河南城建学院
Henan University of Urban Construction

《计算机网络》
论文报告

学　　院：计算机与科学技术

班　　级：0814162

学　　号：081416202

姓　　名：阙文斌

TCP 拥塞控制

摘要：本次论文以 5-39 题展开论述，讨论 TCP 的拥塞窗口随着传输轮次 n 的变化，在各个阶段使用不同 TCP 拥塞控制算法。通过实例详细论述各个算法的工作机制及工作时段。在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，即∑对资源的需求>可用资源。这种情况就叫做拥塞。而 TCP 作为整个网络运输层的通信协议，就有拥塞避免的算法。它使用一套线增积减的模式多样化网络拥塞控制方法来控制拥塞。其主要的算法有四种即慢开始、拥塞避免、快重传、快恢复。

关键词：拥塞避免；慢开始；快重传；快恢复；

一、前言

在题 5-39 中给出了 TCP 的拥塞窗口 cwnd 大小与传输轮次 n 的关系表。

cwnd	1	2	4	8	16	32	33	34	35	36	37	38	39
n	1	2	3	4	5	6	7	8	9	10	11	12	13
cwnd	40	41	42	21	22	23	24	25	26	1	2	4	8
n	14	15	16	17	18	19	20	21	22	23	24	25	26

我们可以做出 cwnd 与传输轮次 n 的关系曲线图，更为直观地观察它们之间的关系。

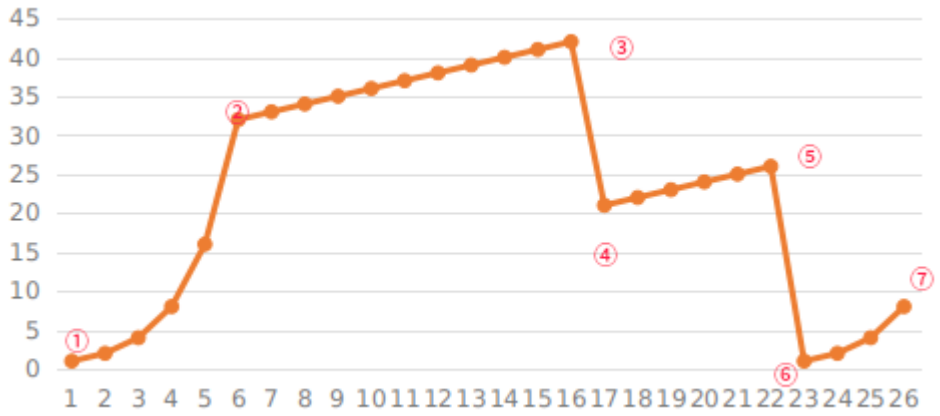

图 1：cwnd 与传输轮次的关系曲线图

为了更清楚地理解曲线的变化情况，我们先介绍四种 TCP 拥塞算法，在介绍这些拥塞算法原理中当中来解释各个点的意义。

二、慢开始

TCP 拥塞控制的主要原理主要是通过一个发送方的拥塞窗口（cwnd）来控制，窗口值的大小代表能够发送出去但还没有接送的 ACK 的最大数据报文段。cwnd 越大那么显然数据发送的速度越快，但网络出现拥塞的可能性越大。TCP 拥塞控制算法就是在这两者之间平衡，找到一个最好的拥塞窗口值，使网络吞吐量最大且不产生拥塞。判断网络拥塞的根据就是出现了超时。通过拥塞窗口值的大小来控制拥塞，那首先得判断出拥塞窗口增大到多少时会出现超时的情况。那么最好

的方法就是从小到大逐渐增大拥塞窗口数值。即是慢开始算法的思想。具体来说，当新的连接建立时，我们将拥塞窗口大小设置为 1 个最大报文段（SMSS）数值。开始的时候发送端设置 cwnd=1，每当有一个报文段被确认接收，就将 cwnd 增加一个 SMSS 大小的值。这样拥塞窗口的值就随着一个传输轮次的往返时间 RTT 呈指数形式增长。如下图所示：

