不多，故不再赘述。直到发送9138号 报文以后，此时

po = 10, so = 4, lo=1, lo=1, cwnd = pipe = 6

当收到9139号报文时，由于确认号ACK=8166532 ≥(8165092 + 1440)，说明进入快速恢复阶段之前发送的所有报文都被接收了。

此时，po = 9, so = 0, lo = 0, ro = 0, cwnd = 6，再次回到拥塞避免状态，快速恢复状态至此结束。

当收到9140号报文以后，po = 7, so=lo= ro = 0

由于收到9139和9140号报文时, 不满足pipe < cwnd，故都不发新数据包。

至此，一个完整的突降式快速恢复阶段分析结束。

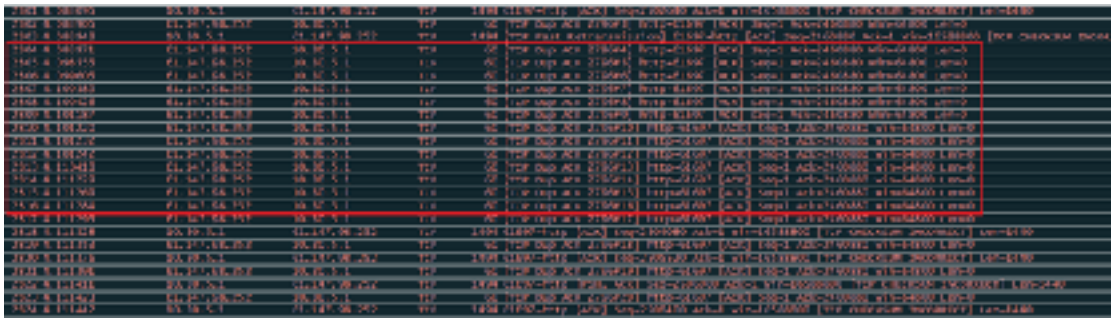


图3-14 典型的突降式快速恢复模式，这种模式也叫做“RFC3517 Fast Recovery”，因为RFC3517就是这么规定的。此模式的特点是，拥塞窗口突降。经常表现为在快速重传后，收到一系列重复确认时，没有任何反应(这段时间叫做silent period at start of recovery)。

3.5.3 PRR算法

与突降式相对应的是比例减少式拥塞窗口调整策略，也叫做PRR (Proportional Rate Reduction)算法。PRR比例减少算法指的是，在快速恢复阶段，拥塞窗口会按比例k收敛至给定的慢启动阈值(ssthresh = k * cwnd)，并在退出快速恢复状态时，等于慢启动阈值。最新的Linux内核拥塞避免阶段都采用PRR算法[1]。

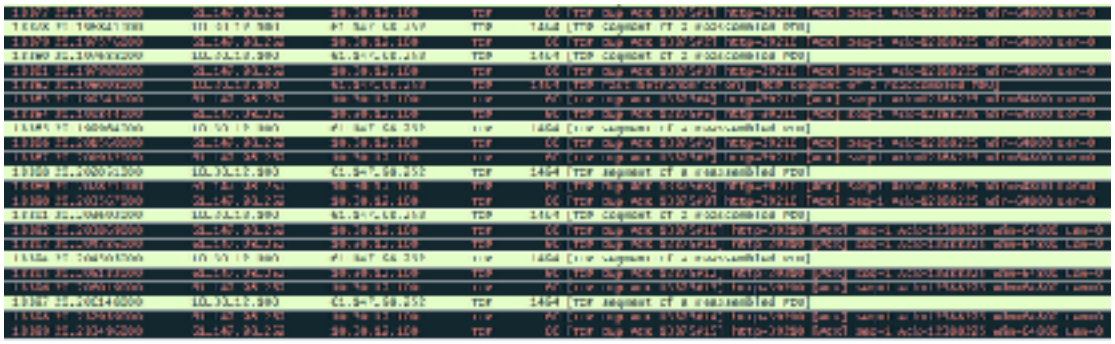


图3-15 典型的Reno算法快速恢复阶段(一)。表现为每收到两个重复确认就发送一个新的数据包。

[1] 以前Linux内核，比如2.6版本，在快速恢复阶段采用Rate-Halving算法，不过该算法后来被淘汰了，在此不再介绍。

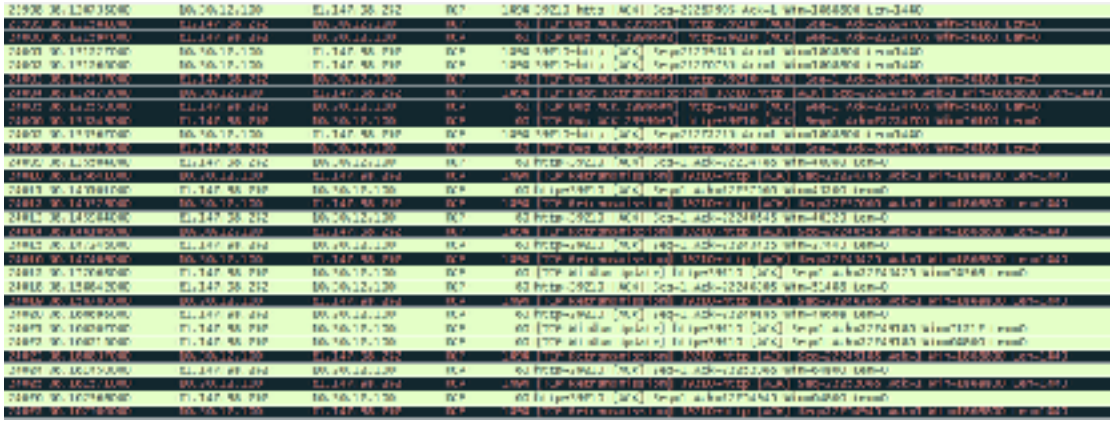


图3-16 典型的Reno算法快速恢复阶段(二)。当收到部分确认时，确认号对应哪个报文，就重传哪个报文。

在Recovery阶段，慢启动阈值的收敛因子是α,则

当有k个报文到达接收端时，若正在传输的报文段个数大于慢启动阈值，拥塞窗口可以累积发送的数据包个数为CEIL(kα)，当pipe < ssthresh时，cwnd = ssthresh。

每当有n个数据包到达接收端，则可以发送 delta * n 个数据包

对于Cubic算法来说，delta = 0.7 ，即平均每收到一个重复ack就发送0.7个报文。

在Recovery阶段，每收到一个ack，都会调整拥塞窗口。拥塞窗口的调整策略是:

If Pipe > ssthresh

Cwnd = pipe + (prr_delivered * ssthresh/prior_cwnd – prr_out)

Else

Cwnd = ssthresh – pipe

```
DeliveredData = change_in(snd.una) + change_in(SACKd)
prr_delivered += DeliveredData
pipe = (RFC 6675 pipe algorithm)
if (pipe > ssthresh) {
// Proportional Rate Reduction
sndcnt = CEIL(prr_delivered * ssthresh / RecoverFS) -
prr_out
} else {
// Two versions of the Reduction Bound
if (conservative) { // PRR-CRB
limit = prr_delivered - prr_out
} else { // PRR-SSRB
limit = MAX(prr_delivered - prr_out, DeliveredData) +
MSS
```

```
    }
    // Attempt to catch up, as permitted by limit
    sndcnt = MIN(ssthresh - pipe, limit)
    }
    On any data transmission or retransmission:
    prr_out += (data sent) // strictly less than or equal
to sndcnt
```

