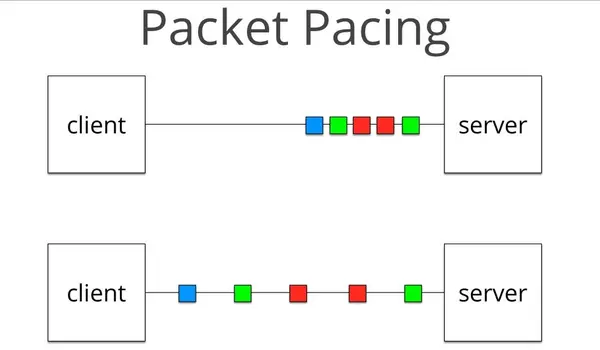
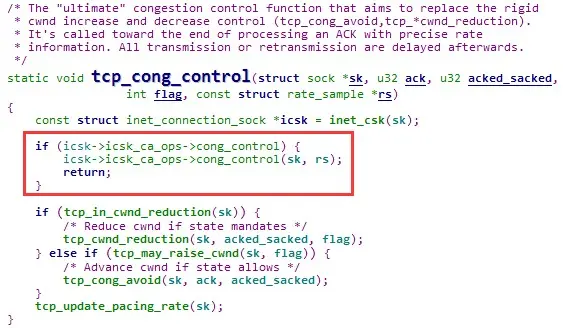
# pacing\_rate

BBR总是在测量最小RTT（10s内），最大Bandwidth（10 Round Trips），并且尽量控制输出到网络的数据包（in-flight）靠近 BDP（without buffer），这样既能保证带宽利用率，又能避免Bufferbloat问题。

现在广泛使用的CUBIC/(new)Reno都是基于丢包的，在算法上重点输出拥塞窗口（cwnd）；  
而BBR输出cwnd和pacing\_rate，且pacing\_rate为主，cwnd为辅，参考



**BBR对TCP的大胆改动：**



红框中是BBR加入时添加的，这里很明显，BBR从TCP接管了充分的控制权。  
从工程实现的角度来看，BBR这个小小的修改把TCP的 可靠传输 / 拥塞控制 解耦了 —— TCP专注于自己的可靠性（当然还有很多其它细节），BBR总是会负责任地告诉你（TCP）现在可以发多少数据，以什么速度发出这些数据。

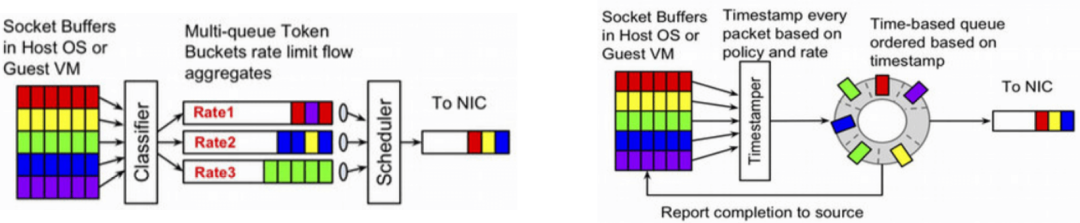
# 影响Pacing rate的性能因素

另外，Pacing就是大家原来说的rate-based congestion control, 给一个Rate发送而不是发送一个窗口，例如TFRC，WebRTC早已这样实现。其性能与控制间隔有关，调整慢了就会导致delay大甚至丢包。

# Shape不再基于排队（queue），而是基于时间戳（EDT）

**两点核心转变：**

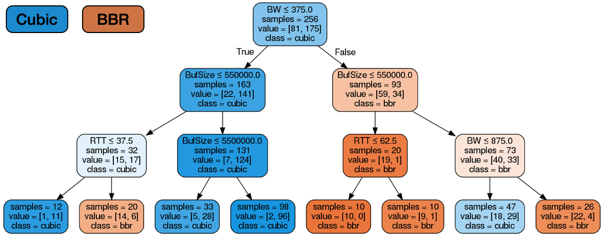
* 每个包（skb）打上一个最早离开时间（Earliest Departure Time, EDT），也就是最早可以发送的时间戳；
* 用时间轮调度器（timing-wheel scheduler）替换原来的出向缓冲队列（qdisc queue）

Fig. 传统基于 queue 的流量整形器 vs. 新的基于 EDT 的流量整形器

# When is BBR useful?

To investigate whether BBR or Cubic achieves higher goodput in different scenarios, we conducted 640 iperf3 experiments in our LAN network. We collected the goodput values of BBR and Cubic during these experiments and generalized the values by a decision tree (using the DecisionTreeClassifier package in Python3) in Figure 1.

In the decision tree, the orange nodes represent instances where BBR achieves higher goodput, whereas the blue nodes represent instances where Cubic achieves higher goodput. We observe in the figure that it is the relative difference between the bottleneck buffer size and BDP that typically dictates when BBR performs well — under small BDP and deep buffer size, Cubic achieves higher goodput, while under large BDP and shallow buffer size, BBR achieves higher goodput.

