这篇文章是下篇，所以如果你对TCP不熟悉的话，还请你先看看上篇《[TCP的那些事儿（上）](http://coolshell.cn/articles/11564.html)》 上篇中，我们介绍了TCP的协议头、状态机、数据重传中的东西。但是TCP要解决一个很大的事，那就是要在一个网络根据不同的情况来动态调整自己的发包的速度，小则让自己的连接更稳定，大则让整个网络更稳定。在你阅读下篇之前，你需要做好准备，本篇文章有好些算法和策略，可能会引发你的各种思考，让你的大脑分配很多内存和计算资源，所以，不适合在厕所中阅读。

**TCP的RTT算法**

从前面的TCP重传机制我们知道Timeout的设置对于重传非常重要。

* 设长了，重发就慢，丢了老半天才重发，没有效率，性能差；
* 设短了，会导致可能并没有丢就重发。于是重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

而且，这个超时时间在不同的网络的情况下，根本没有办法设置一个死的值。只能动态地设置。 为了动态地设置，TCP引入了RTT——Round Trip Time，也就是一个数据包从发出去到回来的时间。这样发送端就大约知道需要多少的时间，从而可以方便地设置Timeout——RTO（Retransmission TimeOut），以让我们的重传机制更高效。 听起来似乎很简单，好像就是在发送端发包时记下t0，然后接收端再把这个ack回来时再记一个t1，于是RTT = t1 – t0。没那么简单，这只是一个采样，不能代表普遍情况。

**经典算法**

[RFC793](http://tools.ietf.org/html/rfc793) 中定义的经典算法是这样的：

1）首先，先采样RTT，记下最近好几次的RTT值。

2）然后做平滑计算SRTT（ Smoothed RTT）。公式为：（其中的 α 取值在0.8 到 0.9之间，这个算法英文叫Exponential weighted moving average，中文叫：加权移动平均）

**SRTT = ( α \* SRTT ) + ((1- α) \* RTT)**

3）开始计算RTO。公式如下：

**RTO = min [ UBOUND,  max [ LBOUND,   (β \* SRTT) ]  ]**

其中：

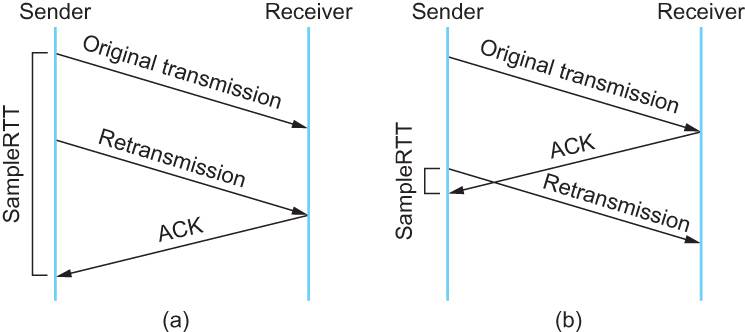
* UBOUND是最大的timeout时间，上限值
* LBOUND是最小的timeout时间，下限值
* β 值一般在1.3到2.0之间。

**Karn / Partridge 算法**

但是上面的这个算法在重传的时候会出有一个终极问题——你是用第一次发数据的时间和ack回来的时间做RTT样本值，还是用重传的时间和ACK回来的时间做RTT样本值？

这个问题无论你选那头都是按下葫芦起了瓢。 如下图所示：

* 情况（a）是ack没回来，所以重传。如果你计算第一次发送和ACK的时间，那么，明显算大了。
* 情况（b）是ack回来慢了，但是导致了重传，但刚重传不一会儿，之前ACK就回来了。如果你是算重传的时间和ACK回来的时间的差，就会算短了。



所以1987年的时候，搞了一个叫[Karn / Partridge Algorithm](http://en.wikipedia.org/wiki/Karn's_Algorithm)，这个算法的最大特点是——**忽略重传，不把重传的RTT做采样**（你看，你不需要去解决不存在的问题）。

但是，这样一来，又会引发一个大BUG——**如果在某一时间，网络闪动，突然变慢了，产生了比较大的延时，这个延时导致要重转所有的包（因为之前的RTO很小），于是，因为重转的不算，所以，RTO就不会被更新，这是一个灾难**。 于是Karn算法用了一个取巧的方式——只要一发生重传，就对现有的RTO值翻倍（这就是所谓的 Exponential backoff），很明显，这种死规矩对于一个需要估计比较准确的RTT也不靠谱。