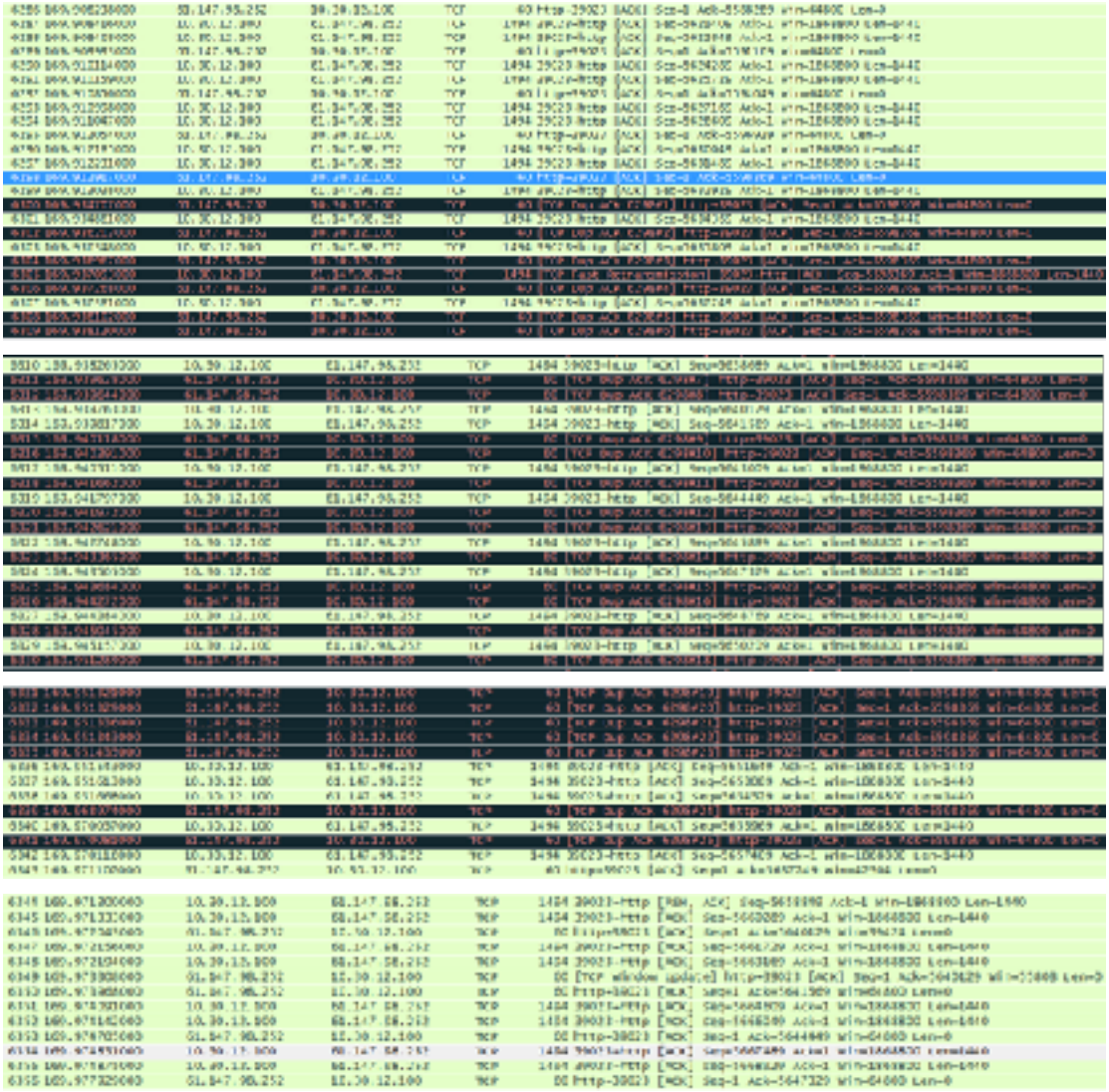


图3-17 Cubic算法的一个完整的Recovery阶段。Cubic算法在Recovery阶段采用PRR算法。 下面将从理论上分析它为什么这么发包。

上图是典型的Cubic算法的Recovery阶段。为了加深对拥塞控制的理解，下面将逐条分析，解释它为什么这么发包。

(1) 在分析之前，先要学会从抓取的报文中区分每一轮的开始和结束。“收包-发包”构成一轮。

从6286~6299号报文，我们可以分析出，收到一个ack后，会在120~160毫秒

左右发送这一轮次的报文，同一轮次的报文发送时间差在40~60毫秒左右。

(2) 往下分析。收到6304号报文时，进入Recovery阶段，我们先分析拥塞控制各个参数的值

Cwnd = 25

Po = (5635809 + 1440 – 5598369) / 1440 = 27

So = 3

Lo = 1

Ro = 0

Pipe = po + ro – so – lo = 23

Ssthresh = 25 * 0.7 = 17

然后，进入prr算法计算，

Pdel = 1

Pout = 0

Pipe > ssthresh 则

Sndcnt = (25 * 0.7*1 + 25 -1) /25 – 0 = 1

故 cwnd = sndcnt + pipe = 24

接着，开始快速重传，ro = 1, pipe = cwnd = 24，故不发新数据

此轮到此结束。Pdel = 1 , poute= 1(因为重传了一个数据包)

(3) 收到6306号报文。

Pipe = 23, pdel = 2,

Sndcnt = (25 * 0.7*2 + 25 -1) /25 – 1 = 1,

Cwnd = sndcnt + pipe = 24

即，可以发送一个数据包，6307号报文被发送出去。

此轮结束后, cwnd = pipe = 24, pdel = 2, pout = 2

(4) 收到6308和6309号报文。由于这两个报文的收取时间很近(18毫秒)，故当作一轮处理，

Pipe = 22, pdel = 4

Sndcnt = (25 * 0.7 * 4 + 25 - 1) /25 – 2 = 1

Cwnd = sndcnt + pipe = 23, 可以发送1个数据包,故6310号报文被发送出去。

此轮结束后， cwnd = pipe =23, pdel = 4, pout = 3

(5) 收到6911和6912号报文。同(4)，该两个报文当作一轮处理。

Pipe = 21, pdel = 6,

Sndcnt = (25 * 0.7 * 6 + 25 – 1)/25 – 3 = 129/25 – 3 = 2

Cwnd = sndcnt + pipe = 23, 故可以发送2个数据包。故这一轮发送了两个数据包，即6913和6914号报文。

此轮结束后，cwnd = pipe = 23, pdel = 6, pout = 5

(6) 收到6315号报文。

Pipe = 22, pdel = 7,

Sndcnt = (25 * 0.7 * 7 + 25 -1)/25 – 5 = 0

Cwnd = pipe = 22, 可以发送0个数据包

此轮结束后, cwnd = pipe = 22, pdel = 7, pout = 5

(7) 收到6116号报文。

Pipe = 21 , pdel = 8,

Sndcnt = (25 * 0.7 * 8 + 25 -1)/25 – 5 = 164/25 – 5 = 1

Cwnd = pipe + sndcnt = 22, 可以发送1个数据包，故6317号报文被发送出去。

此轮结束后，cwnd = pipe = 22, pdel = 8, pout = 6

(8) 收到6318号报文。

Pipe = 21, pdel = 9

Sndcnt = (25 * 0.7 * 9 + 25 – 1)/25 – 6 = 1

Cwnd = pipe + sndcnt = 22, 可以发送1个数据包，故6319号报文被发送出去。

此轮结束后，cwnd = pipe = 22, pdel = 9, pout = 7

(9) 然后，收到6320号报文。和6315号报文一样，不发包，自己推导。

此轮结束后, cwnd = pipe = 21, pdel = 10, pout = 7。

(10)6321号报文。

Pipe = 20, pdel = 11,

Sndcnt = (25 * 0.7 * 11 + 25 -1)/25 – 7 = 1，发送6322号报文

Cwnd = pipe = 21, pdel = 11, pout = 8

(11)6323号报文。

Pipe = 20, pdel = 12,

Sndcnt = (25 * 0.7 * 12 + 25 -1)/25 – 8 = 1,发送6324号报文

Cwnd = pipe = 21, pdel = 12, pout = 9

(11)6325号报文

Pipe = 20, pdel = 13,

Sndcnt = ((25 * 0.7) * 13 + 25 -1)/25 – 9 = 0 (其实，是1的)