Figure 1 — Decision tree for employing BBR versus Cubic under different network conditions.

# 对于BBR算法的粗浅理

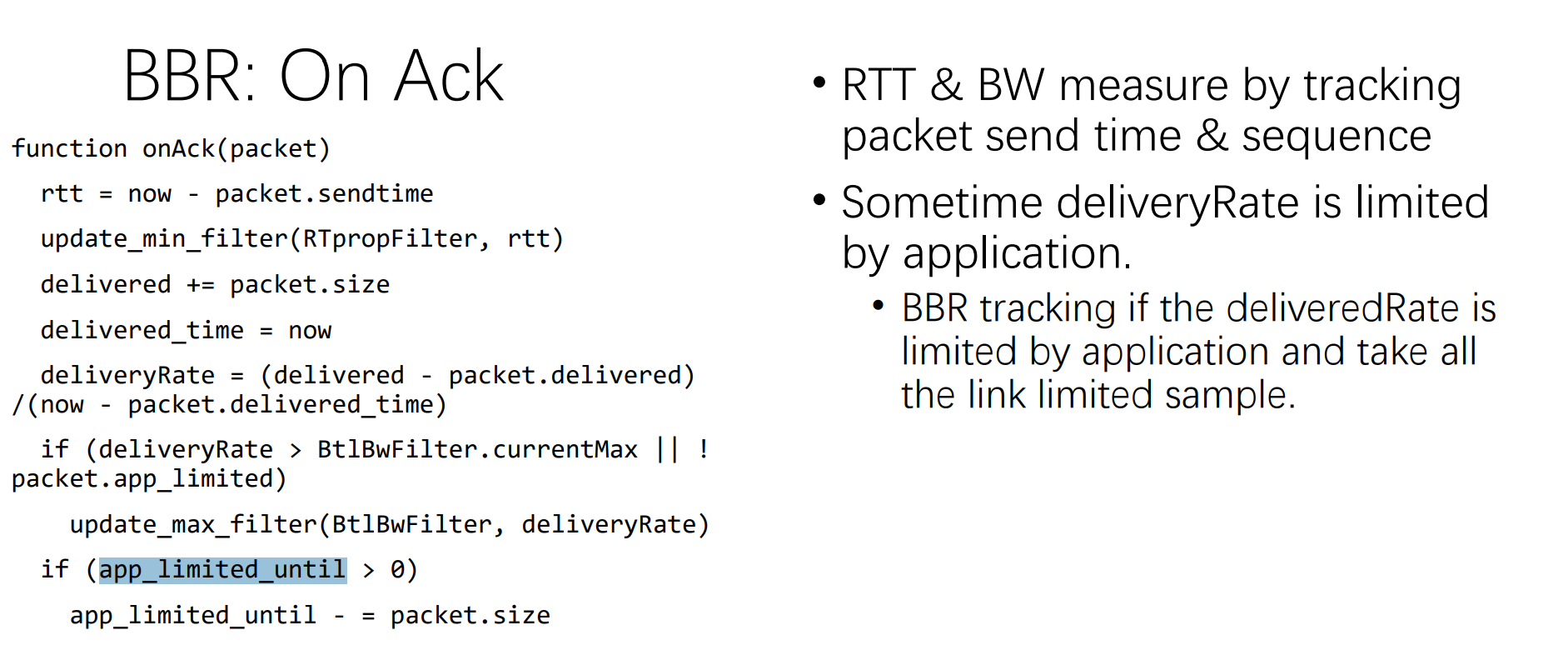
<https://zhuanlan.zhihu.com/p/384098666>

<https://zhuanlan.zhihu.com/p/673547654>

## 核心算法

基于以上的一些简单数学推导，我们可以通过[伪代码](https://zhida.zhihu.com/search?content_id=173485935&content_type=Article&match_order=1&q=%E4%BC%AA%E4%BB%A3%E7%A0%81&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)介绍BBR算法。笔者理解BBR算法的核心就是一个基于线性规划最优性结论的在线近似算法，通过ack的一些数据准确估计RTProp和BtlBw两个值，然后控制包的发送。

