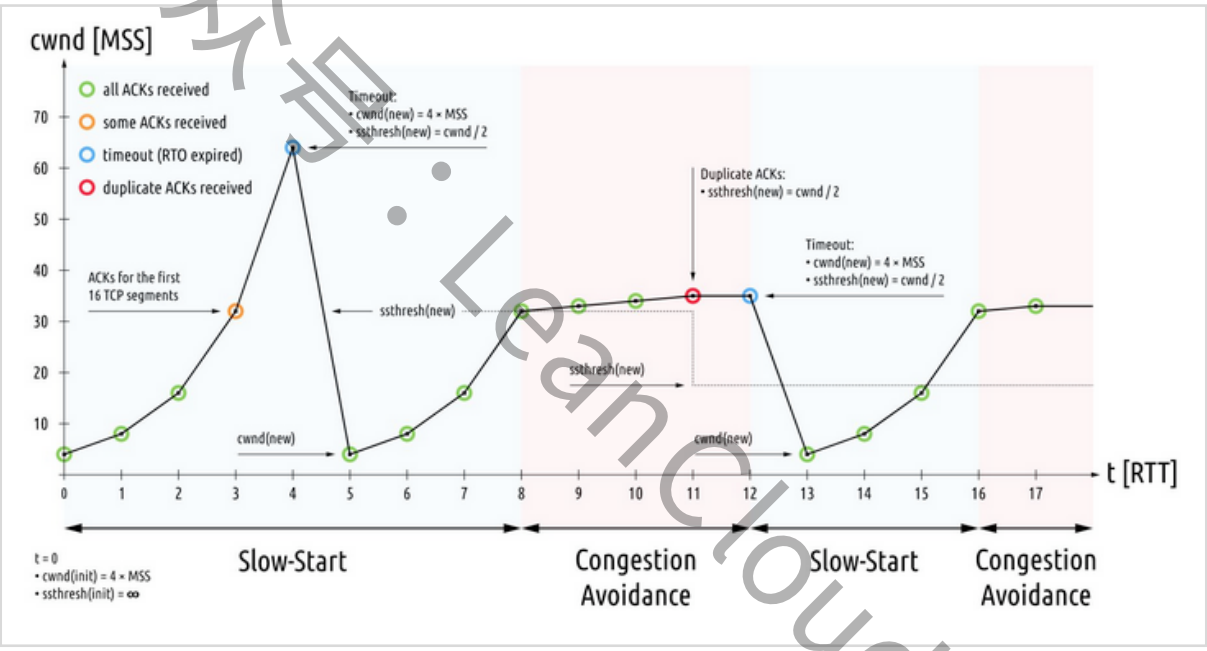
tcp congestion avoidance 算法

#技术/网络

BBR, the new kid on the TCP block | APNIC Blog 这篇文章挺好的，下面很多内容也基于这篇文章而来。

拥塞避免用于避免因为发送者发送数据过快导致链路上因为拥塞而出现丢包。

TCP 连接建立后先经过 Slow Start 阶段，每收到一个 ACK，CWND 翻倍，数据发送率以指数形式增长，等出现丢包，或达到 ssthresh，或到达接收方 RWND 限制后进入 Congestion Avoidance 阶段。下面这个图挺好的，描述了好几个过程，找不到出处了，只是列一下图吧。



一些基础东西可以看：TCP congestion control - Wikipedia

BDP

BDP 是 Bandwidth and Delay Product. 就是带宽 (单位 bps) 和延迟 (单位 s) 的乘积，单位是 bit，也是 Source 和 Destination 之间允许处在 Flying 状态的最大数据量。Flying 也叫 Inflights，就是发送了但还未收到的 Ack 的数据。

来自[3]。

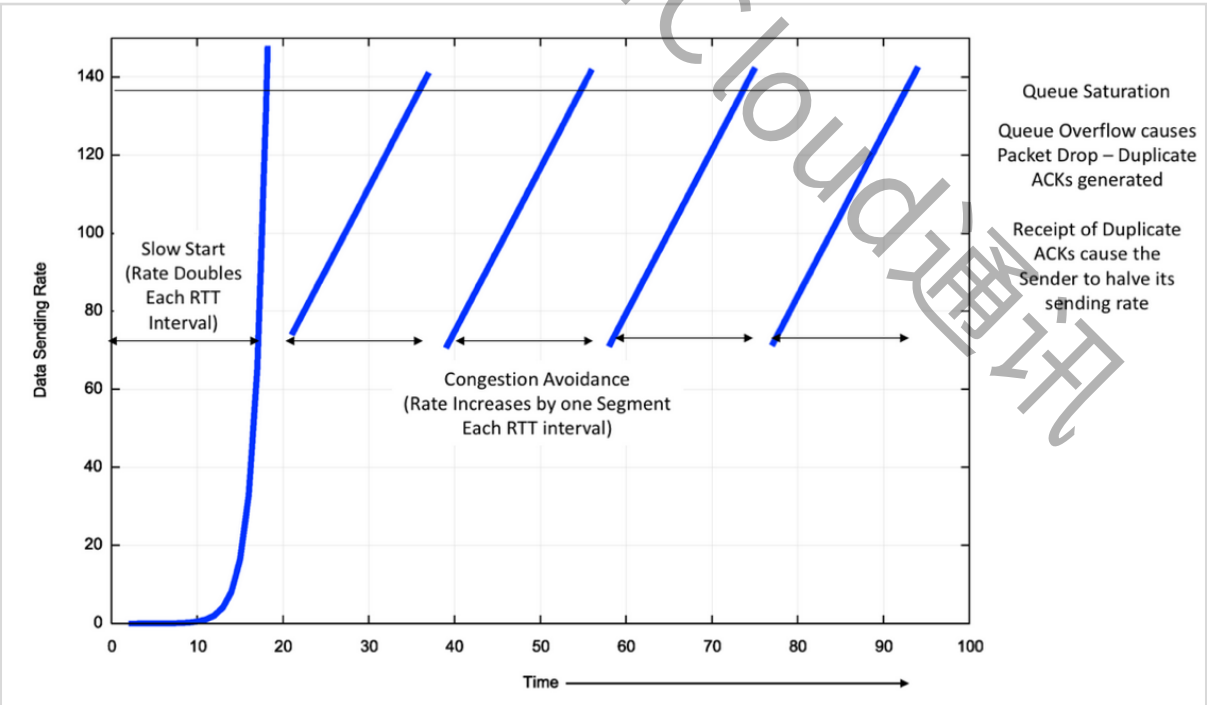
实际发送速率乘以延迟得到的值越接近 BDP 说明算法的效率越高。

Reno

Reno 这种叫做 `ACK-Pacing`，基于 Ack 来确认网络状况。如果能持续收到 ACK，表示网络能正常承载当前发送速率。

> Reno maintains an estimate of the time to send a packet and receive the corresponding ACK (the "round trip time," or RTT), and while the ACK stream is showing that no packets are being lost in transit, then Reno will increase the sending rate by one additional segment each RTT interval.

Reno 下，每收到一个 ACK，CWND 加一，等出现丢包之后发送者将发送速率减半。理想状况下，Reno 能走出如下曲线：



来自[1]

Reno 有如下假设：

- 丢包一定因为网络出现拥塞，但实际可能网络本身不好，可能有固有的丢包概率，所以假设并不严谨；
- 网络拥塞一定是因为网络上某个 buffer overflow 了；
- 网络的 RTT 和带宽稳定不容易变化；
- 将速率减半以后，网络上的 buffer 一定能够从 overflow 变为 drain。也即对网络上 Buffer 大小也有假设；

从上面假设能看出，Reno 下受链路上 Buffer 大小影响很大。当 Buffer 较小的时候，链路上实际处在发送状态的数据量还未达到 BDP (Bandwidth delay product) 时候可能就会出现丢包，导致 Reno 立刻减半发送速率，从而无法高效的利用网络带宽。如果 Buffer 很大，超过 BDP，可能会进入 “Buffer Bloat” 状态，即延迟畸高，因为即使 Reno 速度降为一半依然不能保证使链路 Buffer 清空，或者说可能大部分时间链路上 Buffer 都处在非空状态，且每次 Reno 因为丢包而降速时，会做数据重传，导致之前发送的可能还依旧在链路上排队的数据空占资源没起到作用最终白白丢掉，于是让整条链路上持续存在固有延迟。

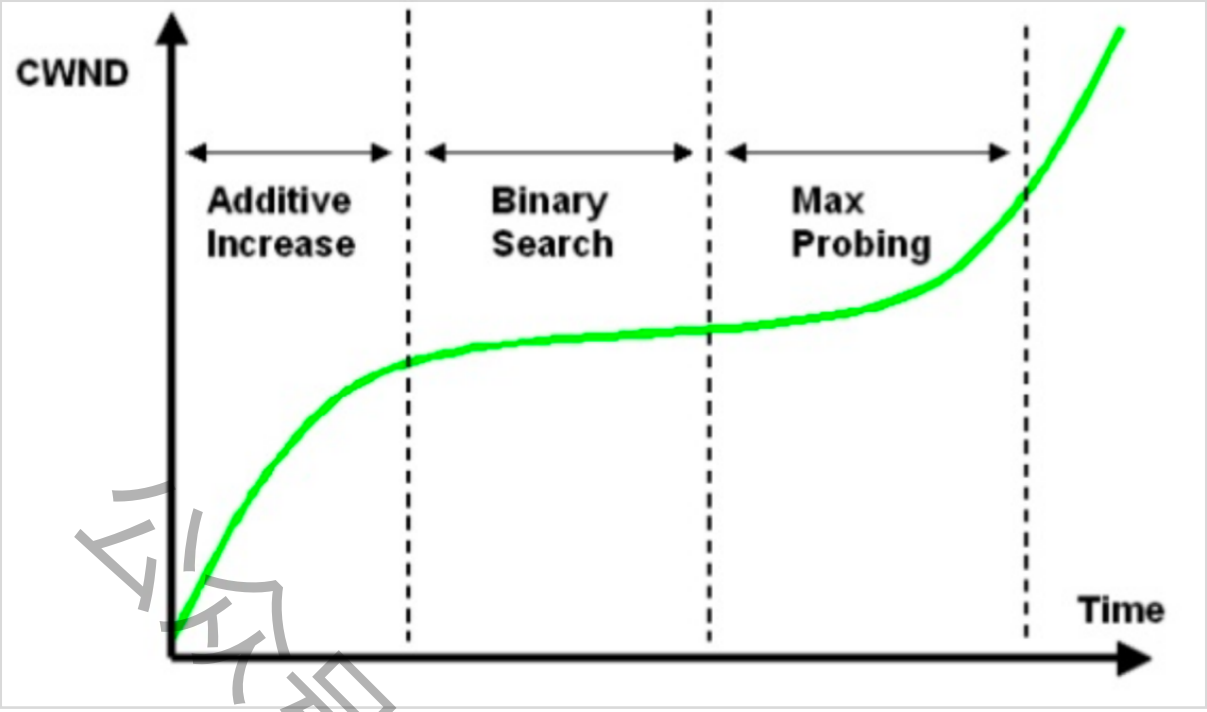
Reno 的问题：

- 上面提到了，受链路 Buffer 影响很大；
- 对高带宽网络，比如带宽是 10Gbps，即使假设延迟非常低只有 30ms，每个 RTT 下 CWND 增加 1500 个八进制数，得好几个小时才能真的利用起这个带宽量，并且要求这几个小时内数据都不能有丢包，不然 Reno 会降速；
- Reno 每收到一个 Ack 就开始扩大 CWND，对链路上一起共享链路的其它 RTT 较大的连接不友好。比如一个连接 RTT 很低，它的 CWND 会比别的共享链路的连接大，不公平的占用更多带宽；

