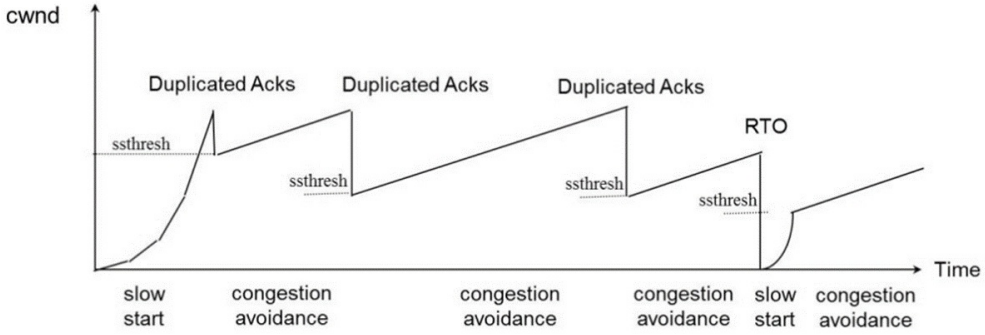
TCP BBR拥塞控制算法解析

1.背景

1.1TCP 基于丢包的拥塞控制

TCP拥塞控制将丢包视为网络出现拥塞的信号，以下为其四个主要过程：



引用自参考资料[5]

（1）慢启动阶段（slow start）

当建立新的TCP连接时，拥塞窗口（congestion window,cwnd）初始化为一个数据包大小。源端按cwnd大小发送数据，每收到一个ACK确认，cwnd就增加一个数据包发送量，这样cwnd就随着回路响应时间（Round Trip Time,RTT）的增加呈指数增长。

（2）拥塞避免阶段(congestion avoidance)

如果TCP源端发现超时或收到3个相同ACK时，即认为网络发生了拥塞。此时就进入拥塞避免阶段。慢启动阈值（ssthresh）被设置为当前拥塞窗口大小的一半；如果超时，拥塞窗口被置1。当cwnd>ssthresh，TCP就执行拥塞避免算法，此时，cwnd在每次收到一个ACK时只增加1/cwnd个数据包。这样，在一个RTT内，cwnd将增加1，所以在拥塞避免阶段，cwnd线性增长。

（3）快速重传阶段(Fast Retransmit)

快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认ACK就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器RTO超时。

（4）快速恢复阶段(Fast Recovery)

与快速重传配合使用，有以下两个要点:

①当发送方连续收到三个重复确认时，就执行“乘法减小”算法，即把ssthresh门限减半。但是接下去并不执行慢开始算法。

②考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。

2.动机

2.1 基于丢包的拥塞控制算法的两大缺陷

（1）不能区分是拥塞导致的丢包还是错误丢包

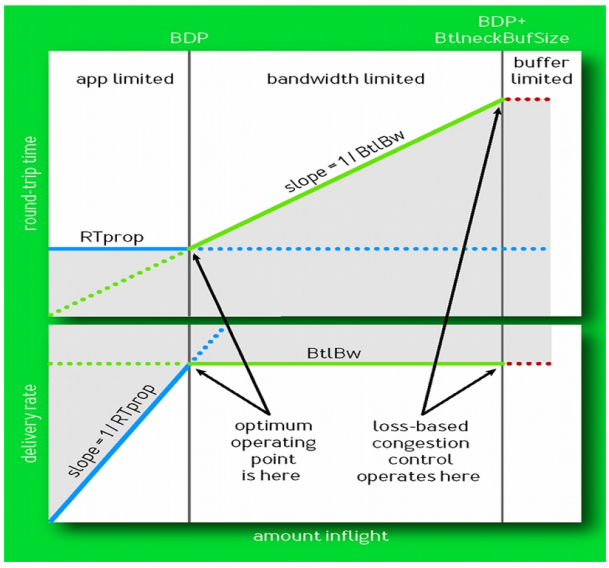
基于丢包的拥塞控制方法把数据包的丢失解释为网络发生了拥塞，而假定链路错误造成的分组丢失是忽略不计的，这种情况是基于当时V. Jacobson的观察，认为链路错误的几率太低从而可以忽略，然而在高速网络中，这种假设是不成立的，当数据传输速率比较高时，链路错误是不能忽略的。在无线网络中，链路的误码率更高，因此，如果笼统地认为分组丢失就是拥塞所引起的，从而降低一半的速率，是对网络资源的极大浪费。