Cwnd = pipe = 20, pdel = 13, pout = 9

(12)6326号报文

Pipe = 19, pdel = 14

Sndcnt = (25*0.7 * 14 + 25 -1)/ 25 – 9 = 1

Cnwd = pipe + sndcnt = 20, 发送一个新数据包，即6327号报文

Cwnd = pipe = 20, pdel = 14, pout = 10

(13)6328号报文

Pipe = 19, pdel = 15,

Sndcnt = (25*0.7 * 15 +25-1)/25 – 10 = 1发送一个新的数据包，即6329号报文

Cwnd = pipe + sndcnt = 20

Cwnd = pipe = 20, pdel = 15, pout = 11

(14) 6330~6335号报文,共6个。

Pipe = 20 – 6 = 14, pdel = 21

此时 pipe < ssthresh, 注意，sndcnt 的计算公式变了！

Sndcnt = ssthresh – pipe = 17 – 14 = 3

Cwnd = pipe + sndcnt = 17。

故可以发送3个新数据包，这就是6336~6338的来历。

此轮结束后, cwnd = pipe = 17, pdel = 21, pout = 14

注意此步骤，因为cwnd已经收敛至ssthresh！所以，根据PRR算法，接下来，如果它还收到重复确认，那么它的行为只能是“来一个重复确认，发送一个新数据了”。

(15) 当收到6343号报文时，ack > high_seq，终于退出Recovery状态，

此时，in_flight = (5657409 + 1440 – 5637249)/1440 = 15。

Delta = cwnd – in_flight = 17 – 15 = 2,故发送两个新数据包。

接下来，我们又看到典型的拥塞避免状态了，说明它又回到了OPEN状态。

至此，我们的分析工作已经结束。从理论分析和抓包的实际情况来看，我们的分析是完全正确的。

由于Windows系统与Linux系统在快速恢复的拥塞窗口调整策略有所差异，我们可以利用这一特征来判断服务器端是Windows操作系统还是Linux操作系统。当然，我们很少能在服务器端的出接口抓包分析，为了实现从数据接收端进行远程操作系统判断，还需要做一点额外的工作[1]。

PRR-RB算法的特点是，**它保证拥塞窗口收敛于最终的慢启动阈值**。

当处在disorder状态的发送端收到足够多的重复ack，使得sacked_out的个数达到设定的范围reordering时，发送端将判定这个报文为已丢失，并由disorder状态进入recovery阶段，也就是快速重传与快速恢复阶段。在recovery状态，TCP

[1] 这一点已申请专利。

将执行快速重传算法，recovery状态的状态迁移表如下：

表 24 recovery状态迁移表

信号	下一个状态	备注
ack = snd_una	recovery	见2.5.2条
snd_una < ack < high_seq	recovery	见2.5.3条
ack ≥ high_seq	open	见2.5.4条
time_out	loss	见2.6.1条

Recovery的拥塞窗口调整策略

这时会进行一些操作，这些操作可以概括为四个步骤：保存状态信息，标记报文丢失，更新拥塞窗口和重传丢失报文。用伪代码表示如下

```
if( sacked_out ≥ reordering)
        {
            save_state_info();
            tcp_update_scoreboard ();
            tcp_cwnd_down();
            tcp_xmit_retransmit_queue();
        }
```

下面将逐一介绍这些操作步骤。

(1) 保存状态信息

当收到reordering个重复确认ack时，判定ack对应的报文信息已经丢失，并从disorder进入recovery状态。但这种将一个报文判定为丢失的方式并不准确，这个报文还是可能在网络上传输并可能在将来到达接收端，如果是这种情况的话，可能从recovery状态回退到open状态。为了保留回退在先前状态的能力，在进入recovery状态之前，先保存当前状态信息，如下表所示：

表 25 被保存的主要变量及其存储位置

被保存变量	存储位置
cwnd	prior_cwnd
snd_ssthresh	prior_ssthresh
snd_una	undo_marker
snd_next	high_seq

保存状态信息以后，将慢启动阈值snd_ssthresh调整为当前拥塞窗口cwnd的一半，snd_cwnd_cnt设置为0[1]，并将状态设置为recovery状态：

```
tp.snd_ssthresh = max(tp.cwnd >> 1,2); /* 慢启动阈值不能小于2 */
tp.snd_cwnd_cnt = 0;
tp.state = recovery;
```

(2) 标记报文丢失

保存状态以后，将发送队列的第一个数据段标记为丢失，如下图所示，这时，丢失数据段的个数lost_out由0自增为1。

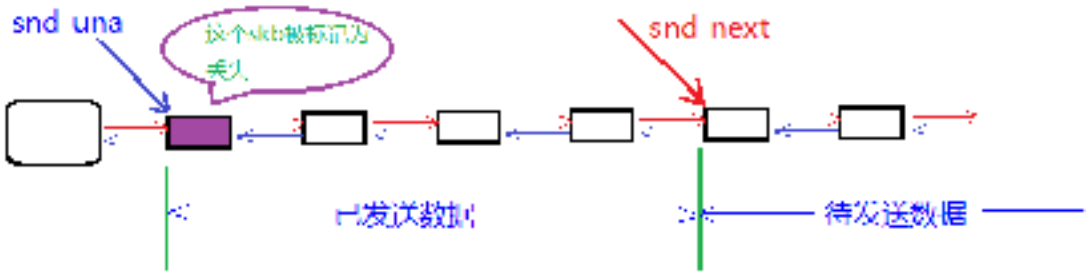


图 28 将发送队列头部数据段标记为丢失过程示意图

(3) 更新拥塞窗口

接下来，将拥塞窗口调整为当前拥塞窗口与in_flight+1之间的较小值：

```
cwnd = min(cwnd, in_flight+1)
```

这一步执行的实际上是cwnd = in_flight+1，因为在标记报文丢失之前，满足cwnd ≥ in_flight。收到此重复确认后，sacked_out和lost_out各自增1，导致in_flight自减2，执行这步操作的时候，满足cwnd > in_flight+1。

(4) 重传丢失报文

最后，将这个标记为丢失的报文重传出去。

---

[1] snd_cwnd_cnt作为一个辅助变量，用来调节处于拥塞避免和recovery状态的拥塞窗口大小，参见2.3.1.2款。

事实上，将标记为丢失的报文重传出去这一操作步骤是靠重新发送重传队列来实现的。由于拥塞窗口被调整为in_flight+1，从重传队列发送这个被视为丢失的报文之后，in_flight自增1，这时候，有cwnd = in_flight，于是不能再发送新数据了。因此，我们可以将进入快速重传的过程概括为：

当收到足够多的重复确认而进入快速重传阶段时，慢启动阈值调整为当前拥塞窗口的一半，拥塞窗口调整为当前正在传输的数据段个数加1，将未被确认的首个数据段标记为丢失并重传出去，不发送新数据。

3.5.4 在快速重传阶段继续收到重复确认

当在快速重传阶段继续收到重复确认时，Linux采用的做法是，如果cwnd > snd_ssthresh，则每收到两个重复ack，将cwnd自减1，并发送丢失的数据段。否则，cwnd不变，每收到一个重复ack就重传一次。在此期间，不发送新的数据段。

cwnd > snd_ssthresh时的cwnd变化过程及数据发送过程如下图所示：

图 29 recovery状态继续收到重复确认时sacked_out、cwnd的变化过程，及报文重传过程。这个过程一般不会持续太久。

3.5.5 快速重传阶段部分确认的处理

在快速重传状态，当收到的ack确认了新数据时，需要分多种情况来考虑，以下以内核2.6版本为准。

若ack确认了新数据，并且 ack < high_seq时，进入一种所谓的“部分确认”(partial acknowledgement)的状态。例如，当发送端发送了序号为n~n+10的数据

段，接着收到3个确认号为n的重复确认后进入recovery状态，然后重传n，接着收到确认号为n+5的ack，这说明n~n+4的数据段都被接收了。这种情况有两种可能性：(1) 重传的数据段n到达了接收端；(2) 原来认为丢失的数据段n其实并没有丢失，只是晚到了一步。这两种情况下所表现的网络拥堵状态是不同的，在第一种情况下，第n+5号报文可能已丢失的可能性要高于第二种情况下的可能性。