BIC

BIC 叫 Binary Increase Congestion Control，是在 Reno 基础上做改进，将 CWND 扩大过程分成三个阶段，第一阶段是在遇到丢包后将 CWND 降为 `β × Wmax`，（一般 β 是 0.5，Wmax 是丢包时 CWND）但记住之前 Wmax 最大值，之后以一个较快速度增加 CWND，在接近 Wmax 后 CWND 增加量是一个 Wmax 和 CWND 之间的二次函数，称为 Binary Increase，逐步去接近 Wmax，到达 Wmax 后又转换为二次曲线去探测下一个极限。具体内容可以看 Wiki，在这里：BIC TCP - Wikipedia

大概是这么个图，好处就是丢包后能快速恢复，并在稳定期尽力保持更长时间，并还能支持探测更高带宽值。

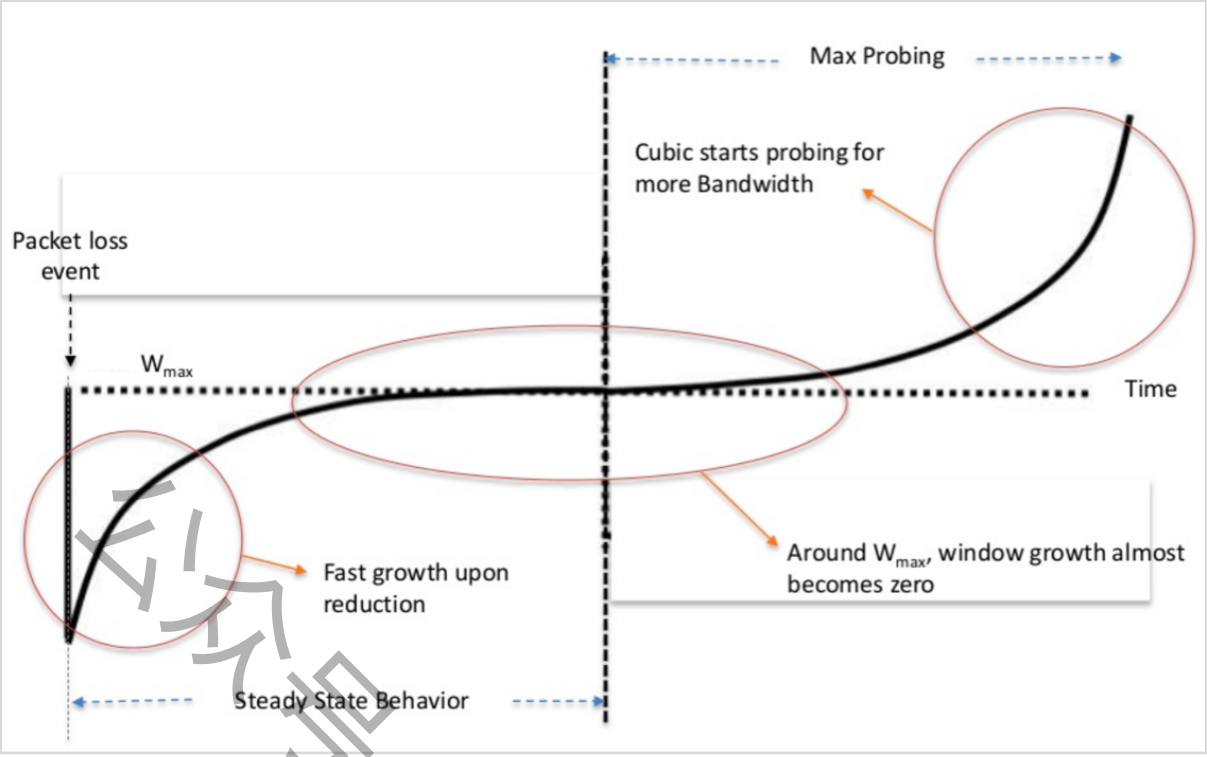


来自[3]

Cubic

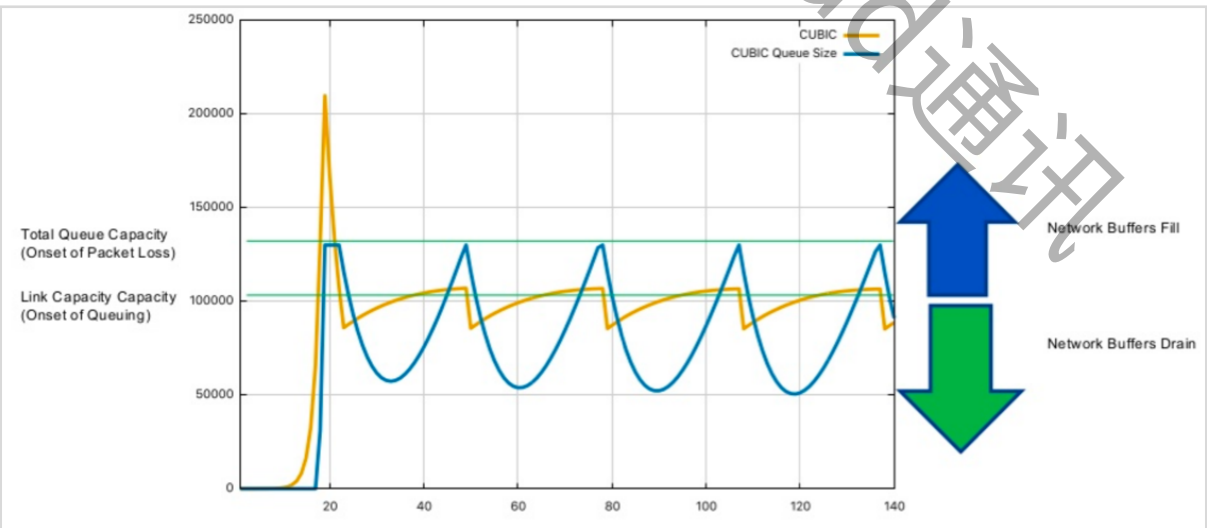
Cubic 在 BIC 的基础上，一个是通过三次函数去平滑 CWND 增长曲线，让它在接近上一个 CWND 最大值时保持的时间更长一些，再有就是让 CWND 的增长和 RTT 长短无关，即不是每次 ACK 后就去增大 CWND，而是让 CWND 增长的三次函数跟时间相关，不管 RTT 多大，一定时间后 CWND 一定增长到某个值，从而让网络更公平，RTT 小的连接不能挤占 RTT 大的连接的资源。

平滑后的 CWND 和时间组成的曲线如下，可以看到三次曲线保留了 BIC 的优点，即在丢包后初期，CWND 能快速增长，减小丢包带来的影响；并且在接近上一个 CWND 最大值时 CWND 增长速度又会非常慢，要经历很长时间才能超越 Wmax，从而能尽力保持稳定，稳定在 Wmax 上；最后在超越 Wmax 后初期也是增长的非常慢，直到后来才快速的指数形式增长，用于探测下一个 Wmax。



来自[3]

Cubic 最终出来的曲线如下。黄色是 CWND，蓝色线是网络上队列拥塞包数。蓝色箭头表示 queue fill，绿色箭头表示 queue drain。看到有两个浅绿色的线，高的是开始丢包，低的是队列开始排队。因为这里画的是说网络上 bandwidth 是固定的，buffer 也是固定的，所以 Cubic 下没有超越 Last Wmax 继续探测下一个 Wmax 的曲线，都是还未到达第一次的 Wmax 时候就因为丢包而被迫降低了 CWND。第一次个黄色尖峰能那么高是因为当时网络上 buffer 是空的，后来每次丢包后 Buffer 还没来得及完全清空 CWND 又涨上来导致数据排队了，所以后来的黄色尖峰都没有第一个高。



来自[3]

Cubic 的优势：

- 因为跟 RTT 无关所以更公平；
- 更适合 BDP 大的网络。Reno 不适合是因为它依赖 RTT，BDP 大的时候 RTT 如果很高，会很久才能将传输效率提上去；

Cubic 缺点：

- 当 Bandwidth 变化时候，Cubic 需要很长时间才能从稳定点进入探测下一个 Wmax 的阶段；
- 更易导致 Bufferbloat。Reno 下如果链路上 Buffer 很大出现拥塞后 RTT 也会很长，很久才会收到 ACK，才会增加 CWND；但 Cubic 的 CWND 增长跟 RTT 无关，到时间就增长，从而更容易加剧链路负担。Bufferbloat 可以看下节。

综合来看 Cubic 适合 BDP 大的高性能网络，性能意思是带宽或者说发送速率，发送速率足够大 Cubic 把 Buffer 占满后才能快速清空，才能有较低的延迟。

一个挺好的讲 Cubic 的 PPT：Cubic，还有论文。Linux 2.6 时候用的 CUBIC。

Bufferbloat

Bufferbloat 在 Bufferbloat - Wikipedia 讲的挺好了。简单说就是随着内存越来越便宜，链路上有些设备的 Buffer 倾向于配置的特别大，而流行的 TCP Congestion Avoidance 算法又是基于丢包的。数据在队列排队说明链路已经出现拥塞，本来应该立即反馈给发送端让发送端减小发送速度的，但因为 Buffer 很大，数据都在排队，发送端根本不知道自己发出去的数据已经开始排队，还在以某个速度 (Reno）甚至更高速度(Cubic，到时间就增加 CWND) 发送。等真的出现丢包时候，发送端依然不知道出现了丢包，还会快速发消息，直到丢的这个包被接收端感知到，回复 ACK 后发送端才终于知道要降低发送速度。而重传的包放在链路上还得等之前的数据包都送达接收端后才能被处理。如果接收端的 Buffer 不足够大，很多数据送达接收端后都会丢弃，得等重传的包到达（Head Of Line 问题）后才能送给上游，大大增加延迟。