（2）引起缓冲区膨胀

我们会在网络中设置一些缓冲区，用于吸收网络中的流量波动，在连接的开始阶段，基于丢包的拥塞控制方法倾向于填满缓冲区。当瓶颈链路的缓冲区很大时，需要很长时间才能将缓冲区中的数据包排空，造成很大的网络延时，这种情况称之为缓冲区膨胀。在一个先进先出队列管理方式的网络中，过大的buffer以及过长的等待队列，在某些情况下不仅不能提高系统的吞吐量甚至可能导致系统的吞吐量近乎于零。并且当所有缓冲区都被填满时，会出现丢包。

2.2 网络工作的最优点是可达到的

FIGURE 1: **DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT**



引用自参考资料[1]

当网络中数据包不多，还没有填满瓶颈链路的管道时，随着投递率的增加，往返时延不发生变化。当数据包刚好填满管道，达到网络工作的最优点（满足最大带宽BtlBw和最小时延RTprop），定义带宽时延积BDP=BtlBw × RTprop，则在最优点网络中的数据包数量=BDP。继续增加网络中的数据包，超出BDP的数据包会占用buffer,达到瓶颈带宽的网络的投递率不再发射变化，RTT会增加。继续增加数据包，

buffer会被填满从而发生丢包。

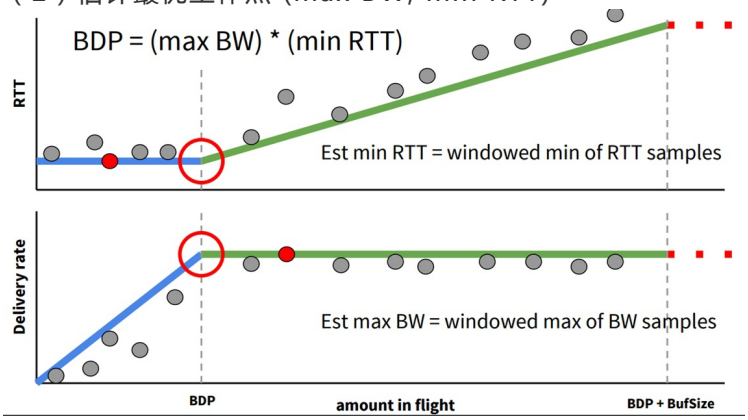
故在BDP线的右侧，网络拥塞持续作用。过去基于丢包的拥塞控制算法工作在bandwidth-limited区域的右侧边界，将瓶颈链路管道填满后继续填充buffer，直到buffer填满发生丢包，拥塞控制算法发现丢包，将发送窗口减半后再线性增加。过去存储器昂贵，buffer的容量只比BDP稍大，增加的时延不明显， 随着内存价格的下降导致buffer容量远大于BDP，增加的时延很大。

在1979年，已经由Leonard Kleinrock证明，在bandwidth-limited区域的左侧边界，对于单个连接或对于整个网络而言，都是网络工作的最优点，即伴随有最大投递率、最小时延和丢包。

3.基本观点

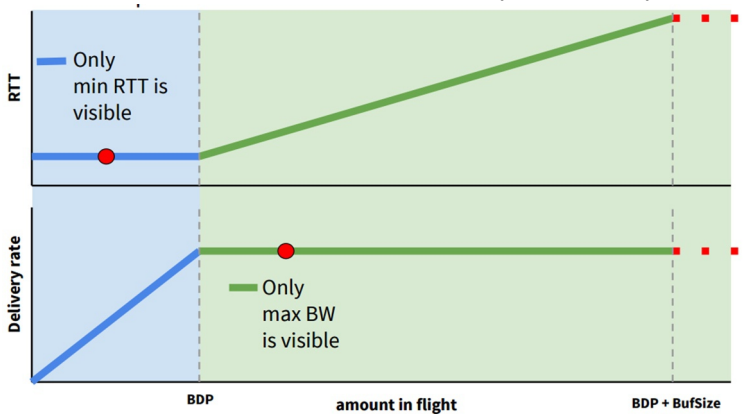
（1）不考虑丢包

（2）估计最优工作点 (max BW, min RTT)



引用自参考资料[2]