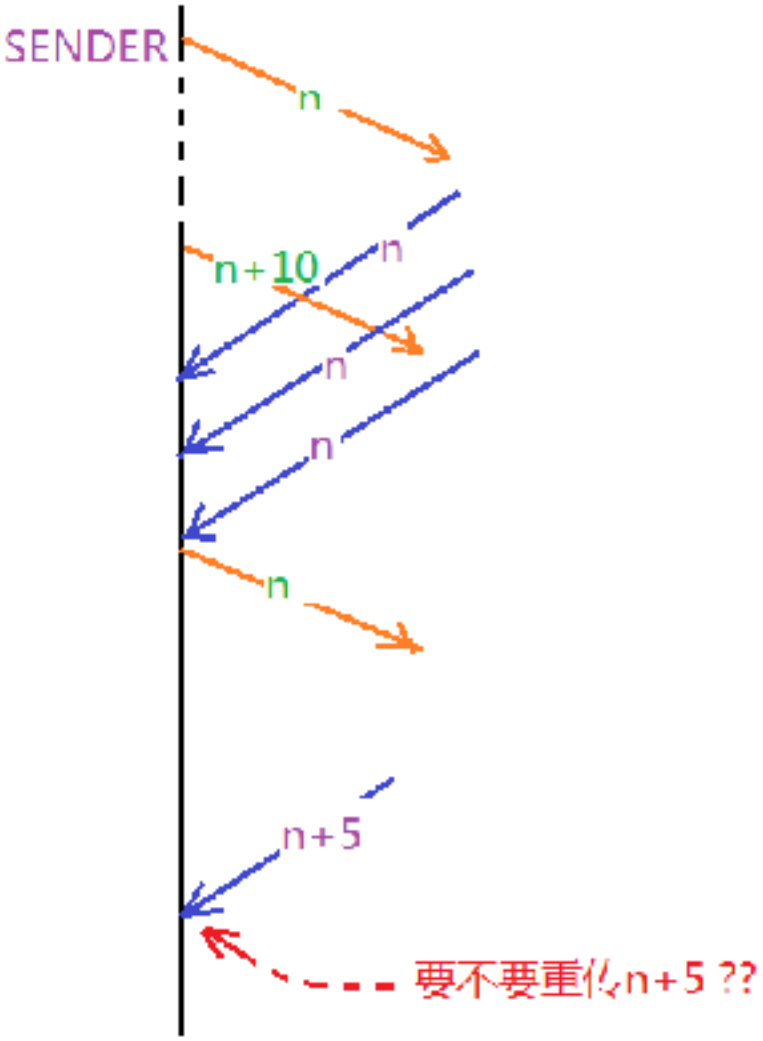


图 210 迷惑的发送端

为避免不必要的重传，并及时重传那些已丢失可能性较大的数据段，Linux采用的做法如下[1]：

先从这个ack报文的时间戳[2]获得此ack的发送时间rcv_tsecr，并与最近一次

[1] 这种做法其实是NewReno算法的一种优化，NewReno算法采用的方法是，只要在recovery状态出现部分确认，比如这里的n+5，将n+5号数据段标记为丢失并立即重传。

[2] 如果不支持时间戳，则立即采用NewReno算法。

重传的时间retrans_stamp比较(在上图中，就是比较发送端发送n与接收端回复n+5的发送时间)。

如果rcv_tsecr < retrans_stamp，说明接收端接收到数据段n的时刻在我们重传n之前，亦即数据段n并没有丢失。这时，先更新reordering，把慢启动阈值snd_ssthresh恢复到之前的阈值prior_ssthresh，然后将cwnd调整为max(cwnd, snd_ssthresh << 1)，再调整为min(cwnd, in_flight+reordering)[1]

```
if(rcv_tsecr < retrans_stamp)
{
  reordering = tcp_update_reordering();
      snd_ssthresh = prior_ssthresh;
      cwnd = max(cwnd, snd_ssthresh<<1);
      cwnd = min(cwnd, in_flight + reordering);
}
```

这种情况下不会重传已经发送的数据，而是根据in_flight ≤ cwnd的原则继续发送新数据。

如果rcv_tsecr ≥ retrans_stamp，将认为第一次发送的数据段n丢失了[2]，那么，这时候回复ack=n+5，很有可能是因为第n+5号数据段也丢失了(这种情况就是一个发送窗口内丢失多个数据段的情况)。这时候，拥塞窗口不调整，将ack对应的数据段标记为丢失并重传出去，如果拥塞窗口允许，就发送新数据，否则不发送。

3.5.5 快速重传阶段全部确认的处理

当进入快速重传时，Linux会把当时待发送的序号snd_next保存在high_seq中，作为判断是否能退出recovery，回到open状态的标志。当收到的ack满足ack ≥ high_seq时，说明进入recovery状态前引起sacked_out的报文都被接收了，将sacked_out重置为0。增大拥塞窗口至max(cwnd, snd_ssthresh<<1)，再把阈值恢复进入recovery前的状态，再对cwnd做一些修整，再把状态设置为open，recovery状态结束：

```
if(ack ≥ high_seq) {
    cwnd = max(cwnd, snd_ssthresh<<1);
    snd_ssthresh = prior_ssthresh;
    cwnd = min(cwnd, in_flight+reordering,)
```

[1] 这里，拥塞窗口的第二步调整是为了防止拥塞窗口的爆发式增长，在3.17版本中没有这一步，并且状态也直接回到open状态。

[2] 这个判断不准确，数据段n不一定丢失，但Linux就这么做的。

```
        state = open;
    }
```

3.6 重传计时器超时与超时重传阶段

当重传计时器超时时，不管先前处于什么状态，都会进入loss状态。本节主要介绍进入与退出loss状态的时机以及相关的一些处理过程。超时时间的计算与重传计时器的设置见下一章。

表 26 loss状态迁移表

信号	下一个状态	备注
ack ≤ high_seq	loss	见2.6.2条
ack > high_seq	open	见2.6.2条
time_out	loss	见2.6.1条

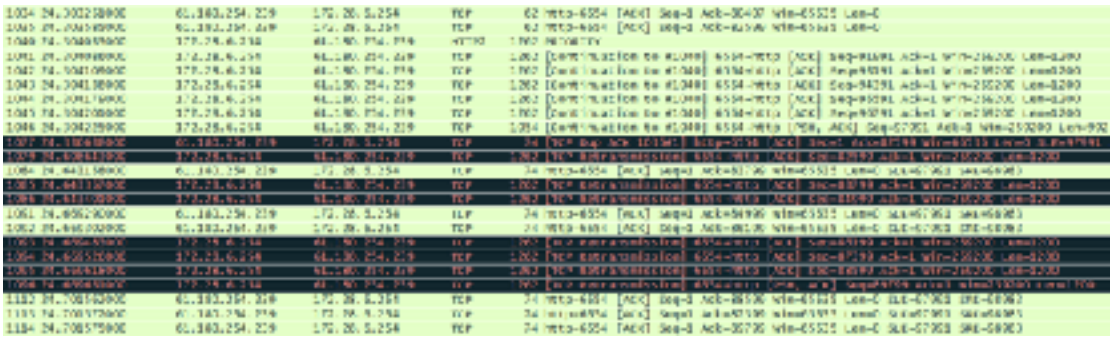
3.6.1 重传超时状态

若发送出去数据以后在较长的时间段内没有获得回复的ack，就会触发重传。

当重传定时器超时时，拥塞窗口直接减为1，并开始慢启动算法。

Loss状态的慢启动算法与OPEN状态的慢启动算法的唯一区别是，Loss状态优先去重传丢失的数据包。

计时器超时，进入loss状态。进入loss状态时，Linux先对一些统计变量和状态变量进行调整，再将状态设成loss状态，并重传第一份报文，重设超时定时器。下面将分别介绍。



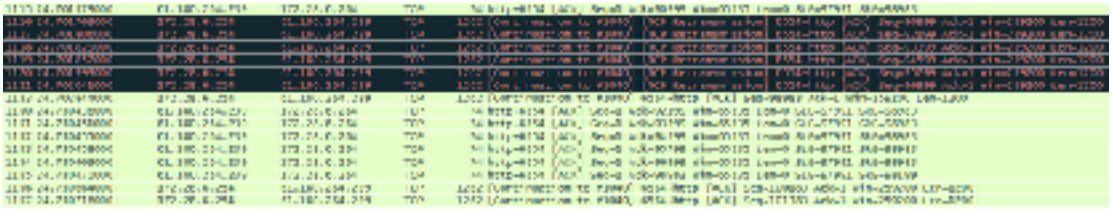


图3-18 典型的Loss状态报文图。如图所示，进入LOSS时，将重新开始慢启动算法，并且直到进入LOSS状态之前发送的所有报文都被确认后，才退出LOSS状态，进入OPEN状态。(图中收到重复确认时不发新数据，是因为暂时没有数据可发了)

3.6.1.1 进入loss状态时的变量调整情况

进入loss状态时，将拥塞窗口调整为1，阈值调整为当前拥塞窗口的一半，但不能小于2。同时，对其它变量进行调整，如表2-6所示：

表 27 进入loss状态时的数据保存情况

变量	值	补充说明
cwnd	1	
prior_ssthresh	snd_ssthresh	见(i)
snd_ssthresh	max(cwnd>>1,2)	
snd_cwnd_cnt	0	
retrans_out	0	
undo_retrans	0	
undo_marker	0/snd_una	见(ii)
sacked_out	0	
reordering	min(reordering,sysctl_tcp_reordering)	见(iii)
high_seq	snd_next	见(iv)
lost_out	count(skb in send_queue)	见(v)