Bufferbloat 一方面是会导致网络上超长的延迟，再有就是导致网络传输不稳定，有时候延迟很小，有的时候延迟又很大等。

可以参考：Bufferbloat: Dark Buffers in the Internet - ACM Queue

PRR

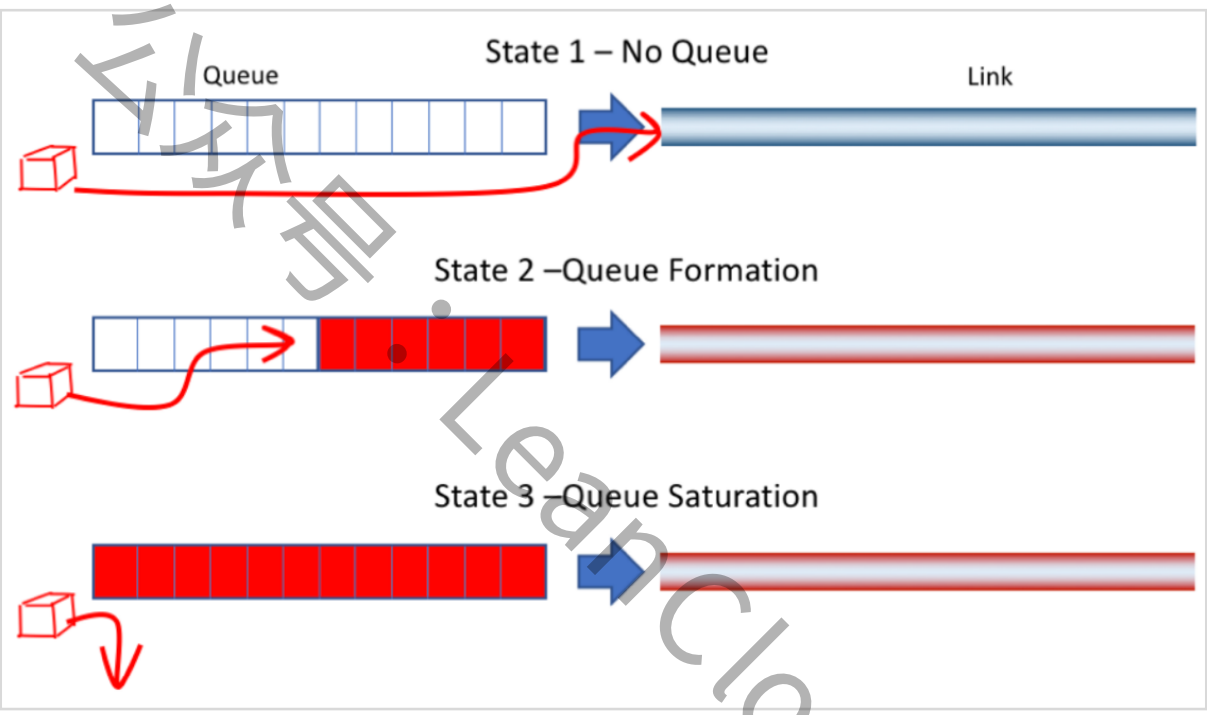
在 CUBIC 之上又有个优化，叫做 Proportional Rate Reduction (PRR)，用以让 CUBIC 这种算法在遇到丢包时候能更快的恢复到当前 CWND 正常值，而不过分的降低到过低的水平。

参考：draft-mathis-tcpm-proportional-rate-reduction-01 - Proportional Rate Reduction for TCP。不进一步记录了。Linux 3.X 的某个版本引入的，配合 CUBIC 一起工作。

链路上的队列模型

为了进一步优化 Cubic，可以先看看链路上队列的模型。

- 当链路上数据较少时，所有数据都在发送链路上进行发送，没有排队的数据。这种情况下延迟最低；
- 当传输的数据更多时候，数据开始排队，延迟开始增大；
- 当队列满的时候进入第三个状态，队列满了出现丢包；



来自[1]

最优状态是 State 1 和 State 2 之间，即没有出现排队导致延迟升高，又能完全占满链路带宽发送数据，又高效延迟又低。
而基于丢包的 Congestion Avoidance 策略都是保持在 State 2 的状态。而 Cubic 是让 CWND 尽可能保持在上一个 Wmax 的状态，也即 State 2 末尾和即将切换到 State 3 的状态。所以 Cubic 相当于尽可能去占用链路资源，使劲发数据把下游链路占满但牺牲了延迟。

于是可以看到，为了保持在 State 1 和 State 2 状态，我们可以监控每个数据包的 RTT，先尽力增大 CWND 提高发送率如果发现发送速率提高后 RTT 没有升高则可以继续提高发送速率，直到对 RTT 有影响时就减速，说明从 State 1 切换到了 State 2。

我们希望让 CWND 尽力保持在下面图中标记为 `optimum operating point` 的点上，即 RTT 又小，链路上带宽又被占满。在这个图上，RTprop 叫做 `round-trip propagation time`，BtlBw 叫做 `bottleneck bandwidth`。横轴是 inflights 数据量，纵轴有两个一个是 RTT 一个是

送达速度。看到图中间有个很淡很淡的黄色的线把图分成了上下两部分，上半部分是 Inflights 数据量和 Round trip time 的关系。下半部分是 Inflights 和 Delivery Rate 的关系。

这里有四个东西需要说一下 CWND、发送速度、inflights，发送者带宽。inflights 就是发送者发到链路中还未收到 ACK 的数据量，发送速度是发送者发送数据的速度，发送者带宽是发送者能达到的最大发送速度，CWND 是根据拥塞算法得到的当前允许的 inflights 最大数据量，它也影响着发送速度。比如即使有足够大的带宽，甚至有足够多的数据要发，但 CWND 不够，于是只能降低发送速度。这么几个东西会纠缠在一起，有很多文章在描述拥塞算法的时候为了方便可能会将这几个东西中的某几个混淆在一起描述，看的时候需要尽力心里有数。

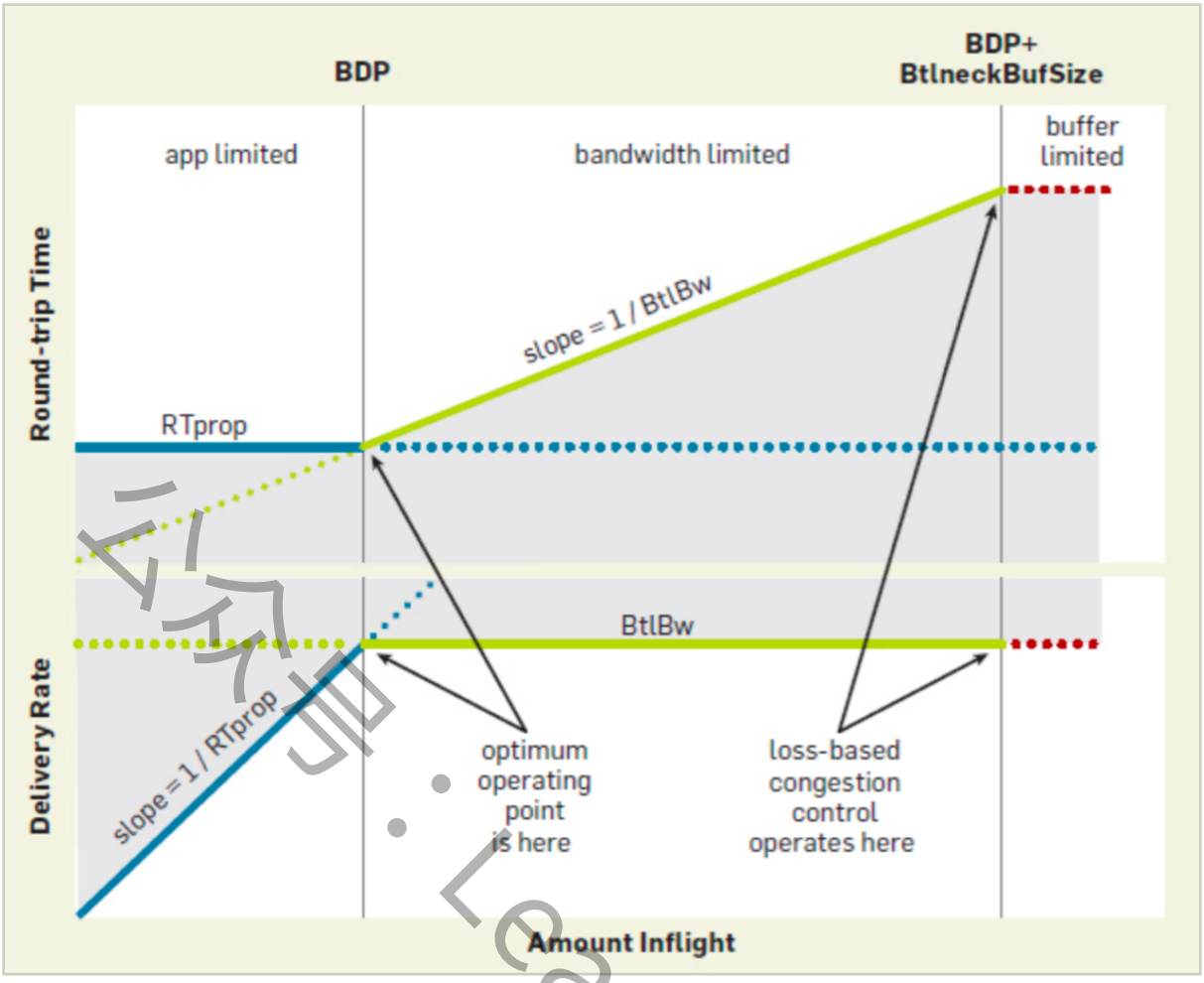
回到图蓝色的线表示 CWND 受制于 RTprop，绿色的线表示受制于 BltBw。灰色区域表示不可能出现的区域，至少会受到这两个限制中的一个影响。白色区域正常来说也不会出现，只是说理论上有可能出现，比如在链路上还有其它发送者一起在发送数据相互干扰等。灰色区域是无论如何不会出现。

先看上半部分，Inflights 和 Round-Trip Time 的关系，一开始 Inflights 数据量小，RTT 受制于链路固有的 RTprop 时间影响，即使链路上 Buffer 是空的 RTT 也至少是 RTprop。后来随着 Inflights 增大到达 BDP 之后，RTT 开始逐步增大，斜率是 `1/BtlBw`，因为 BDP 点右侧蓝色虚线就是 Buffer 大小，每个蓝点对应的纵轴点就是消耗完这么大 Buffer 需要的时间。`消耗 Buffer 的时间 / Buffer 大小` 就是 `1/BltBw`，因为消耗 Buffer 的时间乘以 BltBw 就是 Buffer 大小。感觉不好描述，反正就这么理解一下吧。主要是看到不可能到灰色部分，因为发送速率不可能比 BltBw 大。等到 Inflights 把 Buffer 占满后到达红色区域开始丢包。