**Jacobson / Karels 算法**

前面两种算法用的都是“加权移动平均”，这种方法最大的毛病就是如果RTT有一个大的波动的话，很难被发现，因为被平滑掉了。所以，1988年，又有人推出来了一个新的算法，这个算法叫Jacobson / Karels Algorithm（参看[RFC6289](http://tools.ietf.org/html/rfc6298)）。这个算法引入了最新的RTT的采样和平滑过的SRTT的差距做因子来计算。 公式如下：（其中的DevRTT是Deviation RTT的意思）

**SRTT = SRTT + α (RTT – SRTT)**—— 计算平滑RTT

**DevRTT = (1-β)\*DevRTT + β\*(|RTT-SRTT|)**——计算平滑RTT和真实的差距（加权移动平均）

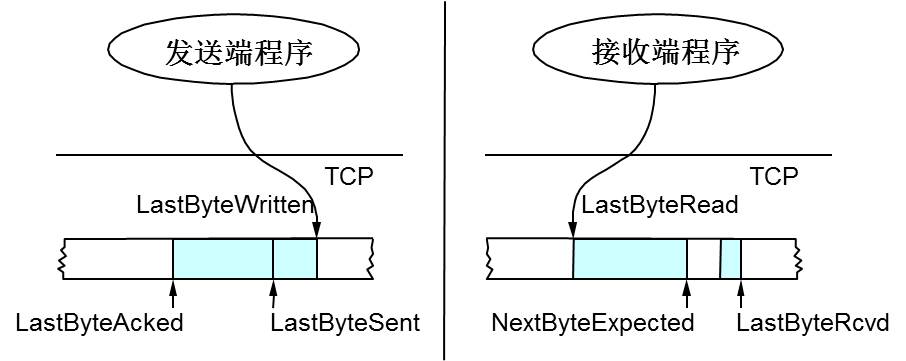
**RTO= µ \* SRTT + ∂ \*DevRTT**—— 神一样的公式

（其中：在Linux下，α = 0.125，β = 0.25， μ = 1，∂ = 4 ——这就是算法中的“调得一手好参数”，nobody knows why, it just works…） 最后的这个算法在被用在今天的TCP协议中（Linux的源代码在：[tcp\_rtt\_estimator](http://lxr.free-electrons.com/source/net/ipv4/tcp_input.c?v=2.6.32#L609)）。

**TCP滑动窗口**

需要说明一下，如果你不了解TCP的滑动窗口这个事，你等于不了解TCP协议。我们都知道，**TCP必需要解决的可靠传输以及包乱序（reordering）的问题**，所以，TCP必需要知道网络实际的数据处理带宽或是数据处理速度，这样才不会引起网络拥塞，导致丢包。

所以，TCP引入了一些技术和设计来做网络流控，Sliding Window是其中一个技术。 前面我们说过，**TCP头里有一个字段叫Window，又叫Advertised-Window，这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据**。**于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来**。 为了说明滑动窗口，我们需要先看一下TCP缓冲区的一些数据结构：



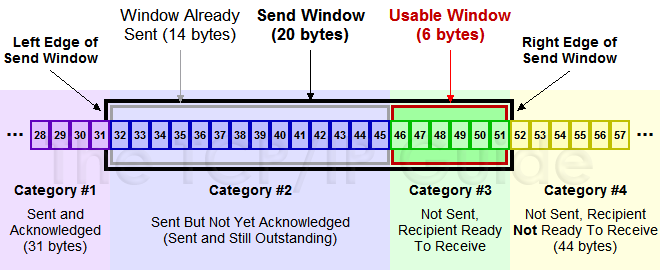
上图中，我们可以看到：

* 接收端LastByteRead指向了TCP缓冲区中读到的位置，NextByteExpected指向的地方是收到的连续包的最后一个位置，LastByteRcved指向的是收到的包的最后一个位置，我们可以看到中间有些数据还没有到达，所以有数据空白区。
* 发送端的LastByteAcked指向了被接收端Ack过的位置（表示成功发送确认），LastByteSent表示发出去了，但还没有收到成功确认的Ack，LastByteWritten指向的是上层应用正在写的地方。

于是：

* 接收端在给发送端回ACK中会汇报自己的AdvertisedWindow = MaxRcvBuffer – LastByteRcvd – 1;
* 而发送方会根据这个窗口来控制发送数据的大小，以保证接收方可以处理。