下面对表中的一些变量进行补充说明：

(i) 如果进入loss状态之前是open或者disorder状态，则需要把当前的慢启动阈值保存起来，以防这是一个错误的超时重传[1]，回到open状态时需要将慢启

[1] “错误的重传”(False Retransmit)是说，当计时器超时时以为所有报文都丢失了，而实际上这些报文并没有丢失，并在超时以后收到这些报文的ack，说明这是一个错误的超时重传。

动阈值snd_ssthresh恢复到之前的阈值prior_ssthresh。

(ii) 如果进入loss状态之前，发送队列中有一些包已经被重传了，说明网络的确很堵，把撤销标记undo_marker设置为0，表示以后不会撤销了，否则。网络不一定很堵，将undo_marker设置为snd_una，表示有可能恢复原来的状态。

(iii) 在进入loss状态之前，reordering的值可能改变，故要在此恢复“出厂设置”。Reordering的值由系统变量sysctl_tcp_reordering设定。

(iv) 将snd_next保存到high_seq中，high_seq将在退出loss状态时发挥作用。

(v) 进入loss状态时，发送队列中的所有报文都被标记为丢失，lost_out设置为发送队列中的报文个数(即packets_out)。这时in_flight = 0。

3.6.1.2 变量调整后的处理

调整变量后，将状态设置成loss状态，并重传第一份认为已丢失的数据段(这一步以后，retrans_out变为1)，然后将超时定时器的超时时间增大一倍，但不能超过120秒，并重新启动重传计时器：

```
tp.state = loss;
retransmit_skb(tp);
tp.rto = min(tp.rto << 1, RTO_MAX); //RTO_MAX == 120 HZ
reset_xmit_timer();
```

有一种可能，即一直收不到ack信号，连续发生超时。当连续超时的次数超过一定限制，则释放路由缓存，这很可能是因为网络中断。

3.6.1.3 收到ACK报文的处理

当在loss状态收到一个ack时，先看看此ack是否确认了哪些数据，是的话则调用慢启动算法来更新拥塞窗口[1]，否则直接进入下一步处理：

(i) 如果从ack判断出这个ack报文在我们最后一次重传之前已经发出，并且undo_marker被置位，说明在发生重传超时之前没有重传过snd_una ~ snd_next 之间的某个数据段，并且被认为丢失的数据段实际上已经到达了对端，而我们却过早进入了超时阶段，即发生了错误的重传。这时，将发送队列中所有skb的lost标志清除。Lost_out 置0，把拥塞窗口与慢启动阈值恢复到进入loss状态之前的值，这时候不再重传，而是直接发送发送队列中的新数据。

(ii) 如果不是(i)的情况，则继续传送重传队列中剩余的数据。当重传队列中的数据传完以后，如果拥塞窗口允许，则发送发送队列中的新数据。

发送新数据之前，snd_next和high_seq是一样的，发送新数据会使得

[1]这时候调用的慢启动算法与open状态的慢启动算法完全相同，见2.3.1条。

snd_next大于high_seq，当这个新发送的数据被ack确认以后，就会退出loss状态而进入open状态。

至此，拥塞状态机的工作过程介绍完毕，Linux内核就是靠这些来调整拥塞窗口的。

4. 正在传输的数据段统计与重要参数分析

4.1 pipe控制

正在传输的数据段in_flight表示由发送端发出，但未到达接收端的所有报文总数。实际上in_flight由由四个变量控制，即：packets_out、retrans_out、sacked_out和lost_out，且in_flight = packets_out+retans_out-sacked_out-lost_out。在上一章将in_flight视为一个已知量，本章将讲解in_flight各个量的统计过程。

4.1.1 packets_out

packets_out表示已发出但未被确认的报文个数，packets_out等于snd_una 至snd_next之间的报文个数，即等于待发送的数据与未确认的数据之间的报文个数。当初始化时，packets_out为0。

当发送syn包时，packets_out=1，以后当发送数据包时，每发送一个新数据段，packets_out自增1。

当收到一个ack时，根据ack的大小，系统会去遍历写队列中的skb，如果skb的末字节序列号是否比ack小，是则说明这个报文已经被接收了，就将packets_out自减1。

4.1.2 retrans_out

当TCP连接初始化或者断开连接的时候，retans_out=0。

当拥塞状态处于recovery或者loss状态，成功重传一个报文段以后，retrans_out自增1，并将被重传的报文置被重传位[1]。

当收到的ack确认了发送队列上已发送的报文时，会看看其是否有TCPCB_SACKED_RETRANS标志，有的话则retrans_out自减1。

当进入loss状态时，会将lost_out置0。同时将所有TCPCB_SACKED_RETRANS标志清掉。

在收到ack时，从写队列中释放已被确认的数据在调整拥塞状态之前执行，

---

[1] TCP_SKB_CB(skb)->sacked |= TCPCB_SACKED_RETRANS.

亦即packets_out与retrans_out先被调整，再调整cwnd。

4.1.3 lost_out

事实上，lost_out是对丢失报文个数的估计，并不能准确反应已丢失报文的个数。比如，当收到三个重复ack时，将lost_out置1，实际上，这个报文不一定丢失。当认为一个报文丢失时，将skb置TCPCB_LOST标志，表示认为它丢失了。当从收到的确认号判断这个报文段已被接收时，在释放本skb之前，会判断其是否带有TCPCB_LOST标记，有则将lost_out自减1。

4.1.4 sacked_out及其修正机制

对于RenoTCP拥塞控制来说，sacked_out是in_flight四个变量里面最难以统计的。根据TCP规范，当一个报文段到达接收端，而这个报文段是乱序的，那么，接收端会立即回复ack以请求所期待得到的报文。即一般意义上来说，当收到一个duplicate ack时，sacked_out自增1，但无法确认是哪个报文离开了网络。比如，如果某些接收端为了“加速”发送端重发乱序数据包的流程，可能在收到一个乱序ack后连续回复多个相同的ack，这时如果还按照每收到一个重复ack就将sacked_out+1的做法就不能反应通过重复确认离开网络的报文数量了。即使这种情况不发生，sacked_out的统计也存在天然的缺陷：当收到的ack不再是duplicate ack时，sacked_out该如何变化？

为此，Linux引入sacked_out修正机制，在sacked_out自增时或者在收到部分确认，需要减少sacked_out时，都对sacked_out进行一次修正：

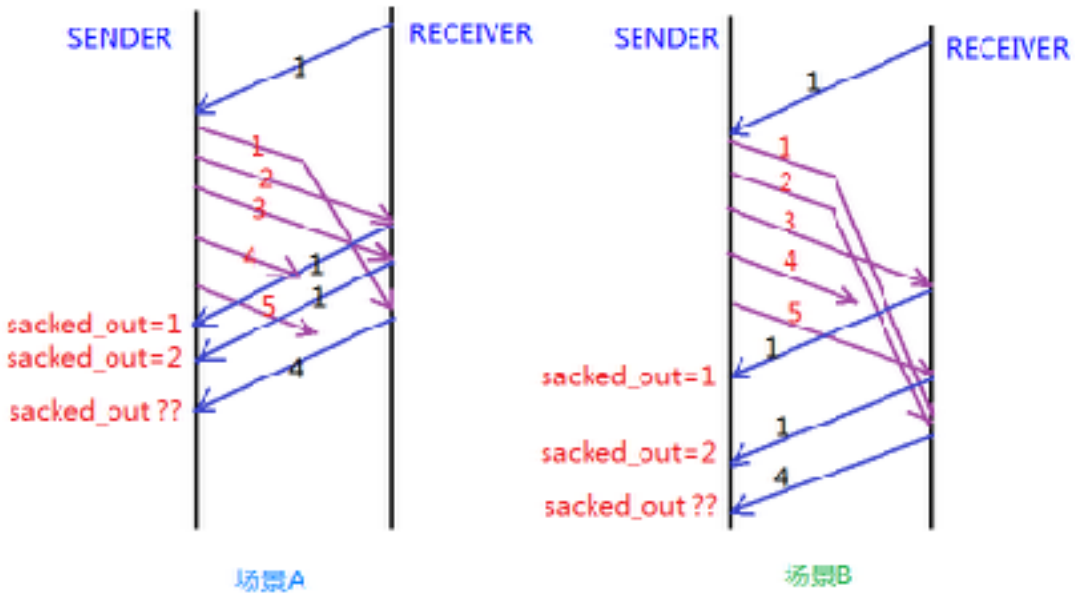


图 31 对于发送端来说，场景A和场景B情况下收到的信息是相同的，只靠重复ack无法辨别当前发生了那种情况，给sacked_out统计带来了困难。

(1) 当收到重复确认时，sacked_out自增1，然后，测试packets_out ≥ sacked_out + lost_out是否满足，不满足则减小sacked_out的值使之满足。伪代码如下：

```
holes = max(lost_out, 1);
if (sacked_out + holes > packets_out) {
        sacked_out = packets_out – holes;
}
```