下半部分，Inflights 比较小，随着 Inflights 增大 Delivery Rate 越大，因为此时还未达到带宽上限，发的数据越多到达率也就越高。等 Inflights 超过 BDP 后，Delivery Rate 受制于 BtlBw 大小不会继续增大，到达速度达到上限，Buffer 开始积累。当 Buffer 达到最大限度后，进入红色区域开始丢包。

之前基于丢包的拥塞算法实际上就是和链路 Buffer 一起配合控制 Inflights 数据量在 BDP 那条线后面与 BDP 线的距离。以前内存很贵，可能只比 BDP 大一点，基于丢包的算法延迟就不会很高，可能比最优 RTT 高一点。但后来 Buffer 越来越便宜，链路上 Buffer 就倾向于做的很大，此时基于丢包的拥塞算法就容易导致实际占用的 Buffer 很大，就容易出现 BufferBloat。

为了达到最优点，发送速率必须达到 BltBw，从而占满链路带宽，最高效的利用带宽；再有需要控制 Inflight 数据量不能超过链路的 `BDP = BtlBw * RTprop`，从而有最优的延迟。发送速率可以超过 BltBw，可以有队列暂时产生，但数据发送总量不能超 BDP，从而让平均下来延迟还是能在最优点附近。



来自[4]

TCP Vegas

Vegas 就如上面说的理想中算法一样，它会监控 RTT，在尝试增加发送速率时如果发现丢包或者 RTT 增加就降低发送速率，认为网络中出现拥塞。但它有这些问题：

- CWND 增长是线性的，跟 Reno 一样需要很久才能很好的利用网络传输速率；
- 最致命的是它不能很好的跟基于丢包的 Congestion Avoidance 共存，因为这些算法是让队列处在 State 2 和 3 之间，即尽可能让队列排队，也相当于尽可能的增大延迟，但 Vegas 是尽可能不排队，一发现排队就立即降低发送速率，所以 Vegas 和其它基于丢包算法共存时会逐步被挤出去；

BBR

估计 BtlBw 和 RTprop

延续前面说的，要把发送速率调整到跟 BDP 差不多大是最优的。因为网络环境会持续变化，所以需要持续监控 RTprop 和 BtlBw 的值。

RTprop 是链路固有传输延迟，我们无法直接监控到它，我们只能通过监控数据包的 RTT 来间接的得到 RTprop 的趋近值。RTT 和 RTprop 关系如下：

$$RTT_t = RTprop_t + \eta_t$$

η 是其它因素导致的 noise 延迟，包括接收端延迟 ack 等原因导致的延迟。因为 RTprop 是链路上的固有延迟，可以认为只有链路上选择的路径变化时候它才会变化，并认为它变化频率很低，变化间隔时间远大于 η。η 一定是正数，RTT 无论如何不可能低于 RTprop，所以可以在一个时间窗口 Wr 内（一般是在几十秒到几分钟之间）监控最小的 RTT 的最小值，认为就是近似为 RTprop。

$$\widehat{RTprop} = RTprop + \min(\eta_t) = \min(RTT_t) \quad \forall t \in [T - W_R, T]$$

BtlBw 也是靠 Ack 来监控。每收到一个 Ack 一方面是知道数据延迟 RTT 是多少，再有送达的数据量是多少。我们在一个短的时间窗口 deltaT 内通过 Ack 计算对面收到了多少数据，得到 `deliveryRate = deltaT 时间内送达数据量 / deltaT`，这个速率一定低于链路上瓶颈速率，bottleneck rate。因为我们计算 deliveryRate 时使用的数据送达量是精确值，是在 Ack 里数据接收方明确告诉我们的。而我们为了做带宽计算等待的 deltaT 时间一定大于实际时间，所以 `deltaT 时间内送达数据量 / deltaT` 计算得到的 deliveryRate 一定小于等于真实 deliveryRate，这个真实 deliveryRate 又一定小于等于链路物理 bottleneck rate。所以：

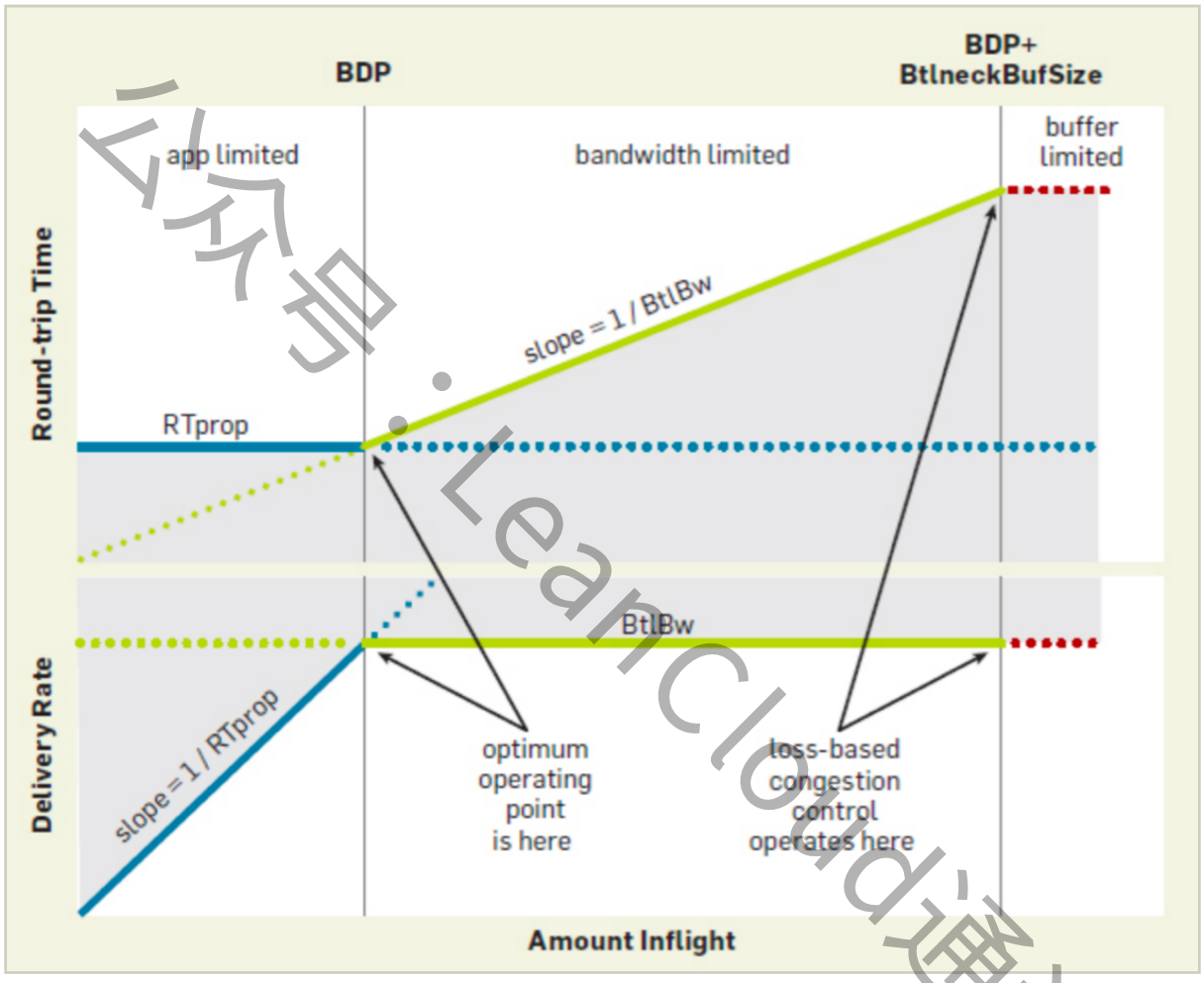
$$\widehat{BtlBw} = \max(deliveryRate_t) \quad \forall t \in [T - W_B, T]$$

这里时间窗口 Wb 一般是 10 个 RTT。

RTprop 和 BtlBw 是两个独立的量，RTprop 变化比如选择的链路变化时，bottleneck 可能是一样的 BltBw 不变；BtlBw 变化时候选择的链路可能没有变化，比如一个无线网络改变了发送速率，链路不变但带宽变大了。

从之前看过的下面这个图能知道，RTprop 只能在 BDP 线左侧被观测到，即发送数据比较少，inflight 数据量没有到 BDP 的时候；BtlBw 只能在 BDP 线右侧能被观察到，即 inflight 数据超过 BDP，延迟开始增大时被观测到。直观一点说就是链路上队列没排队时候，你知道链路延迟是多少，但不知道队列最大消费速度是多少，当队列开始出现排队后，你在发送端计算得到接收率不变了，RTT 延迟也升高了，知道最大接收率是多少，但又不知道链路上实际延迟是多大了，因为多了数据排队的时间。

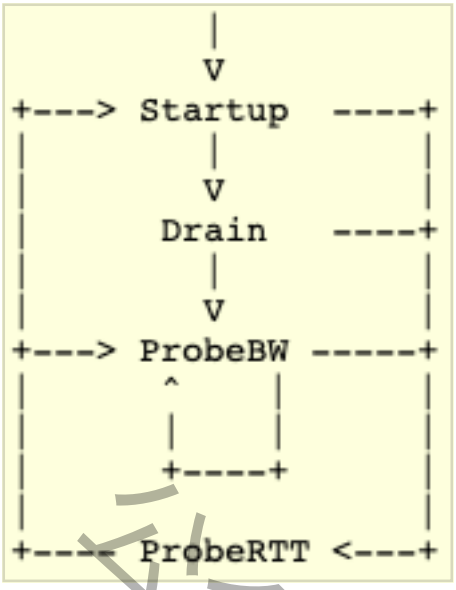
估计带宽时候还一个需要处理的，跟 RTT 不同，RTT 是只要有应用层数据发就能测得，就能更新估计值，但 BtlBw 的话必须应用层有足够数据发才能估计。所以下图可以看到有个 App Limited。BBR 会将因为 app 没足够数据发而导致测得的 BtlBw 过小的情况丢弃，只记录有足够数据发的情况下得到的 BtlBw。实际上链路中实际 BtlBw 是个 hard limit，就是无论我们怎么测都不可能超过链路上实际 BtlBw，所以只要在窗口内测得小的 BtlBw 都可以丢掉，就用测试周期内最大的 BtlBw。



来自[4]

BBR 的状态机

BBR 在控制拥塞时候会在一组状态之间进行切换，如下图：



来自[2]

先简单介绍一下下面再针对每个状态更细一些说。初始状态就是 Startup，BBR 因为不知道当前带宽到底是多少，也会有类似 Slow Start 的过程，逐步增加发送速度探测 BtlBw。等到探测到 BtlBw 后因为 Startup 阶段发了很多数据出去，inflights 很大，可能链路会有排队，所以进入 Drain 阶段去做清理，让链路上的队列清空；之后进入稳定期，会周期性的在 ProbeBW 和 ProbeRTT 两个状态之间进行切换，间歇的探测 BtlBw 和 RTprop。