下面我们来看一下发送方的滑动窗口示意图：

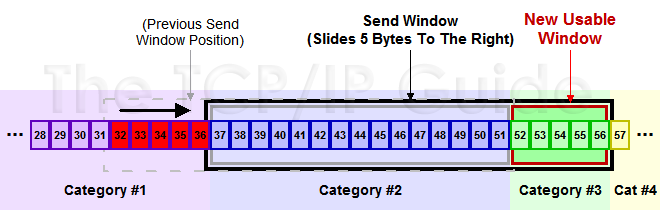


（[图片来源](http://www.tcpipguide.com/free/t_TCPSlidingWindowAcknowledgmentSystemForDataTranspo-6.htm)）

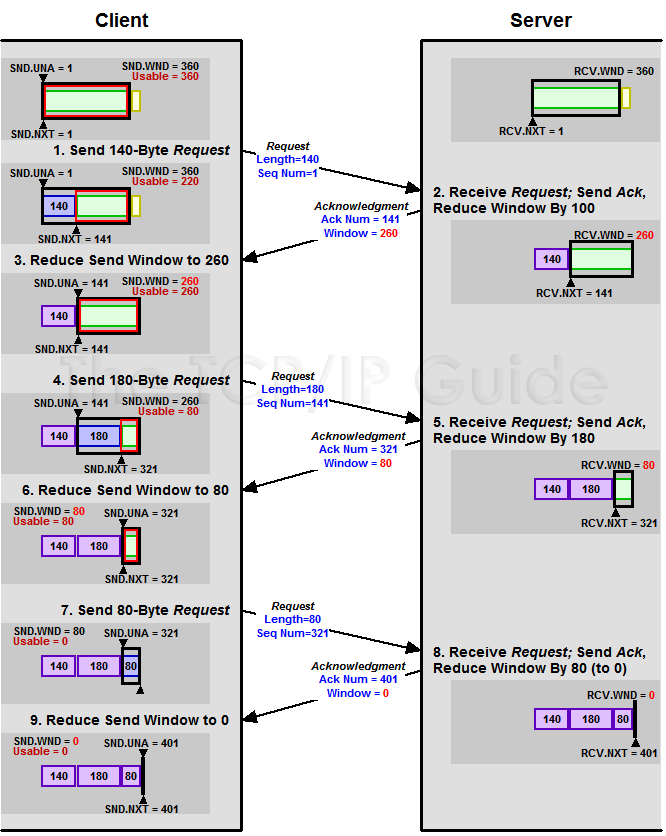
上图中分成了四个部分，分别是：（其中那个黑模型就是滑动窗口）

* #1已收到ack确认的数据。
* #2发还没收到ack的。
* #3在窗口中还没有发出的（接收方还有空间）。
* #4窗口以外的数据（接收方没空间）

下面是个滑动后的示意图（收到36的ack，并发出了46-51的字节）：



下面我们来看一个接受端控制发送端的图示：



（[图片来源](http://www.tcpipguide.com/free/t_TCPWindowSizeAdjustmentandFlowControl-2.htm)）

**Zero Window**

上图，我们可以看到一个处理缓慢的Server（接收端）是怎么把Client（发送端）的TCP Sliding Window给降成0的。此时，你一定会问，如果Window变成0了，TCP会怎么样？是不是发送端就不发数据了？是的，发送端就不发数据了，你可以想像成“Window Closed”，那你一定还会问，如果发送端不发数据了，接收方一会儿Window size 可用了，怎么通知发送端呢？

解决这个问题，TCP使用了Zero Window Probe技术，缩写为ZWP，也就是说，发送端在窗口变成0后，会发ZWP的包给接收方，让接收方来ack他的Window尺寸，一般这个值会设置成3次，第次大约30-60秒（不同的实现可能会不一样）。如果3次过后还是0的话，有的TCP实现就会发RST把链接断了。

**注意**：只要有等待的地方都可能出现DDoS攻击，Zero Window也不例外，一些攻击者会在和HTTP建好链发完GET请求后，就把Window设置为0，然后服务端就只能等待进行ZWP，于是攻击者会并发大量的这样的请求，把服务器端的资源耗尽。（关于这方面的攻击，大家可以移步看一下[Wikipedia的SockStress词条](http://en.wikipedia.org/wiki/Sockstress)）

另外，Wireshark中，你可以使用tcp.analysis.zero\_window来过滤包，然后使用右键菜单里的follow TCP stream，你可以看到ZeroWindowProbe及ZeroWindowProbeAck的包。

**Silly Window Syndrome**

Silly Window Syndrome翻译成中文就是“糊涂窗口综合症”。正如你上面看到的一样，如果我们的接收方太忙了，来不及取走Receive Windows里的数据，那么，就会导致发送方越来越小。到最后，如果接收方腾出几个字节并告诉发送方现在有几个字节的window，而我们的发送方会义无反顾地发送这几个字节。