这里设置holes = max(lost_out,1)是考虑到sacked_out + 1 > packets_out是不合法的：假设发送了1，2，…，k等数据段，packets_out=k，则收到对1的重复确认只能由2，3，…，k等的到达引起的，即saked_out最多为k-1，正常情况下sacked_out加1不可能超过packets_out。

(2) 当收到部分确认时，也需要对sacked_out进行调整。这时sacked_out的调整原则是：先认为sacked_out是由被确认了的数据引起的，如果不够用，再认为是未被确认的数据引起的。

为了更好地理解这句话，举个例子：本端发送了1~12号数据段，然后，连续多次收到确认号为5的ack，使得sacked_out = n，接着收到确认号为9的ack，是个部分确认，说明5~8这些数据段都到达对端了。调整sacked_out时，先认为sacked_out是由6~8这些段引起的，6~8最多能引起3个重复确认，如果够用，即n≤3，则sacked_out=0；不够用，即n>3，说明9以后的数据段也引起重复确认，sacked_out = n-3。伪代码表示为：

```
if (acked – 1≥sacked_out) {
    sacked_out = 0;
} else {
    sacked_out = acked – 1;
}
```

acked表示被部分确认ack确认了的数据段个数。

将reordering = min (packets_out, TCP_MAX_REORDERING)，TCP_MAX_REORDERING被定义为127。

即以场景A为准，这是一种保守的做法，将in_flight保持在一个大的值。

4.2 拥塞误判及其恢复策略

4.2.1 拥塞误判介绍

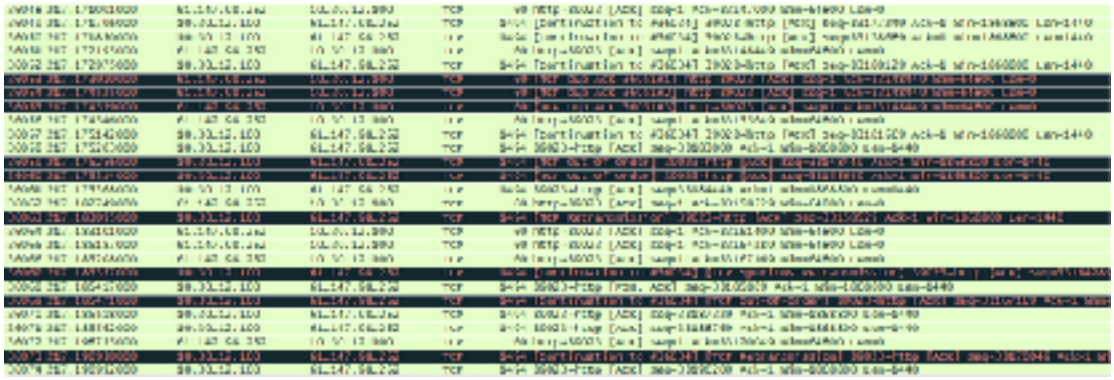
当收到一定个数的重复确认或者发生重传定时器超时时，经常被拥塞控制

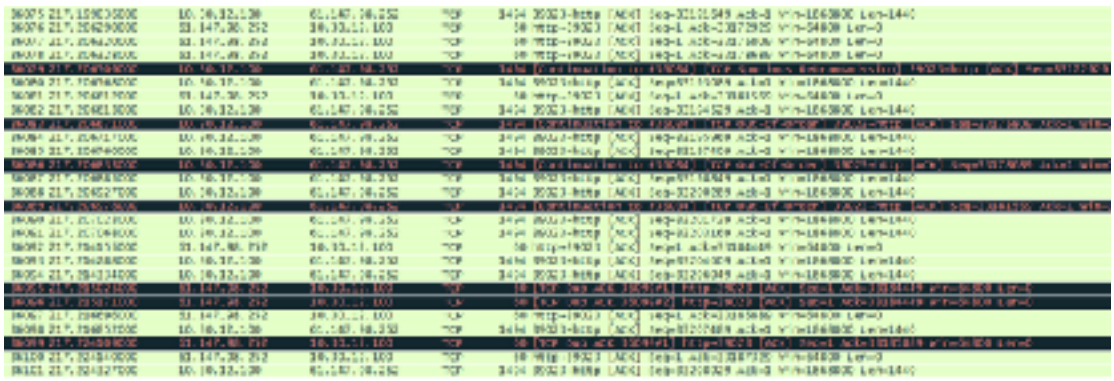
流程认为发生拥塞了，这导致拥塞窗口的减小，瞬时吞吐量也随之降低。然而，这种判断本身就是有问题的。比如，当收到3个重复确认报文时，就一定说明它丢包了吗？这明显不是的。而重传定时器超时时，也不一定说明是丢包了，所以必须具备这种拥塞误判的能力。

拥塞误判指的是，在错误的条件下缩小拥塞窗口和慢启动阈值，使得吞吐率降低。在拥塞控制中，有两种地方会有拥塞误判：进入Recovery时和Loss状态时。

拥塞误判有两类：恢复误判和超时误判。前者是指收到多余的重复确认时发生，后者指的是重传定时器早点发生超时了。

收到一定个数的重复确认，进入Recovery状态时，会引起拥塞窗口和慢启动阈值的减小。而网上，乱序发生的可能性是很高的，如果一旦发生乱序而不丢包，则拥塞窗口和慢启动阈值的减小是不必要的。但如果这种误判发生了，就很可能导致性能不必要的下降。





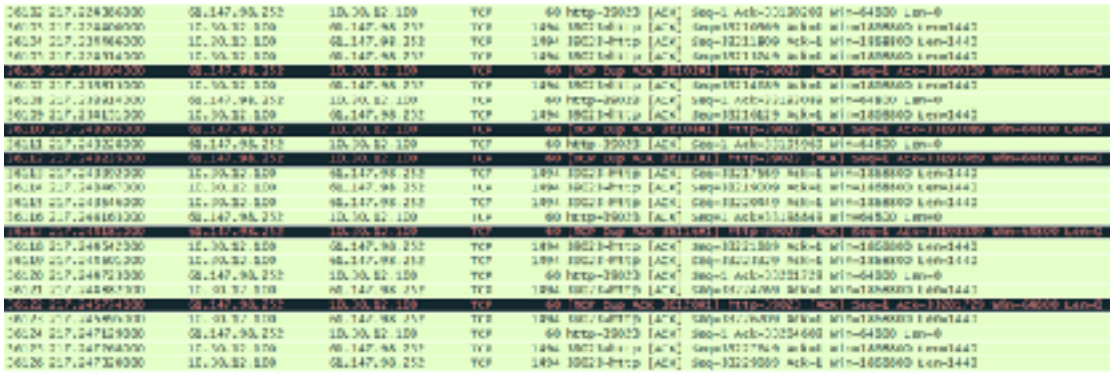


图3-19 典型的Linux快速恢复误判及其后遗症。它除了引起不必要的重传，导致拥塞窗口和慢启动阈值减小外，还拖着一条长长的“重复确认”尾巴。若Linux操作系统作为服务器端，并且不支持时间戳时，这种误判发生的可能性还是比较高的。“重复确认”尾巴是Linux操作系统的特征

图X演示的就是一个典型的快速恢复误判效果：36053~36055是3个连续的重复确认报文，按照拥塞控制算法，此时，将引起快速重传并进入Recovery阶段，慢启动阈值变为拥塞窗口的0.7倍。但是，接收端又立即传来的36056号报文立即确认了该“丢失”的报文。很明显，这是序列号为33148449的报文只是延迟到达接收端了，并没有丢失，而数据发送端此时已经进入Recovery状态，故36059号报文是一个错误的“快速重传”，这导致后面到达的正常ACK被当作部分确认，引发连续多次恢复重传操作。而这些不必要的重传会在退出Recovery状态后，引发一系列的，不必要的重复确认。在图中，从36090号报文开始的重复确认，就是由于Recovery状态的不必要重传引起的。这种不必要的重传一旦发生，就会在回到open状态后，留下一条“重复确认”尾巴。这种“重复确认”尾巴除了加重数据收发方的处理负担以外，还有可能重叠到一起，形成连续的重复确认，导致再次引发误判，再次进入Recovery状态，性能再次降低，影响非常恶劣。