上图红色圆圈所示即为网络工作的最优点，此时数据包的投递率=BtlBW(瓶颈链路带宽)，保证了瓶颈链路被100%利用；在途数据包总数=BDP(时延带宽积)，保证未占用buffer。



引用自参考资料[2]

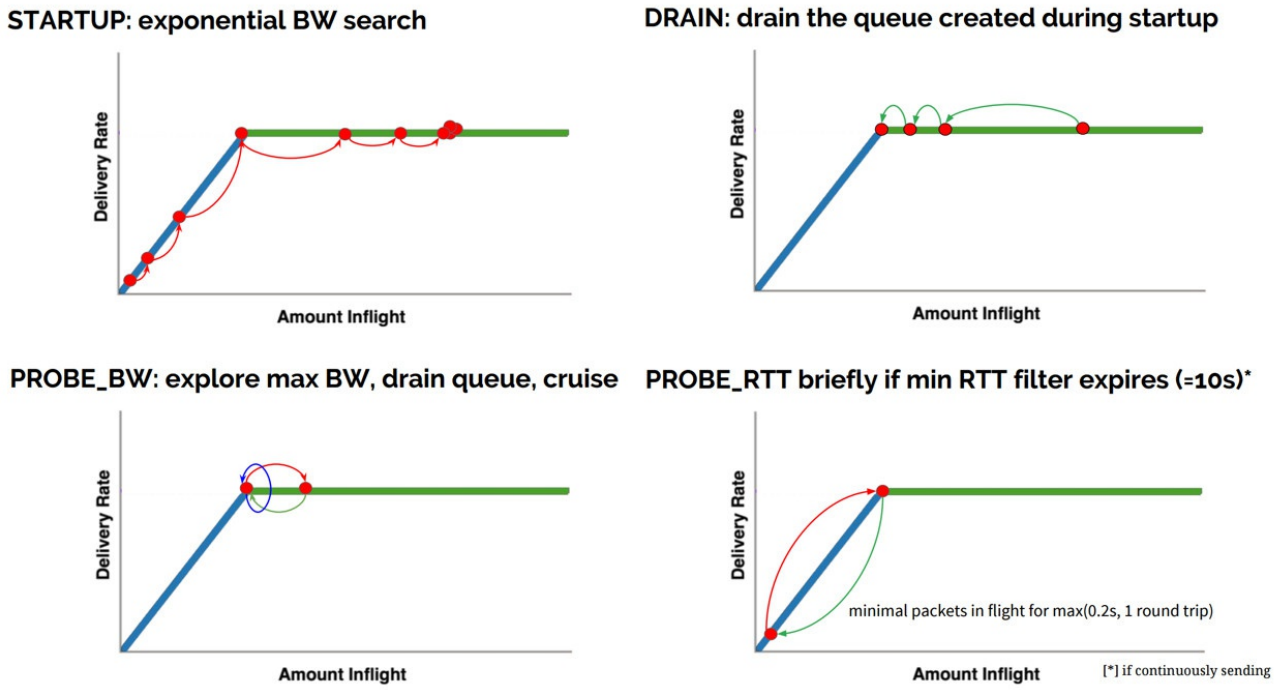
然而max BW和min RTT不能被同时测得。要测量最大带宽，就要把瓶颈链路填满，此时buffer中有一定量的数据包，延迟较高。要测量最低延迟，就要保证buffer为 空，网络中数据包越少越好，但此时带宽较低。

BBR的解决办法是：

交替测量带宽和延迟，用一段时间内的带宽极大值和延迟极小值作为估计值。

4.设计细节

4.1BBR的四个状态（启动、排空、带宽探测、时延探测）



引用自参考资料[2]

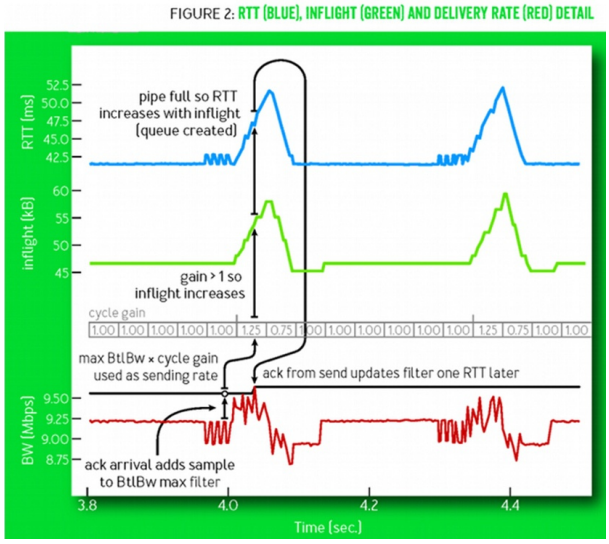
（1）当连接建立时，BBR采用类似标准TCP的slow start，指数增加发送速率，目的也是尽可能快的占满管道，经过三次发现投递率不再增长，说明管道被填满，开始占用buffer它进入排空阶段（事实上此时占的是三倍带宽*延迟）