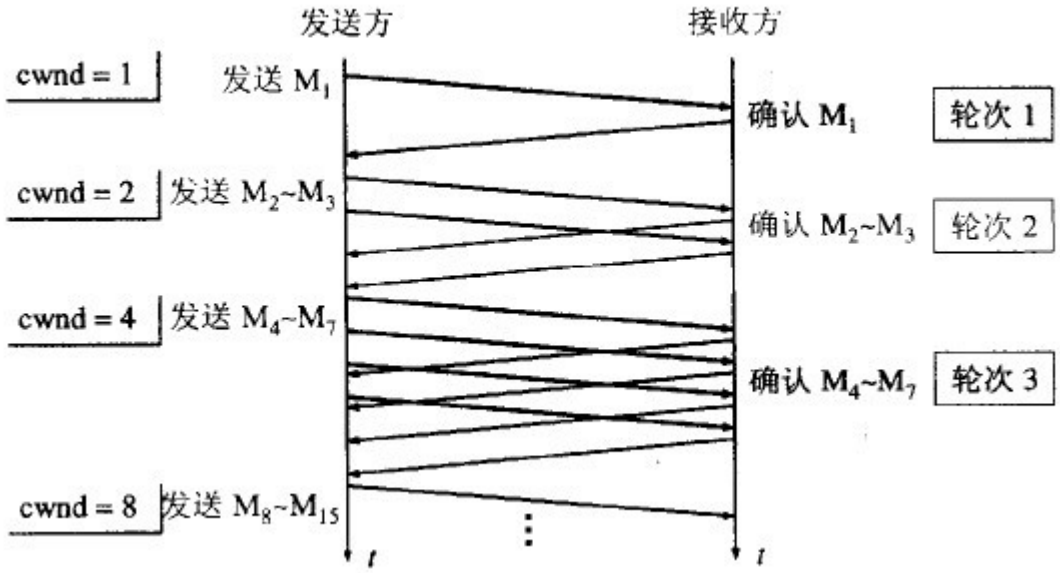

图 2：发送方每接受一个确认就把窗口 cwnd 加 1

我们可以简单计算下：开始 ---> cwnd=1

经过 1 个 RTT 后 ---> cwnd=2*1=2

经过 2 个 RTT 后 ---> cwnd=2*2=4

经过 3 个 RTT 后 ---> cwnd=4*1=8

可见慢开始并不慢，只是开始的起点比较低而已。这样逐渐增大拥塞窗口值的方式比起一下子就把许多报文段输入到网络中要安全的多。更不容易引起网络的拥塞。在我们了解慢开始的工作原理后，我们就可以从在题 5-39 的关系曲线图 1 中很清楚地看到点①和点⑤就是慢开始开始的起点，且根据慢开始 cwnd 增长特点可知在[1-6],[23-26]轮次间隔中是慢开始工作阶段。

三、拥塞避免

随着慢开始算法的实行，拥塞窗口也会增长过大引起网络拥塞。这就需要设置一个慢开始门限 ssthresh 状态变量，当拥塞窗口 cwnd 数值增长到 ssthresh 门限的值时，事实上拥塞窗口的值相对来说比较大了，如果再成倍增加网络拥塞

的可能性就比较大了，这时就需要进入拥塞避免算法了。再从图 1 的关系曲线图中，我们就可以看到点②就是慢开始结束的点，也是拥塞避免算法开始的起点，即点②此时对应的 cwnd 值就是慢开始门限 ssthresh 的值为 32，也就是在第 1 轮次发送时，门限 ssthresh=32。拥塞避免算法的思路就是让拥塞窗口缓慢增大，不同于慢开始的成倍缓慢增大，拥塞避免算法是每经过一个轮次的往返时间就让拥塞窗口增加 1，它的特点是”加法增大”，使拥塞窗口的值呈线性缓慢增大，这要比慢开始算法增长速率缓慢的多。拥塞避免并不能完全避免拥塞，只是使网络不容易出现拥塞。如下图所示：