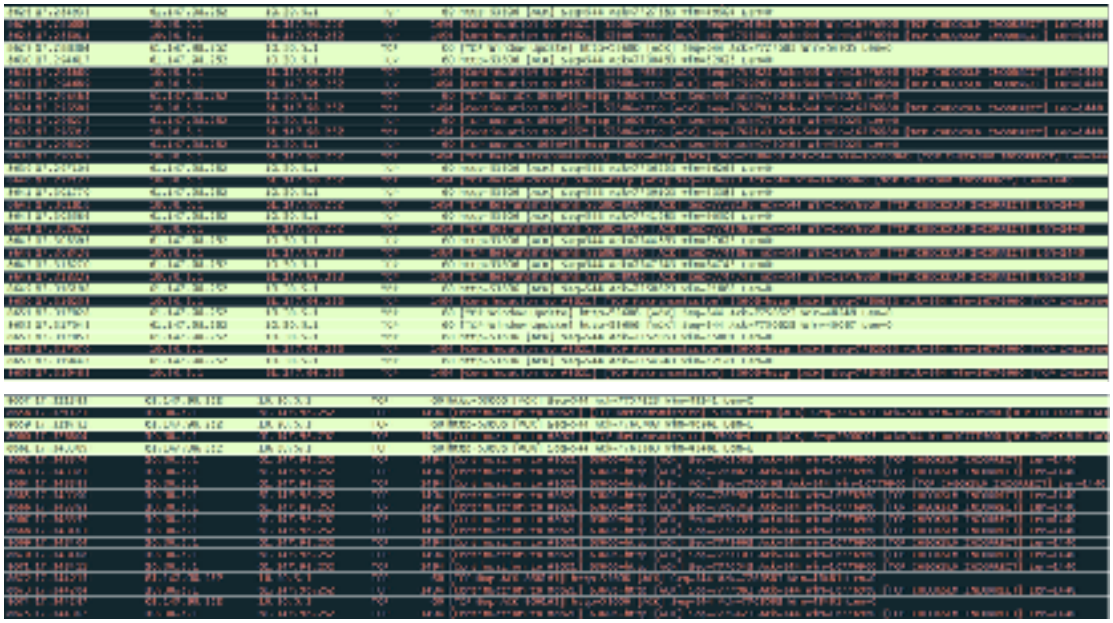


图3-20 典型的Windows操作系统快速恢复误判。与Linux系统不同的是，它收到老的数据包时不回复重复确认，于是就没有“重复确认”尾巴。从拥塞控制的角度考虑，为了提高TCP协议的性能，收到老的数据包时，不回复重复确认比较好。

图X是Windows操作系统的Recovery阶段误判。虽然8638号报文在8639号报文到达之前发送出去，但两者的时间差只有0.8 ms ，而根据前面的报文推导出rtt = 35ms左右，两者差距这么大，说明在这个重传的报文到达接收端之前，这个ack就已经发送出来了，这是明显的乱序情况，而不是丢包。误判导致在接下来的时间段内，发送端把正常的ACK报文当作部分确认处理并引发不必要的重传。由于Windows系统在收到老的数据包时，不回复ACK，于是，我们就看不到在Linux系统下才特有的“重复确认”尾巴了。

超时误判
与恢复误判相对应的是超时误判，后者在重传定时器超时有可能发生，由于网络状态的不稳定性，RTT会经常发生变化，超时是有可能发生的。但相对于恢复误判，超时误判发生的可能性比较小，因为重传定时器的超时时间RTO是比RTT大的。所以至今没有抓获超时误判报文。

4.2.1 误判恢复策略

误判恢复策略的核心是，通过收包时间来判断：

如果在重传的报文到达之前，ack就确认了这个数据包，则说明是个误判。
误判发生时，就回到以前的状态。

Linux的恢复策略是，这种恢复策略过于保守，并且在不支持时间戳时不够用。

所以，自主TCP的快转分支不支持SACK，并且报文中不包含时间戳，Linux系统是不具备误判恢复策略。

为此，在不支持时间戳时，我们可以写自己的误判恢复策略：

当重传时，记录报文的重传时间，当收到确认数据的ACK时，比较两者的时间差，通过时间差来判断该报文的“漏洞”是否由重传填上，如果不是，说明这是个误判。并将拥塞窗口和慢启动阈值恢复到之前的状态。

4.2 重定序临界值reordering[1]

个别报文的丢失与报文不按序到达接收端都可能引起重复确认，而在收到重复确认时，过早地将重复确认判定为报文丢失并开始快速重传可能导致吞吐率下降。为此，Linux系统以reordering作为判断是否开始快速重传的临界值。reordering的初始值由可以人为设定，并在进入loss状态时回到其初始值[2]。而在其它流程中，可能引起reordering的改变。

上文已经提到，误判经常是进入Recovery状态时发生，而是否发生与

---

[1] reordering这个量没有统一的翻译，有“重定序个数”、“重定序长度”等叫法，在此译作重定序临界值。

[2] 关于reordering理论请参考论文RR-TCP: A Reordering-Robust TCP with DSACK.

reordering的关系非常大，所以，Linux系统所采用的算法并不是按照每收到3个重复确认就开始快速重传，而是看情况而变的。当探测到恢复误判时，将调整reordering的计数，当超时发生时，reordering的计数回归正常。

但目前对这个做法有有疑问，如果reordering因偶然发生的reordering导致变大，可能拖延快速重传的发生时间，而这个是有可能导致性能下降的，因为要是真的发生了丢包，则reordering的增大会延迟快速重传的发生时间，从而拖延从快速恢复状态恢复的时间。所以，最好的做法，是较早地进入快速重传，所以reordering应该保持为3，我们宁肯冒着误判的危险早点进入快速重传，并根据ACK报文的接收时间早点退出快速恢复。最好的做法是 较小的reordering 加上激进的恢复误判策略。

从抓包情况来看，windows系统和Linux系统的报文来看，没有SACK和时间戳时，它们都不执行误判恢复算法，大概是因为它们主要运行于具有SACK和时间戳的环境中，所以，在设计并实现自己的TCP协议，并且暂时不支持SACK和时间戳时，是需要自行设计一套误判恢复策略的，以提高性能。

4.3 拥塞控制算法概述

从TCP协议产生至今，TCP拥塞控制的算法得到了很大的发展。但它们的主要思想是不变的，即慢启动阶段涨得很快，拥塞避免阶段探测性增长，收到重复确认时先观望，收到够多的重复确认时开始快速重传，在快速恢复阶段争取早点把可能丢失的报文重传出去，并适当发送新数据包以维持协议的正常运转。当退出快速重传时，拥塞窗口和慢启动阈值按预期的值变小。并回到拥塞避免阶段。同时挂一个重传定时器来处理长时间未收到报文的情况。当重传定时器超时时，批量重传，重新开始慢启动啦。拥塞控制的总体流程都是这样的。

下面将介绍各个算法的特点。以对拥塞控制有一个全面的了解。

各个拥塞控制算法虽然种类繁多，但各个算法的核心只有两点，其余是一样的。

除了个别算法会改动慢启动阶段的增长方法之外，其它都不该动的，即，大家基本都采用翻倍增长模式。实际上，由于这种乘性增的增长速度已经够快了，所以大家都默认了。

那么，拥塞控制算法的主要改动点在于拥塞避免阶段和快速恢复状态回到拥塞避免阶段的拥塞窗口和慢启动阈值。而在收到3个重复确认后和在快速恢复和LOSS状态下的操作是差不多的。

虽然拥塞控制算法只改了这么两点，但是，算法的差异对TCP协议的性能

还是有不少的影响的。在一次TCP连接生命周期中，只要不断网，一般情况下出现重传超时出现的可能性非常的低。而只要不出现重传定时器超时，在TCP连接生命周期中，慢启动阶段发挥作用只有在初步建立连接时发挥作用，故慢启动阶段的拥塞窗口调整策略对TCP性能的影响微乎其微。这也是为什么各种算法都不热衷于改动慢启动算法的原因。

总体来说，在一次TCP连接过程中，其大部分时间是处在拥塞避免阶段的，偶尔处在Recovery恢复状态，即“拥塞避免—Recovery—拥塞避免—Recovery”的状态循环过程中，故对于一个需要传输大量数据的TCP连接来说，其性能主要来自于3个阶段：