（2）在排空阶段，指数降低发送速率，（相当于是startup的逆过程）将多占的2倍buffer慢慢排空

（3）完成上面两步，进入稳定状态后，BBR改变发送速率进行带宽探测：先在一个RTT时间内增加发送速率探测最大带宽，如果RTT没有变化，后减小发送速率排空前一个RTT多发出来地包，后面6个周期使用更新后的估计带宽发包

（4）还有一个阶段是延迟探测阶段：BBR每过10秒，如果估计延迟不变，就进入延迟探测阶段，为了探测最小延迟，BBR在这段时间内发送窗口固定为4个包，即几乎不发包，占整个过程2%的时间。

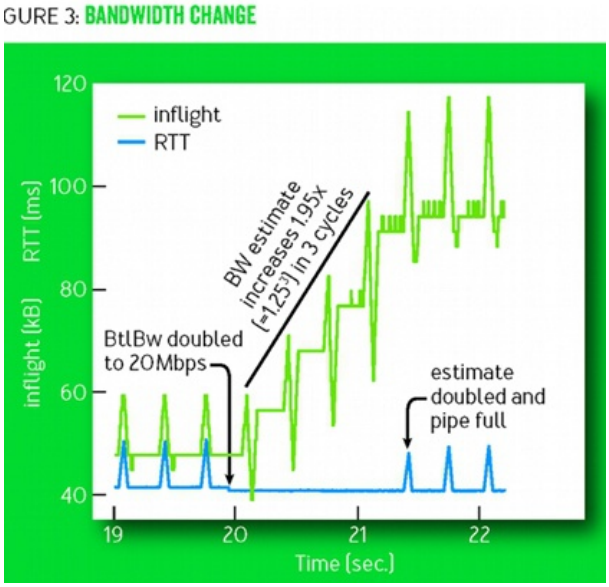
4.2带宽的动态更新（带宽探测）



引用自参考资料[1]

带宽探测占据BBR绝大部分时间。在startup阶段，BBR已经得到网络带宽的估计值。在带宽探测阶段，BBR利用一个叫做 cycle gain的数组控制发送速率，进行带宽的更新。cycle gain数组的值为 5/4, 3 / 4, 1, 1, 1, 1, 1, 1，BBR将max BtlBW*cycle gain的值作为发送速率。一个完整的cycle包含8个阶段， 每个阶段持续时间为一个RTprop。

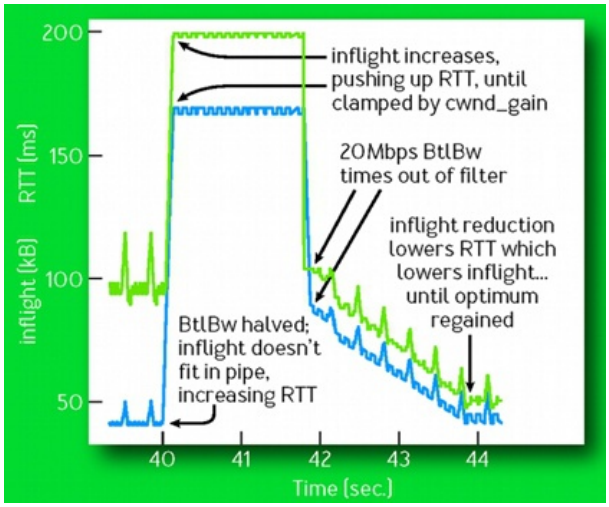
故如果数组的值是1，就保持当前的发送速率，如果是1.25，就增加发送速率至1.25倍BW，如果是0.75，BBR减小发送速率至0.75倍BW。



引用自参考资料[1]