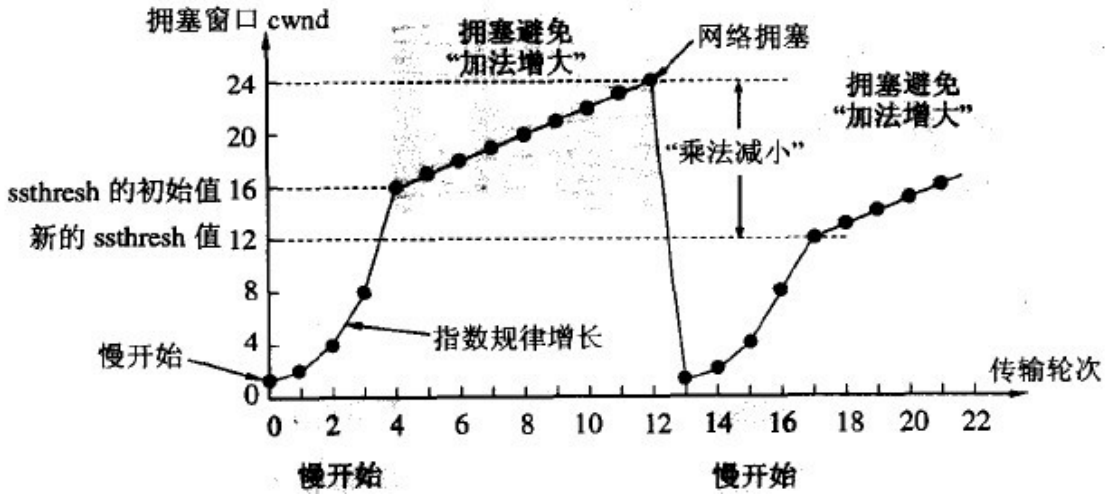

图 3：慢开始和拥塞避免算法的实现举例

当拥塞窗口随着拥塞避免算法的线性递增，网络出现了超时，网络发生拥塞，这是发送方的拥塞窗口门限值就会调整为 ssthresh=cwnd/2=12，且重新设置拥塞窗口 cwnd=1，进入慢开始重复这个过程，TCP 经过这种强烈快速的震荡方式，一点点找到网络传输的平衡点。在图 1 中根据拥塞避免算法 cwnd 增长特点，我们可以知道[6-16],[17-22]轮次间隔中是拥塞避免算法工作阶段，经过简单计算就可以知道第 70 个报文段就是在此时间间隔的第 7 次轮次中发送出的。再从图 3 这个过程和图 1 进行比较我们不难看出在图 1 点⑥时 cwnd 窗口值突然从线性增长的方式转变为 cwnd 初始为 1 呈指数增长，这是就可以判断在点⑤时网络出现超时情况，TCP 结束拥塞避免算法进而重新开始慢开始算法，此时的拥塞窗口门限值应调整为是 ssthresh=cwnd/2=13，即第 24 轮次发送时，门限值 ssthresh=13。

四、快重传和快恢复算法

网络传输并不是总是稳定的，有时，个别报文段会在网络中丢失，其实这时

的网络并未方式拥塞，但是发送方一直没有接收到接收端的确认，发送端就会错误的认为网络发生了拥塞，于是启动慢开始。又将拥塞窗口设置为 1，降低了传输效率。快重传算法就是解决这个问题，让发送方尽快知道发生了个别报文段的丢失。即只要发送方收到了连续的 3 个重复确认，就确定接收方确实没有收到这个报文段，就可以立即进行重传，这样就不会出现超时的情况发生。过程如下图所示：