要知道，我们的TCP+IP头有40个字节，为了几个字节，要达上这么大的开销，这太不经济了。

另外，你需要知道网络上有个MTU，对于以太网来说，MTU是1500字节，除去TCP+IP头的40个字节，真正的数据传输可以有1460，这就是所谓的MSS（Max Segment Size）注意，TCP的RFC定义这个MSS的默认值是536，这是因为 [RFC 791](http://tools.ietf.org/html/rfc791)里说了任何一个IP设备都得最少接收576尺寸的大小（实际上来说576是拨号的网络的MTU，而576减去IP头的20个字节就是536）。

**如果你的网络包可以塞满MTU，那么你可以用满整个带宽，如果不能，那么你就会浪费带宽**。（大于MTU的包有两种结局，一种是直接被丢了，另一种是会被重新分块打包发送） 你可以想像成一个MTU就相当于一个飞机的最多可以装的人，如果这飞机里满载的话，带宽最高，如果一个飞机只运一个人的话，无疑成本增加了，也而相当二。

所以，**Silly Windows Syndrome这个现像就像是你本来可以坐200人的飞机里只做了一两个人**。 要解决这个问题也不难，就是避免对小的window size做出响应，直到有足够大的window size再响应，这个思路可以同时实现在sender和receiver两端。

* 如果这个问题是由Receiver端引起的，那么就会使用 David D Clark’s 方案。在receiver端，如果收到的数据导致window size小于某个值，可以直接ack(0)回sender，这样就把window给关闭了，也阻止了sender再发数据过来，等到receiver端处理了一些数据后windows size 大于等于了MSS，或者，receiver buffer有一半为空，就可以把window打开让send 发送数据过来。
* 如果这个问题是由Sender端引起的，那么就会使用著名的 [Nagle’s algorithm](http://en.wikipedia.org/wiki/Nagle%27s_algorithm)。这个算法的思路也是延时处理，他有两个主要的条件（更多的条件可以看一下[tcp\_nagle\_check](http://lxr.free-electrons.com/source/net/ipv4/tcp_output.c#L1421)函数）：1）要等到 Window Size>=MSS 或是 Data Size >=MSS，2）等待时间或是超时200ms，这两个条件有一个满足，他才会发数据，否则就是在攒数据。

另外，Nagle算法默认是打开的，所以，对于一些需要小包场景的程序——**比如像telnet或ssh这样的交互性比较强的程序，你需要关闭这个算法**。你可以在Socket设置TCP\_NODELAY选项来关闭这个算法（关闭Nagle算法没有全局参数，需要根据每个应用自己的特点来关闭）

setsockopt(sock\_fd, IPPROTO\_TCP, TCP\_NODELAY, (char \*)&value,sizeof(int));

另外，网上有些文章说TCP\_CORK的socket option是也关闭Nagle算法，这个还不够准确。**TCP\_CORK是禁止小包发送，而Nagle算法没有禁止小包发送，只是禁止了大量的小包发送**。最好不要两个选项都设置。**老实说，我觉得Nagle算法其实只加了个延时，没有别的什么，我觉得最好还是把他关闭，然后由自己的应用层来控制数据，我个觉得不应该什么事都去依赖内核算法**。

**TCP的拥塞处理 – Congestion Handling**

上面我们知道了，TCP通过Sliding Window来做流控（Flow Control），但是TCP觉得这还不够，因为Sliding Window需要依赖于连接的发送端和接收端，其并不知道网络中间发生了什么。TCP的设计者觉得，一个伟大而牛逼的协议仅仅做到流控并不够，因为流控只是网络模型4层以上的事，TCP的还应该更聪明地知道整个网络上的事。

具体一点，我们知道TCP通过一个timer采样了RTT并计算RTO，但是，**如果网络上的延时突然增加，那么，TCP对这个事做出的应对只有重传数据，但是，重传会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，于是，这个情况就会进入恶性循环被不断地放大。试想一下，如果一个网络内有成千上万的TCP连接都这么行事，那么马上就会形成“网络风暴”，TCP这个协议就会拖垮整个网络。**这是一个灾难。

所以，TCP不能忽略网络上发生的事情，而无脑地一个劲地重发数据，对网络造成更大的伤害。对此TCP的设计理念是：**TCP不是一个自私的协议，当拥塞发生的时候，要做自我牺牲。就像交通阻塞一样，每个车都应该把路让出来，而不要再去抢路了。**

关于拥塞控制的论文请参看《[Congestion Avoidance and Control](http://ee.lbl.gov/papers/congavoid.pdf)》(PDF)