BBR 稳定期工作机制

先记录稳定期再写 Startup 等。BBR 每收到一个 ACK，就会估计 RTprop 和 BtlBW (deliveryRate)，尽力保证 inflight 数据量跟估计得到的 BDP 差不多。BBR 下 inflights 数据量由内部的 target_cwnd 控制，而 target_cwnd 是个比 BDP 稍微大一点点的量。

假设链路上 RTprop 和 deliveryRate 一直保持不变，BBR 处在稳定状态一直发送数据，它保证 inflights 的数据量不会超估计的 BDP 很多。此时链路上 Bottleneck 在发送端，由发送端控制发送速度。如果链路上带宽提高了，因为 Bottleneck 在发送端，发送端会感知不到带宽变化。所以 BBR 需要周期性的提高发送速率将 Bottleneck 从发送端移到链路上去探测链路上的带宽，这就是 ProbeBW 状态的来源。名字也可以看出它含义是探测带宽，探测的是链路上 Bottleneck 的带宽。

ProbeBW 状态下先开始增周期，即提高发送率到稳定期的 1.25 倍，直到出现丢包或 inflights 数据量达到 1.25 倍 BDP 为止。观察延迟是否升高，如果延迟升高且 deliveryRate 不变，说明链路上带宽没有变化且产生了队列堆积；接下来会进入减周期，降低发送率到稳定期的 0.75 倍，等待一个 RTprop 或 inflights 数据量低于 BDP 为止，用以让链路上在增周期出现堆积的队列清空。之后再保持 inflights 等于 BDP 稳定数个 RTprop 后再次开始增周期。

之前提到 BBR 每次收到 ACK 会尝试更新 RTprop，而 RTprop 取的是窗口期内最低的 RTprop。如果 BBR 运行了很长时间一直没有更新 RTprop，即很长一段时间都没有比当前使用的 RTprop 更低的 RTprop 时，BBR 会进入 ProbeRTT 状态，用于探测 RTprop。比如链路上带宽出现减少，链路上出现堆积，保持发送速度或继续进行 ProbeBW 的话会让链路上堆积更加严重，RTT 上升。所以 RTprop 一直不会被更新。

ProbeRTT 下 BBR 将 CWND 降到很低的值，典型的是 4 个 MSS，持续一段至少 200ms 或一个 RTprop。这么一来 Inflights 数据量会突降。再判断链路是否处在 full_pipe 状态，是的话则进入 ProbeBW，不是则进入 Startup。

full_pipe 判断主要是靠最近的增周期中，发送率提高后 deliveryRate 是否有大幅度增加，有相应幅度的增加说明链路可能还没有满载，我们加快发送速率还是能发的出去，没增加则表示链路是满载的，发送速度加快但是接收速率没跟上。有个 Linux 内的 BBR 实现，可以看：tcp_bbr.c source code linux/net/ipv4/tcp_bbr.c - Woboq Code Browser

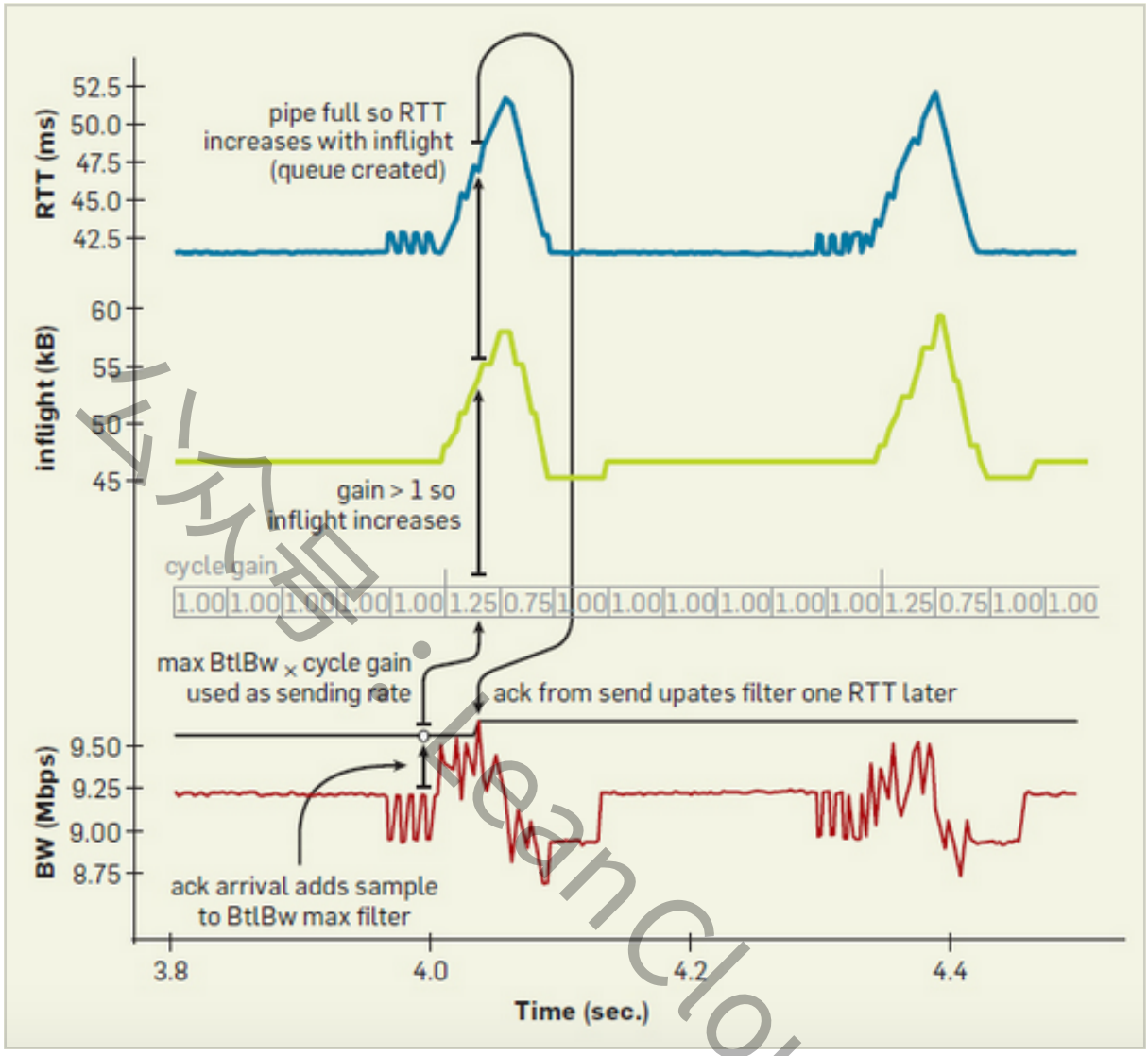
之前提到了 RTprop 和 BtlBw 两个量是测准一个另一个就测不准，稳定期测量这两个值的任务大多数时候都由 ProbeBW 独自完成，在增周期提高发送带宽发多数据到链路探测 BtlBW，在减周期减小发送带宽减少发送速率从而减小 Inflights，看估计的 RTprop 是否降低，降低了则说明测到了最新的 RTprop。如果很长时间都没有更低的 RTprop 出现，可能链路发生了切换，这时候才需要切换到 ProbeRTT 去探测最新的 RTprop。从 ProbeRTT 的机制能看到它对性能是有影响的，所以 BBR 是尽力减少 ProbeRTT 时间占比，大部分时间都在 ProbeBW 状态。

整体过程如下图，在一个 10Mbps 带宽，延迟 40ms 的网络下取了 700ms 时长的数据。图中纵轴有 RTT，inflight，和 Bandwidth (BW)。带宽部分有两个值，红色的是发送方计算出来的 Delivery Rate，紧挨着红色线的黑线是当前估计出来的 BtlBw。可以看到 Delivery Rate 波动但估计出来的 BtlBw 变动不大，因为是每次有估计出 Delivery Rate 后将这个值放入估计 BtlBw 的滑动统计窗口内，取滑动窗口内的最大值为 BtlBw，也即那个黑色线，所以红色线变动后，黑色线不会立即变动。黑色线上面的灰色框是 cycle gain 用于控制发送速度，`cycle gain * 估计的 BtlBw` 就是发送方当前需要的发送速度。每个周期通过控制 cycle gain 来定期的提高发送速度，以探测当前链路上 BtlBw 是否提高。

对于黑色的圈我们从起点开始说，起点是红线，即 Delivery Rate 计算出来后，放入 max BtlBw 滑动窗口估算 BtlBw 是多少。之后 1.25 这个 cycle_gain 时用之前估算的 BtlBw 乘以 1.25 作为发送速度发消息，增周期发消息多了以后链路 Inflights 增大，RTT 增大，发出的消息看到是在 1.00 这个 cycle_gain 才被 Ack，计入 Delivery Rate，并让 max BtlBw 稍微增加了一点点，看到红线峰值比红线上面的细黑线高了一点，于是将细黑线也向上推了一点点。

图中 RTT，Inflights，cycle_gain，Delivery Rate 大致都是对齐的，可以看到增周期时候

Delivery Rate ， Inflights 和 RTT 增加，减周期是减少。



来自[4]

下图是带宽从 10Mbps，40ms 延迟提到到 20Mbps 又降回 10Mbps 的过程。带宽提高后看到在 ProbeBW 的增周期， BBR 发现提高发送速度后 RTT 没有变化，且计算出来的 deliveryRate 有升高，更新 BtlBw 到新值，稳定期发送速度比原来高了 25%。在接下来三个周期内，每一次发送速率提高 25% 后延迟都没有变化，且 deliveryRate 得到提高，最终在第四个探测周期重新出现蓝色小三角，即链路上队列有排队后说明链路进入 full_pipe 状态开始稳定发送速率。

下半部分是带宽从 20Mbps 降低到 10Mbps，BBR 内因为维护了 BtlBW max filter 即从一个窗口期内采样得到的 BtlBw 值中取最大值作为当前链路的 BtlBw。所以即使带宽出现突降，因为 20Mbps 的 BtlBw 还在 max filter 内缓存着，这段时间 BBR 依然认为 BtlBw 是 20Mbps，会按照原来的发送速度继续发数据。于是带宽突降后延迟和 inflight 数据量大幅度增加。 但 inflights 数据量最大不能超过 BBR 内的 target_cwnd，其值等于 `cwnd_gain * 估计的 BDP`，BBR 会始终控制发送速率保持 inflights 在 target_cwnd 内，cwnd_gain 比 1 大，但不会大很

多，所以带宽突然变小后 inflights 不会无限增加，并且会维持一个固定值，在图中表现为 40s ~ 42s 之间的一条水平线。这个期间即使处在 ProbeBW 阶段也无法执行增周期按 1.25 倍速率发数据，因为 target_cwnd 是满的，必须遵从它的限制，它限制了发送端不能让 inflights 数据量比它大。Inflights 和 RTT 能是一条水平线说明链路上 Buffer 比较大，能承载 target_cwnd 下的数据量且不出现丢包。