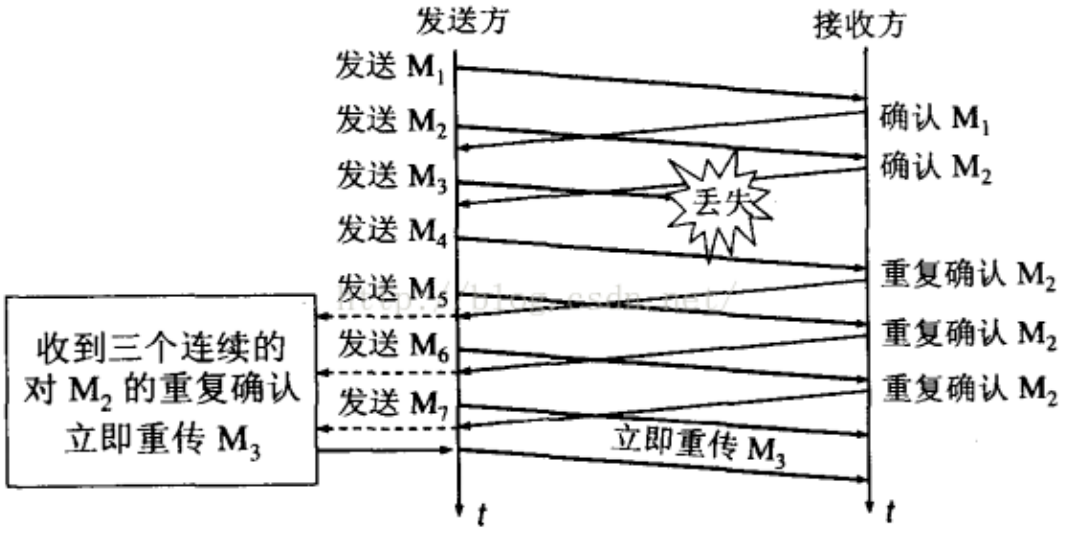

图 4：快重传示意图

如上图所示，发送方知道只是丢失了个别报文段以后就不再启动慢开始算法，而是执行快恢复算法，于是发送方调整拥塞窗口门限值 ssthresh=cwnd/2，同时设置拥塞窗口 cwnd=ssthresh=8，并开始执行拥塞避免算法。这种方式常被成为“乘法减小”。这就是 TCP 通过线增积小的模式控制网络拥塞的方法。通过这种拥塞控制的方法使得 TCP 的性能有明显的改进。结合快重传、快恢复算法以及拥塞避免算法的工作机制，我们就可以从图①中看出当点③结束拥塞避免算法时拥塞窗口 cwnd 并没有初始为 1，而是减小一倍即执行了快重传和快恢复算法。由此我们可以判断点③并没有出行网络超时而是发送方接收到了三个重复确认，而此时的拥塞窗口门限值 ssthresh=cwnd/2=21，即第 18 轮次发送时，门限 ssthresh=21。再假定第 26 轮次后又收到三次重复确认，发送方知道是丢失报文段时，这是拥塞窗口 cwnd 和 ssthresh 应该设置为 cwnd/2=4。

五、总结

通过对 TCP 四种拥塞控制的研究和题 5-39 的实例，让我对 TCP 的拥塞控制有了更加深入的理解。更加明白现在网络的发展，离不开这些优秀的算法。因为

本人对数学一直有很浓厚的兴趣，而这些算法本质上来说是数学的实际应用，再随着对这四种算法的研究我的兴致高昂起来。我明白课本上的这四种算法只是众多拥塞控制的算法的基础，而不管是为了此时的一时兴起还是为了以后的继续学习，我都忍不住在网络上搜索了一些最新的 TCP 拥塞算法。初次接触，我只能算是小白，也不可能有什么有用的见解，所以只是将前辈们对这些最新的算法的解析和理解放在下文中，以供我以后的学习，并不算的论文正文。

1、New Reno 算法。在课堂上老师讲解快重传和快恢复算法时，我就一直有个疑问，那就是如果发送方接收到了 3 个连续重复确认并不止意味着传输中只丢失了一个包，而是丢失了多个包，那么按照快重传和快恢复算法的工作机制，这个算法只会快速重传一个，剩下的包只能等到 RTO 超时，再次进入快重传和快恢复算法，这样就会导致传输就进入了恶循环模式，确定超时一个拥塞窗口 cwnd 值就会减半，多个超时就会造成 TCP 传输速度呈指数下降，正如慢开始呈指数增长一般倒退回去，网络传输效率会骤降，岂不是得不偿失？