上图显示的是一个a 10-Mbps, 40-ms的数据流在第20s时网络带宽增加一倍至20 Mbp时，BBR是如何更新的。

向上的尖峰表明它增加发送速率,向下的尖峰表明它降低发送速率，如果带宽不变，增加发送速率肯定会使RTT增加，降低发送速率是为了让他要把上一周期多发的包排空，如果带宽增加，则增加发包速率时RTT不变。这样经过三个周期之后，估计的带宽也增加了一倍。

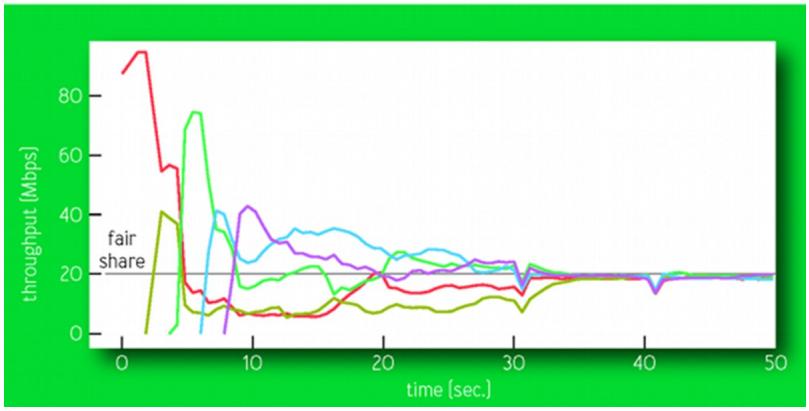


引用自参考资料[1]

上图所示为在第40s,是当网络带宽由20 Mbps减半至10 Mbps时，BBR的带宽探测如何发挥作用。因为发送速率不变，inflight增加，多出来的数据包占用了buffer，RTT增加，BBR从而减小发送速率，经过4秒后，有效带宽也降低为一半。

4.3多条BBR数据流分享瓶颈链路带宽

FIGURE 6: THROUGHPUTS OF 5 BBR FLOWS SHARING A BOTTLENECK



引用自参考资料[1]

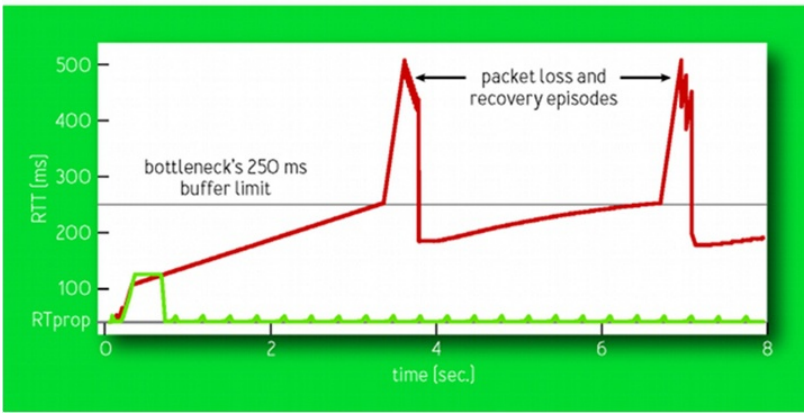
上图所示为多条独立的BBR数据流分享100-Mbps/10-ms瓶颈链路的情况：一条连接独占整个链路时，它的可用带宽近似为链路的物理带宽，n条连接共享链路时，最理想最公平的情况就是BW/n。