拥塞控制主要是四个算法：**1）慢启动**，**2）拥塞避免**，**3）拥塞发生**，**4）快速恢复**。这四个算法不是一天都搞出来的，这个四算法的发展经历了很多时间，到今天都还在优化中。 备注:

* 1988年，TCP-Tahoe 提出了1）慢启动，2）拥塞避免，3）拥塞发生时的快速重传
* 1990年，TCP Reno 在Tahoe的基础上增加了4）快速恢复

**慢热启动算法 – Slow Start**

首先，我们来看一下TCP的慢热启动。慢启动的意思是，刚刚加入网络的连接，一点一点地提速，不要一上来就像那些特权车一样霸道地把路占满。新同学上高速还是要慢一点，不要把已经在高速上的秩序给搞乱了。

慢启动的算法如下(cwnd全称Congestion Window)：

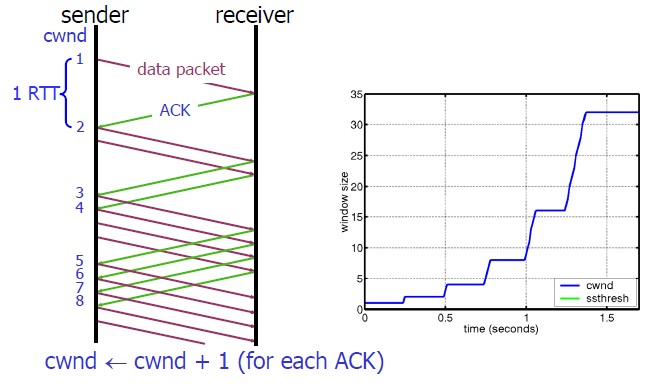
1）连接建好的开始先初始化cwnd = 1，表明可以传一个MSS大小的数据。

2）每当收到一个ACK，cwnd++; 呈线性上升

3）每当过了一个RTT，cwnd = cwnd\*2; 呈指数让升

4）还有一个ssthresh（slow start threshold），是一个上限，当cwnd >= ssthresh时，就会进入“拥塞避免算法”（后面会说这个算法）

所以，我们可以看到，如果网速很快的话，ACK也会返回得快，RTT也会短，那么，这个慢启动就一点也不慢。下图说明了这个过程。



这里，我需要提一下的是一篇Google的论文《[An Argument for Increasing TCP’s Initial Congestion Window](http://static.googleusercontent.com/media/research.google.com/zh-CN/pubs/archive/36640.pdf)》Linux 3.0后采用了这篇论文的建议——把cwnd 初始化成了 10个MSS。 而Linux 3.0以前，比如2.6，Linux采用了[RFC3390](http://www.rfc-editor.org/rfc/rfc3390.txt)，cwnd是跟MSS的值来变的，如果MSS< 1095，则cwnd = 4；如果MSS>2190，则cwnd=2；其它情况下，则是3。

**拥塞避免算法 – Congestion Avoidance**

前面说过，还有一个ssthresh（slow start threshold），是一个上限，当cwnd >= ssthresh时，就会进入“拥塞避免算法”。一般来说ssthresh的值是65535，单位是字节，当cwnd达到这个值时后，算法如下：

1）收到一个ACK时，cwnd = cwnd + 1/cwnd

2）当每过一个RTT时，cwnd = cwnd + 1

这样就可以避免增长过快导致网络拥塞，慢慢的增加调整到网络的最佳值。很明显，是一个线性上升的算法。

**拥塞状态时的算法**

前面我们说过，当丢包的时候，会有两种情况：

1）等到RTO超时，重传数据包。TCP认为这种情况太糟糕，反应也很强烈。

* + sshthresh =  cwnd /2
  + cwnd 重置为 1
  + 进入慢启动过程

2）Fast Retransmit算法，也就是在收到3个duplicate ACK时就开启重传，而不用等到RTO超时。

* + TCP Tahoe的实现和RTO超时一样。
  + TCP Reno的实现是：
    - cwnd = cwnd /2
    - sshthresh = cwnd
    - 进入快速恢复算法——Fast Recovery

上面我们可以看到RTO超时后，sshthresh会变成cwnd的一半，这意味着，如果cwnd<=sshthresh时出现的丢包，那么TCP的sshthresh就会减了一半，然后等cwnd又很快地以指数级增涨爬到这个地方时，就会成慢慢的线性增涨。我们可以看到，TCP是怎么通过这种强烈地震荡快速而小心得找到网站流量的平衡点的。