结果在网上了解了一下，原来针对这个情况，早在 1995 年就有提出 TCP New Reno 算法，New Reno 算法主要是在没有选择性确认(sack)的支持下改进快恢复算法。当发送方连续接收到了三个连续重复确认，进入快重传，重传重复确认指定的那个包。如果只有一个包丢失了，那么重传这个包后回来的确认会对整个已经被发送方传输的数据包确认回来。如果没有，就说明是多个包丢失，我们称为部分确认。而发送方一但收到部分确认出现，就可以推测出来是有多个包丢失了，于是继续重传滑动窗口里没有被确认的第一个包，直到发送方接收不到部分确认为止，这时才真正结束快恢复算法。很显然这个算法延长了快恢复算法和快重传算法的过程。但好处是 TCP 可以在发生拥塞是判断出丢失一个数据包还是丢失多个数据包，使每次发生拥塞时保证拥塞窗口的值只减半一次，保证了 TCP 的吞吐量。

2、CUBIC 算法。Cubic[3]是 Linux 内核 2.6 之后的默认 TCP 拥塞控制算法，使用一个立方函数（cubic function）作为拥塞窗口的增长函数，即：

$$f(x) = C(x - \sqrt[3]{\frac{(1-\beta)W_{max}}{C}})^3 + W_{max}$$

图:5：TCP Cubic 拥塞窗口增长函数

其中，C 是调节因子，x 是从上一次缩小拥塞窗口经过的时间，Wmax 是上一次发生拥塞时的窗口大小，， β 是乘法减小因子。从函数中可以看出拥塞窗口的增长不再与 RTT 有关，而仅仅取决上次发生拥塞时的最大窗口和距离上次发生拥塞的时间间隔值。Cubic 拥塞窗口增长曲线图如下：