补充一下 BBR 里有两个听起来很像的东西，pacing_gain 和 cwnd_gain。pacing_gain 用来在 ProbeBW 内周期性的控制发送速率，发送速率等于估计的 `BtlBw * pacing_gain`。而 cwnd_gain 用于控制 target_cwnd，限制 inflights 数据量。

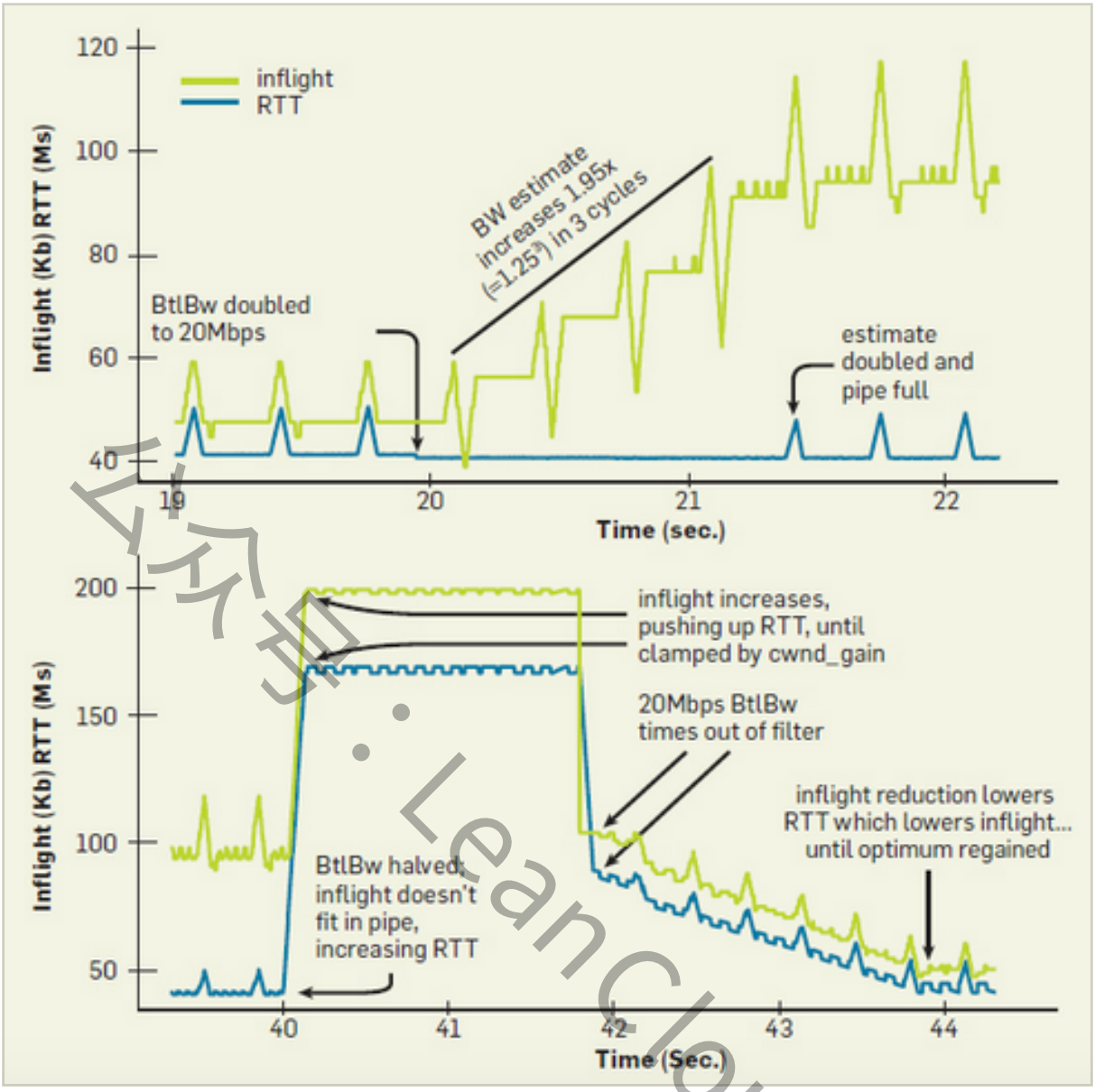
在 42s 开始，之前 20Mbps 的 BtlBw 估计值过期了，从 BtlBw max filter 的采样窗口中滑出，根据过去一段时间 Ack 计算出来的到达重新估算 BtlBw，根据这个重新估算的 BtlBw 调整发送速度，所以发送速率大幅度下滑。因为当前的 Inflights 数量远大于当前的 target_cwnd 也即 `cwnd_gain * 新估计的 BtlBw`，所以发送方会完全不发数据等待 Inflights 下降，在图上表现出 Inflights 突然掉下来。当 Inflights 小于当前 target_cwnd 后，ProbeBW 的周期特性又开始显现，一个一个的小三角开始出来了。最终发送速率重新回到稳态。

看到这里时候我开始一直有个疑问，在 42s 到 44s 期间，队列内一直有堆积，一个周期发送速度是 1.25，一个周期发送速度是 0.75，岂不是刚好把发多少数据消费掉，但队列内堆积的 inflights 不是依然保持没有被消费掉吗？不该是下降趋势。

后来知道，ProbeBW 期间一个周期大致上是一个 RTT 时间，但发送速度 1.25 的周期和发送速度 0.75 的周期并不是严格等长的。1.25 周期时长是 inflights 数量到达 1.25 倍估计的 BDP 或有数据丢包。0.75 周期长度是一个完整的估计的 RTprop，或者 inflights 低于估计的 BDP。发送者能感知到 inflights 低于 BDP 的时候实际 inflights 一定低很多了，所以 BBR 的 inflights 曲线都是像心跳一样，上面一个三角下面一个三角，而不是只有上面的三角。在 19 ~ 20s 的时候，上三角比下三角大。20 ~ 21s 因为带宽变大了，但发送带宽只是缓慢增加上去，所以下三角比上三角大很多，等到稳态以后又变成上三角大于下三角。42s 以后因为每个减周期都要等到 Inflights 低于估计的 BDP，所以绿线一路向下。

对于蓝色的线我们知道链路上延迟是固有时间，所以它最低点是一条直线，增周期时延迟只会有上三角，没下三角。

注意下图没有 ProbeRTT 出现。我理解是因为 RTprop 还未超时就被更小的值更新了。



来自[4]

BBR 的 startup

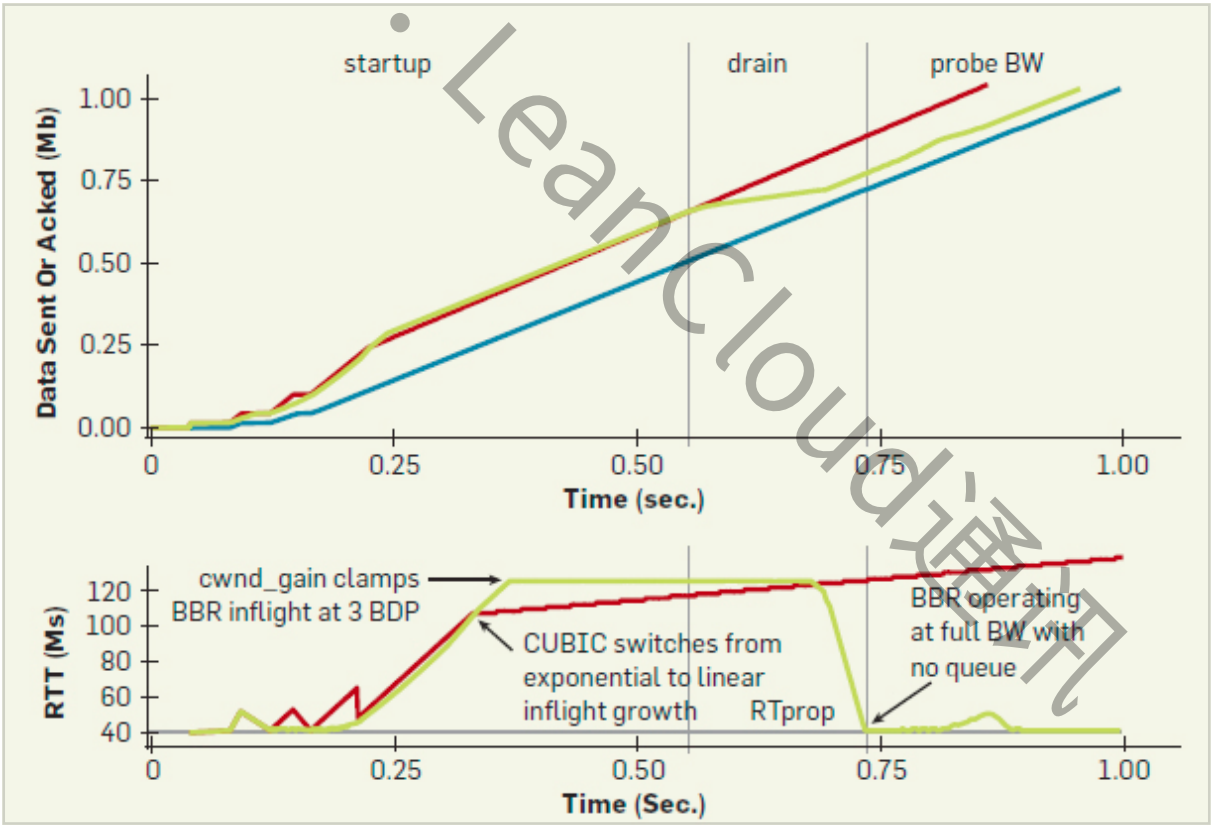
链路上带宽跨度很大，从几个 bps 到上百 Gbps，所以 BBR 一开始也会以指数形式增大 BtlBw，每一个 RTT 下发送速度都增大 `2/ln(2)` 约 2.89 倍，从而在 `O(log(BDP))` 个 RTT 内找到链路的 BtlBw，log 的底是 2。BBR 在发现提高发送速度但 deliveryRate 提高很小的时候标记 full_pipe，开始进入 Drain 阶段，将排队的数据包都消费完。BBR 能保证排队的数据最多为实际 BDP 的 1.89 倍。BBR 下并没有 ssthresh，即 CUBIC 那样增加到某个配置值后开始进入线性增加 CWND 阶段。

Drain 阶段就是把发送速率调整为 Startup 阶段的倒数。比如 Startup 阶段发送速度是 2.89，那 Drain 阶段发送速度是 `1/ 2.89`。BBR 会计算 inflights 数据包量，当与估计的 BDP 差不多的时候，BBR 进入 ProbeBW 状态，后续就在 ProbeBW 和 ProbeRTT 之间切换。