每条连接的startup阶段都会尝试增加带宽，所以吞吐量会有一个向上的尖峰，已经在链路中的连接会检测到拥塞而减小自己的发送速率，吞吐量会下降。

最后通过反复的带宽探测，他们都会趋于收敛，带宽得到均分。

5.实验结果

FIGURE 5: FIRST 8 SECONDS OF 10-MBPS, 40-MS CUBIC AND BBR FLOWS



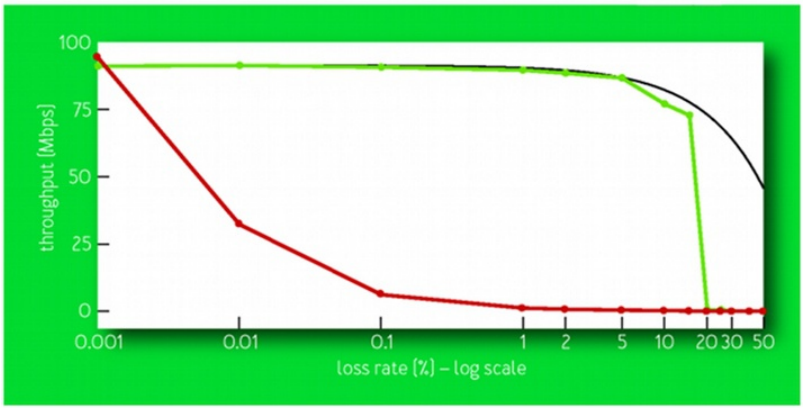
引用自参考资料[1]

上图所示为CUBIC（红线）和BBR（绿线）的RTT 随时间的变化：

CUBIC（红线）：可见周期性地延迟变化，缓存几乎总是被填满。

BBR（绿线）：除了在statup和drain阶段，RTT会发生较大的变化，在网络带宽保持不变时，RTT只有细微的变化，这些小尖峰是BBR尝试增加发包速率产生的，每8个往返延迟为周期的延迟细微变化。

FIGURE 8: BBR VS. CUBIC GOODPUT UNDER LOSS

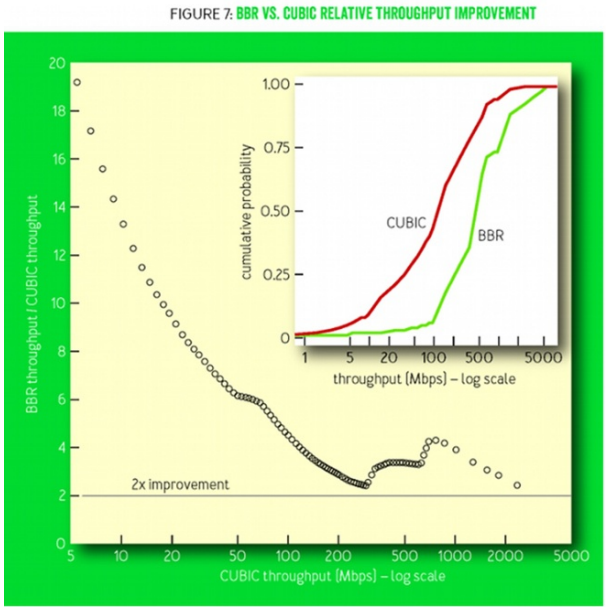


引用自参考资料[1]

上图所示为在有随机丢包情况下BBR（绿线）和CUBIC（红线）吞吐量的比较。

如图所示，CUBIC（红线）在万分之一丢包率的情况下，CUBIC带宽只剩30% 千分之一丢包率时只剩10%，在百分之一丢包率时就几乎没有吞吐量 。

而 BBR在丢包率5%以下几乎没有带宽损失。



引用自参考资料[1]

上图中，横轴是CUBIC的吞吐量，纵轴是BBR的吞吐量与CUBIC吞吐量之比，可见在吞吐量情况低的情况下，BBR与CUBIC吞吐量之比很大 ，说明吞吐量越低的网络，BBR性能越卓越。

6.总结

（1）吞吐量的显著提高

BBR已经在Google跨数据中心的内部广域网（B4）上部署，相对于CUBIC，BBR的吞吐量提高了133

（2）易于集成

能够在开源的Linux kernel中应用且只需要改变数据发送端。

7.参考资料

[1]Cardwell,Neal,et al.BBR:Congestion-Based Congestion Control.Queue14.5(2016):50.

[2]Yuchung Cheng,Neal Cardwell.Making Linux TCP Fast[EB/OL]. [2016-10]. http://netdevconf.org/1.2/slides/oct5/04_Making_Linux_TCP_Fast_netdev_1.2_final.pdf

[3]李博杰.Linux Kernel 4.9 中的 BBR 算法与之前的 TCP 拥塞控制相比有什么优势[EB/OL].[2016-12-15].https://www.zhihu.com/question/53559433.

[4]Bomb250.来自Google的TCP BBR拥塞控制算法解析[EB/OL].[2016-10-16].http://blog.csdn.net/dog250/article/details/52830576.

[5]陈皓.TCP 的那些事儿(下)[EB/OL].[2014-5-28].http://coolshell.cn/articles/11609.html.