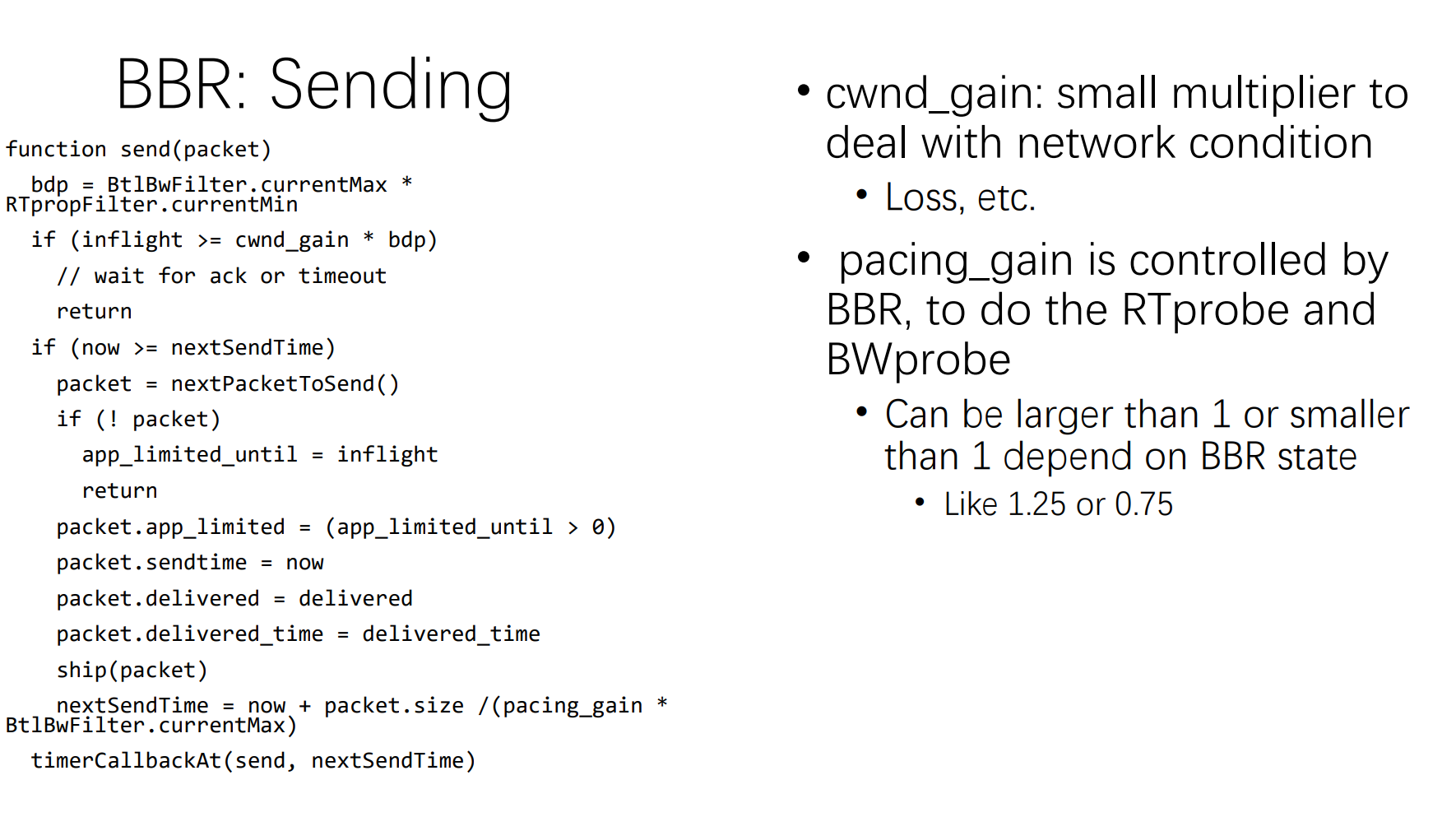




谷歌原论文中给的伪代码如上。结合上述数学推导看起来非常好理解。当收到ACK的时候，发送方首先计算RTT，更新RTT的最小值，然后计算deliveryRate。

这里需要解释一下app\_limited，这是RFC2861中TCP连接数据传输的一种状态，指TCP的数据传输速率受限与应用层的数据写入速率，并没有到达拥塞窗口上限。根据那个折线图也可以知道在app\_limited阶段deliveryRate和带宽无关。





首先计算BDP（最优点的inflight data）。这里需要解释一下cwnd\_gain。原文（[2]）中的解释是普通的网络和客户端的一些问题，比较模糊。不过个人认为这里说的是[3]中给出的[bufferbloat](https://zhida.zhihu.com/search?content_id=173485935&content_type=Article&match_order=1&q=bufferbloat&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)的解释：即客户端估计的发送窗口大小总是略大于实际大小，这是因为网络中的一些buffer用来吸收了网络中的流量波动。

之后在可以发送时，如果没有待发送的包，则说明应用层中的包不足，将app\_limited标记为 true，以及标记 until的值供接受ack的时候使用。发送包之后计算nextSendTime。

这里需要解释一下这个公式。pacing\_gain是一个控制参数。然后为了让包到达传输层的速度和发送速度匹配，下一次时间就是现在的时间加上包大小/带宽（发送掉这个包的时间）。

这个pacing-gain的意义就在于发送方很难探测到带宽的变化，只能通过一个类似于随机下降的参数去学习。若pacing-gain大于1，发送过于密集之后发送时就会产生队列而使RTT增加，deliveryRate下降，减少BtlBw.currentMax值。小于1的时候同理。这里的想法笔者认为有点像随机梯度下降，这加速了这两个值的收敛以更好地应对不稳定网络的情况。