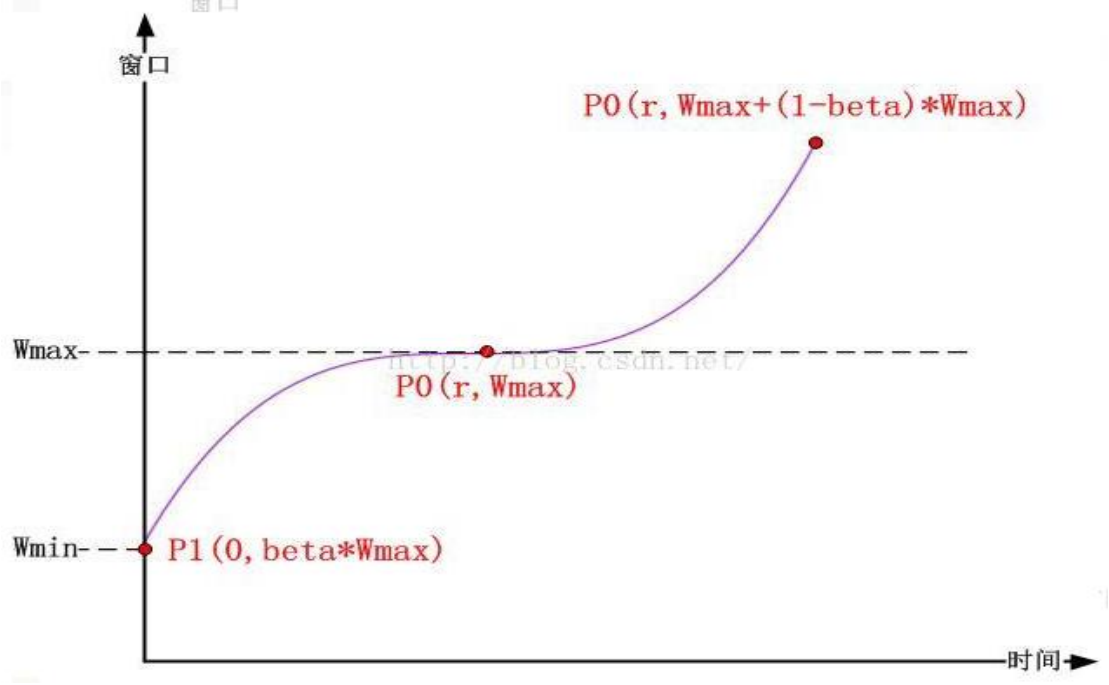

图 6：Cubic 拥塞控制增长曲线图

凸曲线部分为稳定增长阶段，凹曲线部分为最大带宽探测阶段。如图所示，在刚开始时，拥塞窗口增长很快，在接近 Wmax 口时，增长速度变的平缓，避免流量突增而导致丢包；在 Wmax 附近，拥塞窗口不再增加；之后开始缓慢地探测网络最大吞吐量，保证稳定性，在远离 Wmax 后，增大窗口增长的速度，保证了带宽的用率。当出现丢包时，将拥塞窗口进行乘法减小，再继续开始上述增长过程。此方式可以使得拥塞窗口一直维持在 Wmax 附近，从而保证了带宽的利用率。Cubic 算法的优点在于只要没有出现丢包，就不会主动降低自己的发送速度，可以最大程度的利用网络剩余带宽，提高吞吐量，在高带宽、低丢包率的网络中可以发挥较好的性能。

3、BBR 算法。BBR 算法是 Google 在 2016 年底提出的一种新的拥塞控制算法。Google 将 TCP BBR 拥塞控制算法提交到了 Linux 内核，从 4.9 开始，Linux 内核已经用上了该算法。根据实地测试，在部署了最新版内核并开启了 TCP BBR 的机器上，网速甚至可以提升好几个数量级。

BBR 算法不将出现丢包或时延增加作为拥塞的信号，而是认为当网络上的数据包总量大于瓶颈链路带宽和时延的乘积时才出现了拥塞，所以 BBR 也称为基于拥塞的拥塞控制算法。BBR 算法周期性地探测网络的容量，交替测量一段时间内

的带宽极大值和时延极小值，将其乘积作为作为拥塞窗口大小（交替测量的原因是极大带宽和极小时延不可能同时得到，带宽极大时网络被填满造成排队，时延必然极大，时延极小时需要数据包不被排队直接转发，带宽必然极小），使得拥塞窗口始的值始终与网络的容量保持一致。

BBR 的拥塞窗口是精确测量出来的，不会无限的增加拥塞窗口，也就不会将网络设备的缓冲区填满，避免了出现缓冲区填满问题，使得时延大大降低。

又因为 BBR 算法不将丢包作为拥塞信号，所以在丢包率较高的网络中，BBR 依然有极高的吞吐量。BBR 算法是反馈驱动的，有自主调节机制，不受 TCP 拥塞控制状态机的控制，通过测量网络容量来调整拥塞窗口，发送速率由自己掌控，而传统的拥塞控制算法只负责计算拥塞窗口，而不管发送速率，怎么发由 TCP 自己决定，这样会在瓶颈带宽附近因发送速率的激增导致数据包排队或出现丢包。BBR 更加适用于现在高速的网络，据网络传言 BBR v2.0 版本也即将出炉。

仔细看完这两种算法我已经筋疲力尽了，关于 Cubic 算法算是大略知道其中工作机制，而关于 BBR 实在是不太容易明白，如它是怎么来测量网络容量的、如何算是反驱动的等等。感觉更为复杂，俨然更像一种体系结构或者框架，牵扯很多知识。我不禁察觉到我要学习的东西还是有很多很多。

参考文献：

[1] 谢希仁.计算机网络（第七版）[M].北京：电子工业出版社，2017.1
[2] IG 一直在努力.浅谈 TCP 拥塞控制[EB/OL].博客园，2017.12
[3] Dog250.TCP 拥塞控制算法-从 BIC 到 CUBIC[EB/OL].CSDN，2016.11
[4] Jessica 程序猿.TCP 的拥塞控制[EB/OL].博客园，2015.6