**快速恢复算法 – Fast Recovery**

**TCP Reno**

这个算法定义在[RFC5681](http://tools.ietf.org/html/rfc5681)。快速重传和快速恢复算法一般同时使用。快速恢复算法是认为，你还有3个Duplicated Acks说明网络也不那么糟糕，所以没有必要像RTO超时那么强烈。 注意，正如前面所说，进入Fast Recovery之前，cwnd 和 sshthresh已被更新：

* cwnd = cwnd /2
* sshthresh = cwnd

然后，真正的Fast Recovery算法如下：

* cwnd = sshthresh  + 3 \* MSS （3的意思是确认有3个数据包被收到了）
* 重传Duplicated ACKs指定的数据包
* 如果再收到 duplicated Acks，那么cwnd = cwnd +1
* 如果收到了新的Ack，那么，cwnd = sshthresh ，然后就进入了拥塞避免的算法了。

如果你仔细思考一下上面的这个算法，你就会知道，**上面这个算法也有问题，那就是——它依赖于3个重复的Acks**。注意，3个重复的Acks并不代表只丢了一个数据包，很有可能是丢了好多包。但这个算法只会重传一个，而剩下的那些包只能等到RTO超时，于是，进入了恶梦模式——超时一个窗口就减半一下，多个超时会超成TCP的传输速度呈级数下降，而且也不会触发Fast Recovery算法了。

通常来说，正如我们前面所说的，SACK或D-SACK的方法可以让Fast Recovery或Sender在做决定时更聪明一些，但是并不是所有的TCP的实现都支持SACK（SACK需要两端都支持），所以，需要一个没有SACK的解决方案。而通过SACK进行拥塞控制的算法是FACK（后面会讲）

**TCP New Reno**

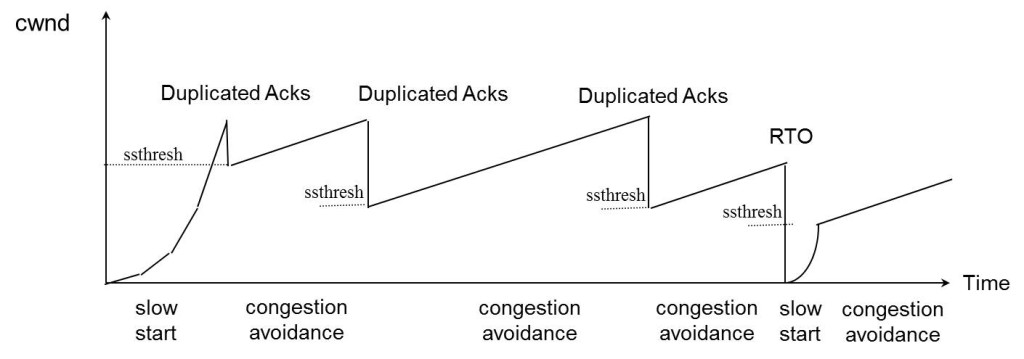
于是，1995年，TCP New Reno（参见 [RFC 6582](http://tools.ietf.org/html/rfc6582) ）算法提出来，主要就是在没有SACK的支持下改进Fast Recovery算法的——

* 当sender这边收到了3个Duplicated Acks，进入Fast Retransimit模式，开发重传重复Acks指示的那个包。如果只有这一个包丢了，那么，重传这个包后回来的Ack会把整个已经被sender传输出去的数据ack回来。如果没有的话，说明有多个包丢了。我们叫这个ACK为Partial ACK。
* 一旦Sender这边发现了Partial ACK出现，那么，sender就可以推理出来有多个包被丢了，于是乎继续重传sliding window里未被ack的第一个包。直到再也收不到了Partial Ack，才真正结束Fast Recovery这个过程

我们可以看到，这个“Fast Recovery的变更”是一个非常激进的玩法，他同时延长了Fast Retransmit和Fast Recovery的过程。

**算法示意图**

下面我们来看一个简单的图示以同时看一下上面的各种算法的样子：



**FACK算法**

FACK全称Forward Acknowledgment 算法，论文地址在这里（PDF）[Forward Acknowledgement: Refining TCP Congestion Control](http://conferences.sigcomm.org/sigcomm/1996/papers/mathis.pdf) 这个算法是其于SACK的，前面我们说过SACK是使用了TCP扩展字段Ack了有哪些数据收到，哪些数据没有收到，他比Fast Retransmit的3 个duplicated acks好处在于，前者只知道有包丢了，不知道是一个还是多个，而SACK可以准确的知道有哪些包丢了。 所以，SACK可以让发送端这边在重传过程中，把那些丢掉的包重传，而不是一个一个的传，但这样的一来，如果重传的包数据比较多的话，又会导致本来就很忙的网络就更忙了。所以，FACK用来做重传过程中的拥塞流控。