下图是在 10Mbps，40ms 延迟网络下的 Startup 阶段的图，绿色是发送方发送数据量，蓝色是接收方收到的数据量。红色是 CUBIC 在同样环境的发送数据量，作为对比。下图绿色线的斜率就是发送速度，同一时间点绿色线上的点和蓝色线上的点的差值就是那个时刻 inflights 数据量。

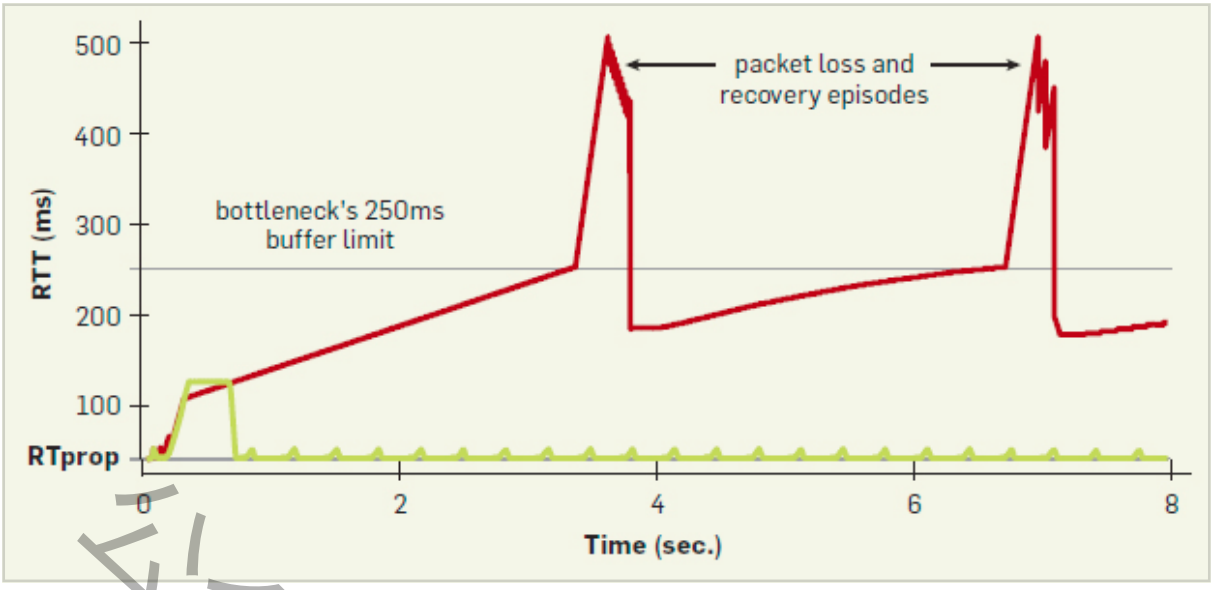
看到 BBR 在 0.25s 之前是曲线，10Mbps 40ms 延迟的网络下 BBR 允许的 BDP 为 0.04Mb，0.25 时间点上绿色和蓝色线差值大约 0.1Mb，大致是 0.04Mb 的 2.89 倍。即 BBR 在一开始指数的快速提升发送速度，但快到了 0.25s 的时候，RTT 开始升高，inflights 提高后 deliveryRate 并没有相应提高，BBR 开始维持在一个速度等待几个 RTT 周期以确认链路确实承载不了当前的发送速度，所以 0.25s ~ drain 之前绿线斜率不变。到 Drain 后 RTT 迅速下降，Acked 数据量图的斜率也降低很多。在找到 RTprop 之后进入 ProbeBW 状态。

红线的话就是看到 CUBIC 更早的切换到线性增长，但之后会逐步增加发送包数量直到丢包。但下图上半部分看红线看不太出来发送率在增加，只是能隐约的感觉到 0.25 时间点时红线和蓝线的间隔似乎小于 0.75 时间点时他俩的间隔。



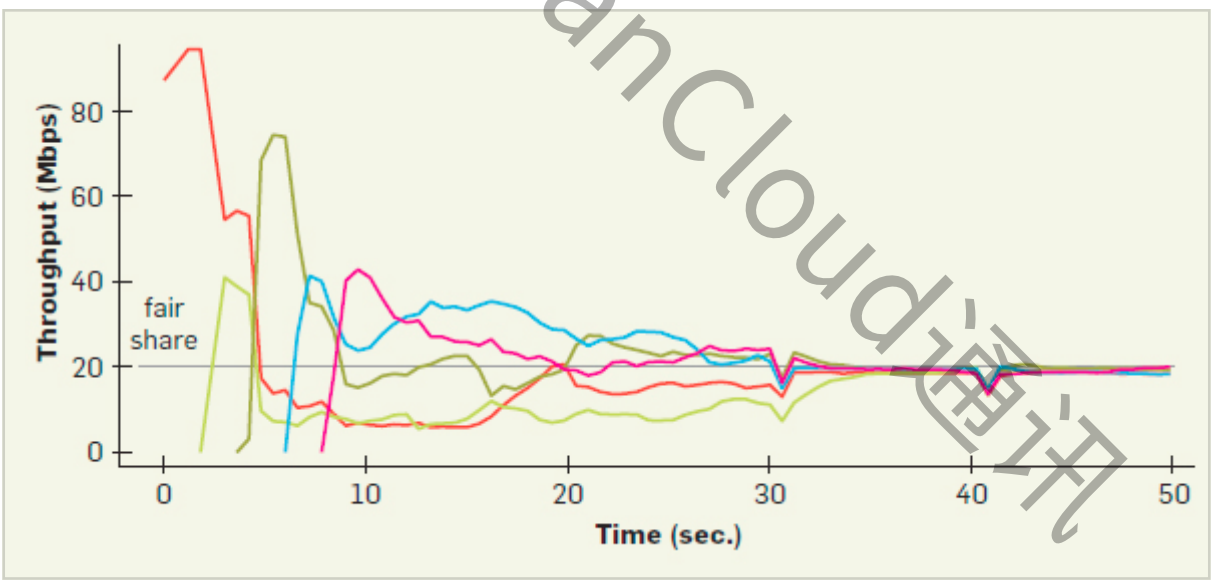
来自[4]

下面是 BBR 和 Cubic 在延迟时间上的对比。BBR 开始延迟大但随即恢复。CUBIC 一直增长下去直到丢包，再减小再增长。



来自[4]

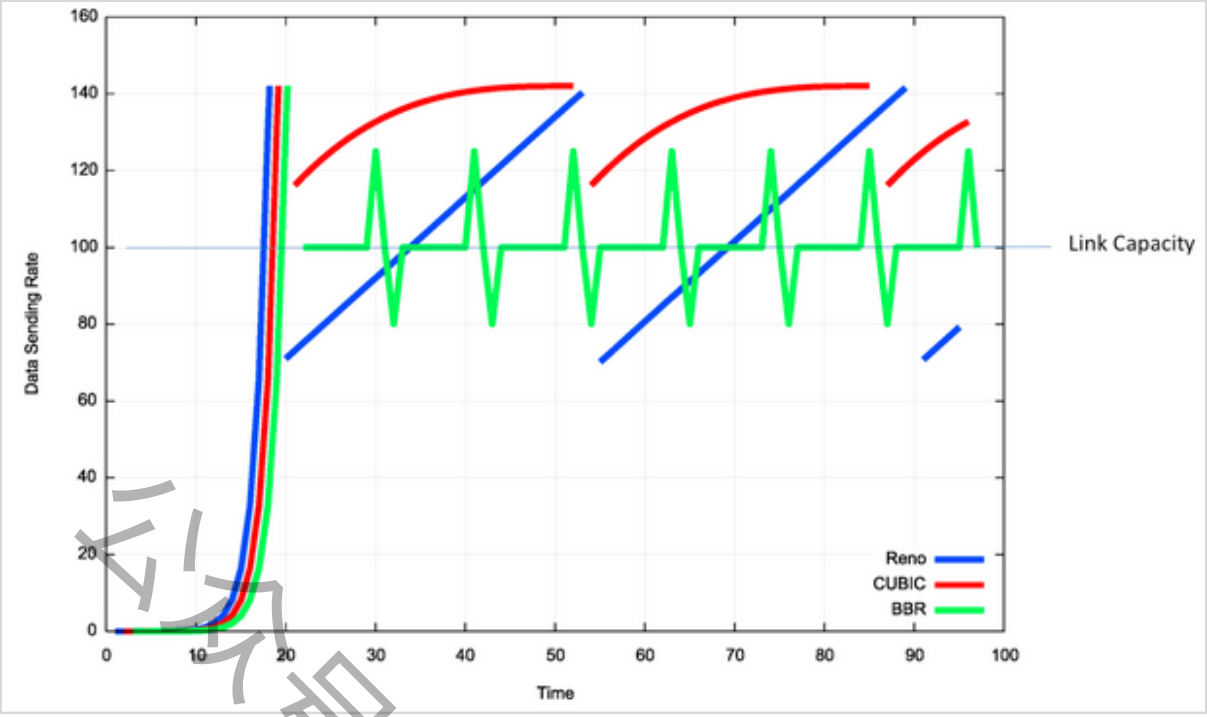
当多个 BBR 流在同一个链路时如下图。基本上就是靠 BtlBw max filter 内 BtlBw 超时过期以及 ProbeRTT 逐步让每个 BBR 流都找到自己合适的份额。在一开始只有红色的线，之后绿的来了，红色线继续按原来速度发数据链路一定会拥塞，并且因为有两个流了红线的 deliveryRate 一定会下降，BtlBw 超时后红线降低发送速率。大致上就是这么相互作用，每来一个流需要调整一下直到稳定。



来自[4]

BBR 跟 CUBIC 等基于丢包的算法共存时会有问题吗?