(1) 拥塞避免阶段的初始窗口大小，即退出快速恢复状态时的慢启动阈值

(2) 拥塞避免阶段的拥塞窗口调整策略

(3) 快速恢复状态的恢复快慢及发包行为

可见，虽然拥塞控制算法主要改了(1)和(2)，但这正是对TCP拥塞控制的性能最大的两处，故一个拥塞控制算法的好坏对拥塞控制性能的影响还是比较大的。

除此之外，还有第三点，第(3)点的差异不大，主要是支持选择性确认和不支持选择性确认的区别，但这个地方进行优化的可能性比较少，这个会在下文进行介绍。

可见，影响拥塞控制性能的主要有两点：退出Recovery阶段时的慢启动阈值和拥塞避免阶段的拥塞窗口调整策略。这也是各个拥塞控制算法在这两点死磕的原因。对于这一点，传统的Reno算法，即每收到等于当前拥塞窗口的ACK报文就把当前拥塞窗口自增1，这就显得“Too young too simple”了。为此，各种算法也是各显神通了，对此，各个算法从不同的角度考虑问题。得出的算法也各有差异。限于篇幅，在此就不再介绍各个算法是怎么玩的了，而实际上，只要理解了拥塞控制算法的整体流程，再去分析各个算法就显得很轻松了。只要做到真正从灵魂上了解拥塞控制，再去设计拥塞控制算法也是可行的。

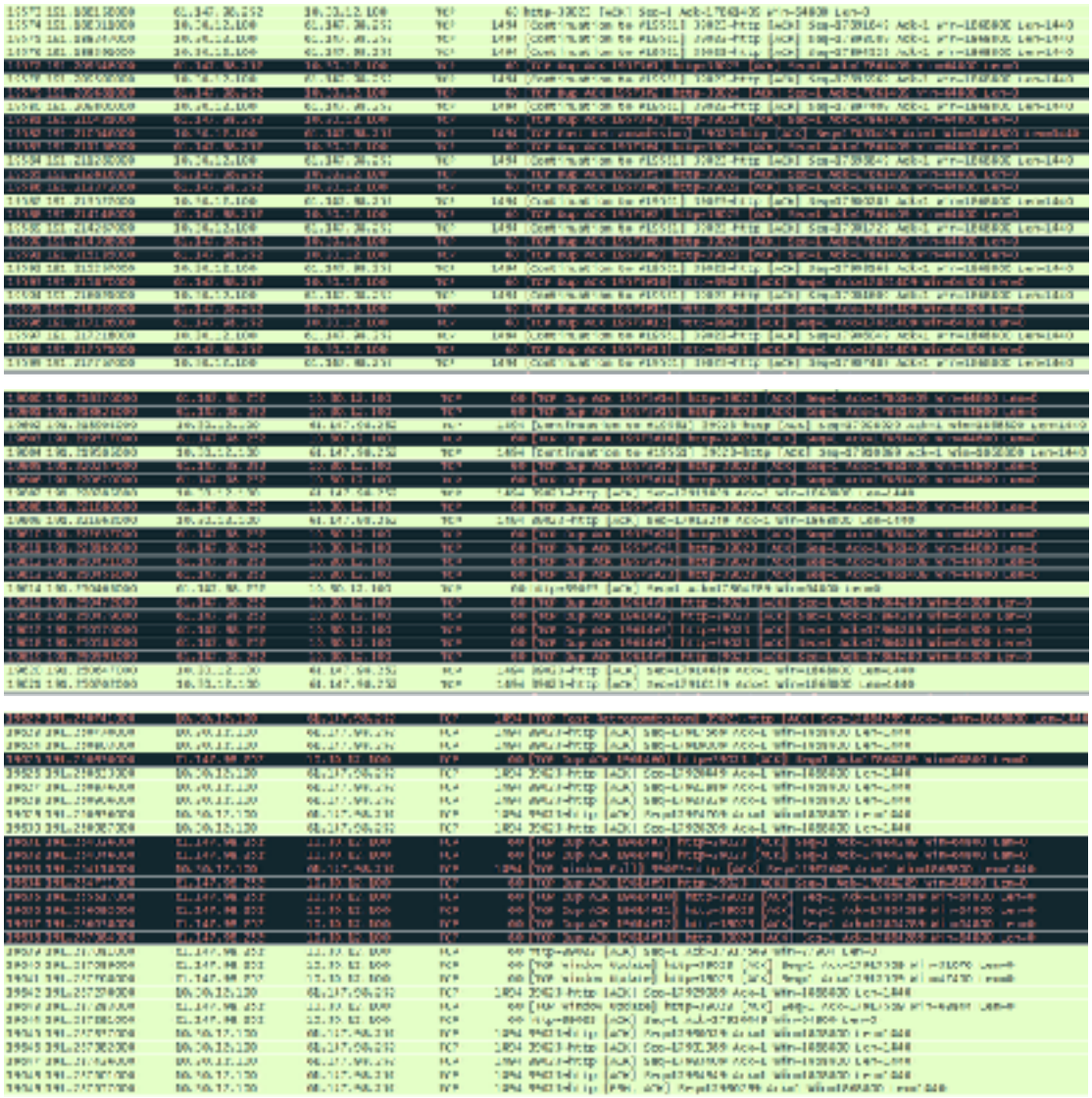
各个拥塞控制算法各有优缺点，一个优秀的拥塞控制算法应该博采众长，自成一套，在不同的情况下做不同的处理，在此就不再往下分析了。

4.3 选择确认SACK

如上文所说，除了拥塞控制算法的选择之外，另外一个影响拥塞控制算法性能的是快速恢复阶段的处理方式，而与此有关的是TCP报文中的选择确认(Selective Acknowledgement)字段。

选择确认是指携带在重复确认报文中的一个字段，表示当前乱序获得的数据区间。相比于不带选择确认的重复确认报文，它能指出当前有哪些字段乱序到达接收端了，即，携带了更多的信息，而这些信息对于数据发送端还是很有帮助的，通过这些信息，它能更清楚地了解到，可能有哪些报文丢失了。

如前面所说的，当不支持选择确认时，在recovery阶段，有用的信息只有ACK确认号，我们不能知道除此之外，其它报文的到达情况，所以，每次重传时只能重传一个丢失的报文，这就引起性能的降低，尤其是在高速网络中并且偶尔存在多包丢失的情况时。



图表 22 典型的多包丢失情况——丢了两个连续的报文。图中报文的RTT是50ms左右，因为没有SACK，故晚了大约一个RTT才退出Recovery状态。它必须等待第一个丢失的报文被确认后才重传第二个报文。如果有SACK，它的退出时间将会早一点

4.3.1 SACK对性能的影响分析

上图就是一个典型的多包丢失情况，如果支持SACK的话，发送端会早点探测到多包丢失，并早点退出拥塞避免状态的。

然而，SACK虽然携带更多的数据量，但是，为了处理SACK信息是有一定开销的。如果是单包丢失的情况，SACK只会加重处理的负担，说白了，SACK只有利于多包丢失的情况。

那么，如何在SACK支持与否之间做出取舍呢？让我们推导一下SACK对发包数量的影响。

SACK的唯一作用，是多包丢失的情况下，可以快速退出快速恢复阶段。也就是说，如果丢包率很低，或者丢包时常常只丢一个包，那么SACK是多余的。

4.3 Cubic拥塞控制算法

在目前的项目中，采用的拥塞控制算法是Cubic算法，并且Linux内核也默认采用Cubic拥塞控制算法，Cubic算法具有良好的TCP友好性，并且性能不错，下面将对Cubic算法进行介绍。

与多数拥塞控制算法一样，

良好的TCP拥塞控制算法展望

良好的TCP拥塞控制算法，具有良好的TCP友好性。

(1) 良好的拥塞避免算法及慢启动阈值调整策略

(2) 良好的恢复算法(有必要吗)

(3) 误判及恢复策略(误判发生的可能性很高)

附录

PRR算法的相关证明

Cubic 算法系数的相关说明

第二版展望

第一版有部分内容没有完善，第二版将补充上去。除此之外，第二版将着重介绍TCP协议工程实现中的一些“关键”技术细节，**包括**

(i) 业界主流拥塞控制算法介绍

(ii) TCP有限状态机

(iii)传输层与socket层的交互 (socket编程接口与epoll等模型)