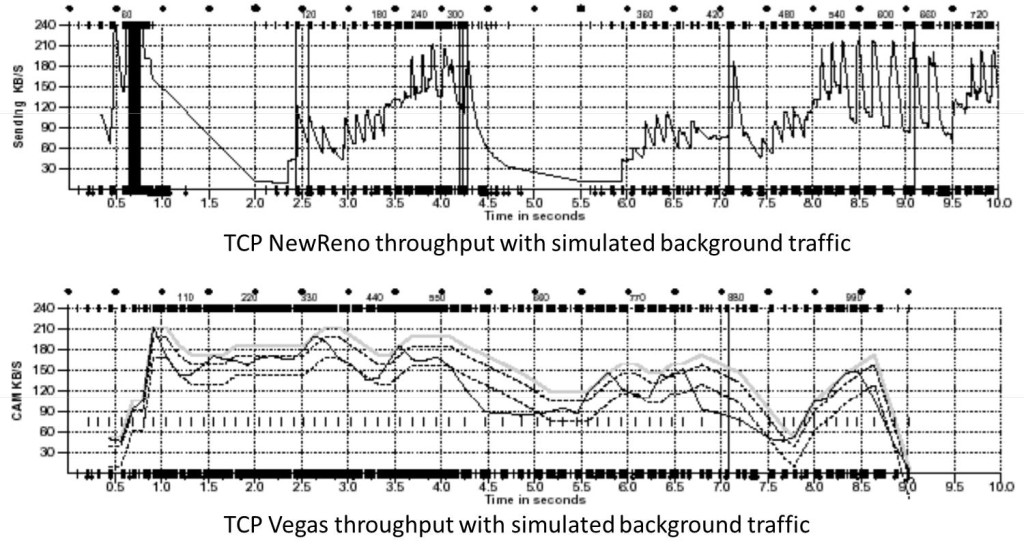
* 这个算法会把SACK中最大的Sequence Number 保存在**snd.fack**这个变量中，snd.fack的更新由ack带秋，如果网络一切安好则和snd.una一样（snd.una就是还没有收到ack的地方，也就是前面sliding window里的category #2的第一个地方）
* 然后定义一个**awnd = snd.nxt – snd.fack**（snd.nxt指向发送端sliding window中正在要被发送的地方——前面sliding windows图示的category#3第一个位置），这样awnd的意思就是在网络上的数据。（所谓awnd意为：actual quantity of data outstanding in the network）
* 如果需要重传数据，那么，**awnd = snd.nxt – snd.fack + retran\_data**，也就是说，awnd是传出去的数据 + 重传的数据。
* 然后触发Fast Recovery 的条件是： (**( snd.fack – snd.una ) > (3\*MSS)**) || (dupacks == 3) ) 。这样一来，就不需要等到3个duplicated acks才重传，而是只要sack中的最大的一个数据和ack的数据比较长了（3个MSS），那就触发重传。在整个重传过程中cwnd不变。直到当第一次丢包的snd.nxt<=snd.una（也就是重传的数据都被确认了），然后进来拥塞避免机制——cwnd线性上涨。

我们可以看到如果没有FACK在，那么在丢包比较多的情况下，原来保守的算法会低估了需要使用的window的大小，而需要几个RTT的时间才会完成恢复，而FACK会比较激进地来干这事。 但是，FACK如果在一个网络包会被 reordering的网络里会有很大的问题。

**其它拥塞控制算法简介**

**TCP Vegas 拥塞控制算法**

这个算法1994年被提出，它主要对TCP Reno 做了些修改。这个算法通过对RTT的非常重的监控来计算一个基准RTT。然后通过这个基准RTT来估计当前的网络实际带宽，如果实际带宽比我们的期望的带宽要小或是要多的活，那么就开始线性地减少或增加cwnd的大小。如果这个计算出来的RTT大于了Timeout后，那么，不等ack超时就直接重传。（Vegas 的核心思想是用RTT的值来影响拥塞窗口，而不是通过丢包） 这个算法的论文是《[TCP Vegas: End to End Congestion Avoidance on a Global Internet](http://www.cs.cmu.edu/~srini/15-744/F02/readings/BP95.pdf)》这篇论文给了Vegas和 New Reno的对比：



关于这个算法实现，你可以参看Linux源码：[/net/ipv4/tcp\_vegas.h](http://lxr.free-electrons.com/source/net/ipv4/tcp_vegas.h)， [/net/ipv4/tcp\_vegas.c](http://lxr.free-electrons.com/source/net/ipv4/tcp_vegas.c)