列一下 Reno，CUBIC，BBR 三个在时间和发送速率上的图做对比。



来自[1]

之前提到 TCP Vegas，Vegas 会被基于丢包的算法挤掉份额，不能跟这些算法共存，所以无法流行起来。对于 BBR 的话，它抢占份额的方式主要靠 ProbeBW 期间按 1.25 倍估计值发送数据，会给链路带去压力，为自己抢占生存空间；再有就是 Startup 阶段，BBR 相对 CUBIC 来说会为链路带去更多数据去抢占份额，这个只在初始阶段有用，但如果 BBR 上的数据如果能持续发送，初始期占的份额可能就能一直保持下去。

对 BBR 和基于丢包的 Congestion Avoidance 的分析有两种不同的结果：

- BBR 在 ProbeBW 阶段虽然会将发送速度提高到 1.25 倍但毕竟持续时间短，一半以上的时间还是以估计速率在发数据。如果不能给链路带去足够压力，BBR 会被其它基于丢包的算法挤掉，因为 BBR 会认为延迟升高了，需要降低自己的发送速度，这么逐步降低；
- BBR 可能错误的估计链路延迟，比如估算链路是否有排队主要通过 RTT 有没增加完成，但 RTT 即使不增加可能链路也有排队，比如链路上队列处在基本满的状态，BBR 认为此时的 RTT 就是链路 RTT 最小值，于是保持速度发数据，它还容忍丢包，但基于丢包的算法在此时就会降速。

据说 Google 给出来说 BBR 可以很好的跟 CUBIC 等算法共存，但参考 [1] 的Sharing 一节在两种场景测试发现 BBR 比 CUBIC 占用更多资源或者说份额。

BBR 如何处理丢包

在网上搜很多文章都说 BBR 根本不管丢包，完全基于自己的周期即 ProbeBW 内的周期去计算发送率，CWND 来发数据包，所以 BBR 根本无视丢包。但实际上 BBR 作者写的 BBR 介绍里

明确说了 BBR 是会管丢包情况的，来自 BBR: Congestion-Based Congestion Control - ACM Queue：

> The network path and traffic traveling over it can make sudden dramatic changes. To adapt to these smoothly and robustly, and reduce packet losses in such cases, BBR uses a number of strategies to implement the core model. First, BBR treats cwnd_gain×BDP as a target that the current cwnd approaches cautiously from below, increasing cwnd by no more than the amount of data acknowledged at any time. Second, upon a retransmission timeout, meaning the sender thinks all in-flight packets are lost, BBR conservatively reduces cwnd to one packet and sends a single packet (just like loss-based congestion-control algorithms such as CUBIC). Finally, when the sender detects packet loss but there are still packets in flight, on the first round of the loss-repair process BBR temporarily reduces the sending rate to match the current delivery rate; on second and later rounds of loss repair it ensures the sending rate never exceeds twice the current delivery rate. This significantly reduces transient losses when BBR encounters policers or competes with other flows on a BDP-scale buffer.

更长的在这里：Modulating cwnd in Loss Recovery

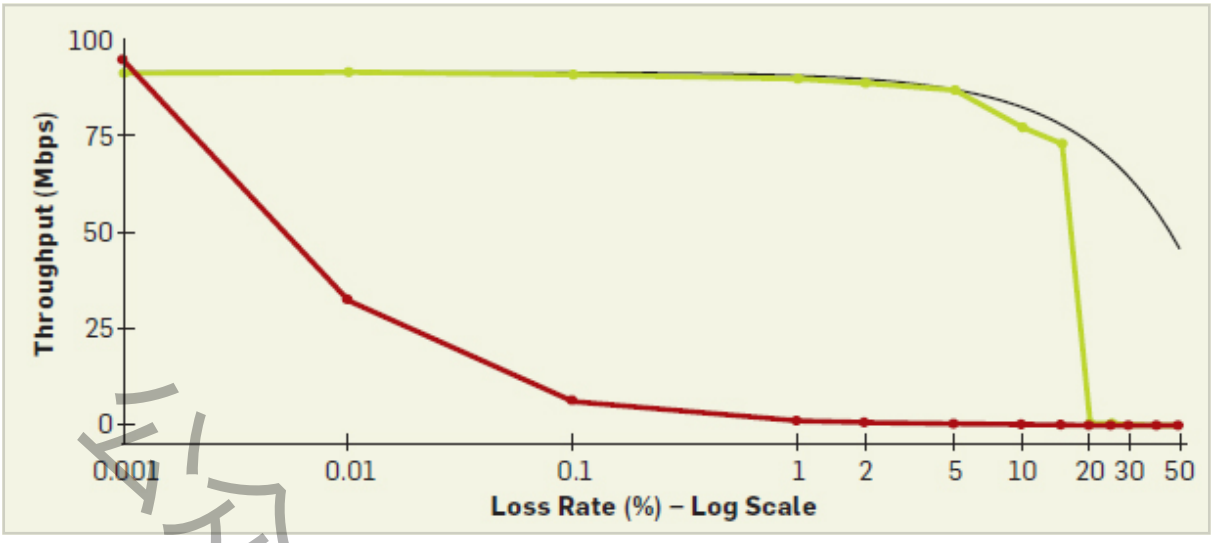
细节很多，我看的很晕。在看 BBR 处理丢包策略前，至少得先回忆一下 TCP 的丢包处理机制，一个是 Retransmission Timeout (RTO)，一个是 Fast Retransmission，这两个概念就不记录了，可以翻一下 Wiki。大致上是说如果遇到重传 RTO，发送者会认为所有 Inflights 数据都丢了，CWND 会降为 1 MSS，行为和 CUBIC 类似。在之后持续收到 ACK 的话，会逐步增加 CWND 值。每次 ACK 时候一方面会更新各种统计值，再有会更新 CWND 值，CWND 的增幅就是每次收到 ACK 时候 ACK 的数据量。每个 ACK 都会尝试去更新，直到 CWND 达到 target_cwnd 即当前估计的 BDP 和 cwnd_gain 的乘积。

如果是 Fast Retransmission，发送者则认为链路遇到丢包但仍然存在有效 Inflights 数据。 这时首先将 CWND 降低到当前与当前 deliveryRate 相匹配的值，之后一段时间发送速率不能超过 deliveryRate 的两倍。直到退出 Fast Retransmission 阶段后，CWND 恢复为丢包出现前记录的 CWND 值。就是说在 RTO 或 Fast Retransmission 前会记录当时 CWND，再出 Fast Retransmission 或 RTO 后恢复到记录的 CWND。

总之说 BBR 处理丢包的时候需要区分 RTO 和 Fast Retransmission 两个场景，RTO 下与 CUBIC 差不多；只有 Fast Retransmission 下，发送速率还能保持在较高位置，但也是受影响的。

下面是 CUBIC （红色）和 BRR （绿色）在面对丢包的时候吞吐量表现。CUBIC 在丢包 1% 的时候基本就无法工作了，BRR 能扛到 20% 的丢包率。黑线是理想状况下的吞吐量，因为有丢

包存在，所以吞吐量随着丢包率升高而递减。



来自[4]

对于右侧绿线是个断崖式有这么一些原因：

- 丢包率高了以后一个是会引起 Fast Retransmission，会对吞吐量有影响，另一个是可能出现 RTO，出现 RTO 后 CWND 立即降到 1 跟 CUBIC 表现就一致了，吞吐量会大幅度下滑；
- 丢包率高了之后会严重影响实际 Delivery Rate 的计算，让计算得到的值比实际值小，进而让 BBR 错误估计当前 BtlBw，减小发送速率。因为丢包率固定不变，减小发送速率后 Delivery Rate 可能会进一步减小，最终降为 0；
- 最后，这个图本身就是指数的，即使是理想状态也会是个断崖；

对于为什么是丢包率接近 ProbeBW 增周期增益 (1.25) 时会出现吞吐量大幅度下滑我理解是这样的：

在估计 BtlBw 是是取过去一段时间内的最大值，比如丢包率是 10%，如果没有增周期发送者通过 Delivery Rate 计算得到的带宽比实际 BtlBw 会低 10%，之后会调整发送速度变成 90% BtlBw，因为依然有 10% 的丢包率，在 90% BtlBw 从估计带宽用的滑动窗口过期移除后，发送速率会降为 `90% * 90% BtlBw = 81% BtlBw`，最终这么一步步下降到 0，这是没有增周期的场景。

但是因为有增周期的存在，假设当前估算的带宽为 BtlBw，普通周期发送速率就是 BtlBw，增周期是 1.25 BtlBw，减周期是 0.75 BtlBw。当丢包率是 10%，普通周期时计算出来的 Delivery Rate 是 90% BtlBw，增周期计算出来的是 `1.25 * 90% * BtlBw = 1.125 BtlBw`，但实际上 BtlBw 是链路带宽不可能超越 BtlBw，所以增周期 Delivery Rate 还是 BtlBw，减周期是 `0.75 * 90% * BtlBw = 0.675 BtlBw`，这么一来根据过去一段时间 Delivery Rate 最大值估算带宽时得到的带宽还是 BtlBw，所以 10% 丢包率的时候吞吐量受丢包的影响不大。

当丢包率增长到 20% 的时候，根据上面的算法，增周期计算出来的 Delivery Rate 就马马虎虎刚好是 1 倍 BtlBw 了，因为 `1.25 * 80% * BtlBw = 1 BtlBw`。只要有风吹草动让计算的 Delivery Rate 低于最初的 BtlBw，那一步步的计算的 Delivery Rate 就会和实际带宽偏离越来越大，最终降到 0。也就是说增周期的增益和丢包率的乘积能大于 1，就能保持发送速度，小于 1 则会断崖式下滑，大于 1 是不可能的。

所以 BBR 对 CUBIC 在丢包方面的优势是说，CUBIC 是算法结构上导致丢包一定会让性能大幅度下滑。而 BBR 是在一定程度上能通过配置去容忍丢包。比如增周期增益提升到 1.5，抗丢包能力一定会大幅度提升，但性能受重传数据影响会越来越大。

采样的时候如果应用层没数据发送怎么办

BBR 数据都是基于采样的，比如 ProbeBW 时候要按照 1.25 倍速率发数据，如果应用层没有那么多数据要发怎么办?

BBR 采样时候会标记数据是不是 app limited。如果是因为 app 原因导致没有足够数据可发从而让 deliveryRate 降低了，收到 ACK 的时候如果这个 deliveryRate 没有高于现在估计的 BtlBw 最大值，则会丢弃这个 deliveryRate 不放入 BtlBw max filter。高于当前 BtlBw 最大值能放入 max filter 的原因是链路上的实际 BtlBw 是个 Hard Limit，不管怎样都不可能超过，所以如果 app limited 状态下 deliveryRate 都高于当前估算的 BtlBw，则说明估计值太小了。

既然 BtlBw max filter 是直接取采样得到的 deliveryRate 的最大值，那 app limited 时候即使测得的 deliveryRate 不高于现在 BtlBw 也直接放入 max filter 不就好了，反正因为它小也不会被取到?

目前来看虽然理论上说每次 BBR 都取的是 deliveryRate 的最大值就行了，就是 BtlBw，但实际上 BtlBw max filter 并不是简单的取最大值就完了，里面还有点基于时间的函数对窗口期内的所有 deliveryRate 做了一些计算才得到的 BtlBw。可以看参考 [2]。所以 app limited 的值如果没有超过 BtlBw 则直接丢弃。

参考:

1. https://blog.apnic.net/2017/05/09/bbr-new-kid-tcp-block/
2. BBR Congestion Control
3. Cubic
4. https://cacm.acm.org/magazines/2017/2/212428-bbr-congestion-based-congestion-control/fulltext
5. https://queue.acm.org/detail.cfm?id=3022184
6. https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/IFIP-Networking-2018-

TCP-BBR.pdf
7. AusNOG 2019: TCP and BBR
8. A bit about TCP BBR - http://blog.cerowrt.org/
9. tcp_bbr.c source code linux/net/ipv4/tcp_bbr.c - Woboq Code Browser
10. RFC 6937 - Proportional Rate Reduction for TCP