**HSTCP(High Speed TCP) 算法**

这个算法来自[RFC 3649](http://tools.ietf.org/html/rfc3649)（[Wikipedia词条](http://en.wikipedia.org/wiki/HSTCP)）。其对最基础的算法进行了更改，他使得Congestion Window涨得快，减得慢。其中：

* 拥塞避免时的窗口增长方式： cwnd = cwnd + α(cwnd) / cwnd
* 丢包后窗口下降方式：cwnd = (1- β(cwnd))\*cwnd

注：α(cwnd)和β(cwnd)都是函数，如果你要让他们和标准的TCP一样，那么让α(cwnd)=1，β(cwnd)=0.5就可以了。 对于α(cwnd)和β(cwnd)的值是个动态的变换的东西。 关于这个算法的实现，你可以参看Linux源码：[/net/ipv4/tcp\_highspeed.c](http://lxr.free-electrons.com/source/net/ipv4/tcp_highspeed.c)

**TCP BIC 算法**

2004年，产内出BIC算法。现在你还可以查得到相关的新闻《Google：[美科学家研发BIC-TCP协议 速度是DSL六千倍](https://www.google.com/search?lr=lang_zh-CN%7Clang_zh-TW&newwindow=1&biw=1366&bih=597&tbs=lr%3Alang_1zh-CN%7Clang_1zh-TW&q=%E7%BE%8E%E7%A7%91%E5%AD%A6%E5%AE%B6%E7%A0%94%E5%8F%91BIC-TCP%E5%8D%8F%E8%AE%AE+%E9%80%9F%E5%BA%A6%E6%98%AFDSL%E5%85%AD%E5%8D%83%E5%80%8D&oq=%E7%BE%8E%E7%A7%91%E5%AD%A6%E5%AE%B6%E7%A0%94%E5%8F%91BIC-TCP%E5%8D%8F%E8%AE%AE+%E9%80%9F%E5%BA%A6%E6%98%AFDSL%E5%85%AD%E5%8D%83%E5%80%8D)》 BIC全称[Binary Increase Congestion control](http://research.csc.ncsu.edu/netsrv/?q=content/bic-and-cubic)，在Linux 2.6.8中是默认拥塞控制算法。BIC的发明者发这么多的拥塞控制算法都在努力找一个合适的cwnd – Congestion Window，而且BIC-TCP的提出者们看穿了事情的本质，其实这就是一个搜索的过程，所以BIC这个算法主要用的是Binary Search——二分查找来干这个事。 关于这个算法实现，你可以参看Linux源码：[/net/ipv4/tcp\_bic.c](http://lxr.free-electrons.com/source/net/ipv4/tcp_bic.c)

**TCP WestWood算法**

westwood采用和Reno相同的慢启动算法、拥塞避免算法。westwood的主要改进方面：在发送端做带宽估计，当探测到丢包时，根据带宽值来设置拥塞窗口、慢启动阈值。 那么，这个算法是怎么测量带宽的？每个RTT时间，会测量一次带宽，测量带宽的公式很简单，就是这段RTT内成功被ack了多少字节。因为，这个带宽和用RTT计算RTO一样，也是需要从每个样本来平滑到一个值的——也是用一个加权移平均的公式。 另外，我们知道，如果一个网络的带宽是每秒可以发送X个字节，而RTT是一个数据发出去后确认需要的时候，所以，X \* RTT应该是我们缓冲区大小。所以，在这个算法中，ssthresh的值就是est\_BD \* min-RTT(最小的RTT值)，如果丢包是Duplicated ACKs引起的，那么如果cwnd > ssthresh，则 cwin = ssthresh。如果是RTO引起的，cwnd = 1，进入慢启动。   关于这个算法实现，你可以参看Linux源码： [/net/ipv4/tcp\_westwood.c](http://lxr.free-electrons.com/source/net/ipv4/tcp_westwood.c)

**其它**

更多的算法，你可以从Wikipedia的 [TCP Congestion Avoidance Algorithm](http://en.wikipedia.org/wiki/TCP_congestion-avoidance_algorithm) 词条中找到相关的线索

**后记**

好了，到这里我想可以结束了，TCP发展到今天，里面的东西可以写上好几本书。本文主要目的，还是把你带入这些古典的基础技术和知识中，希望本文能让你了解TCP，更希望本文能让你开始有学习这些基础或底层知识的兴趣和信心。

当然，TCP东西太多了，不同的人可能有不同的理解，而且本文可能也会有一些荒谬之言甚至错误，还希望得到您的反馈和批评。