BBR 论文中文翻译（原文：BBR: Congestion-Based Congestion Control)

译者：林佳烁 邮件：15622383059@163.com

Measureing bottleneck bandwidth and round-trip propagation time

因为各种原因，今天的互联网并不能像我们所期望的那样很好地传输数据。世界上大部分蜂窝网络用户都会经历几秒乃至几分钟的时延；在公共局域如机场和会议厅等，WIFI质量经常是非常差的。物理和气候学者想要与全球范围内的合作者传输千兆字节级别的数据，但可能会发现他们精心准备的Gbps级别的底层网络在洲际传输中只能达到Mbps级别。

这些问题之所以会发生，是因为在80年代设计TCP拥塞控制的时候，TCP将丢包作为“拥塞”的信号。在那个年代，这个假设确实是正确的，但这是因为受限于技术原因。而当网卡的速率从Mbps级别进化为Gbps级别，内存从KB级别进化到GB级别之后，丢包与拥塞的关系变得不那么紧密了。

今天的这些TCP拥塞控制算法，包括至今为止最好的算法CUBIC，都是基于丢包的。他们是造成这些问题的主要元凶。当瓶颈链路的缓存较大时，这些基于丢包的拥塞控制算法流填满了缓存，造成了bufferbloat。当瓶颈链路的缓存较小时，这些算法会又将丢包作为发生拥塞的信号，从而降低速率导致了较低的吞吐量。为了解决这些问题，必须提出一种不是基于丢包的拥塞控制算法，这需要设计者对网络拥塞是如何并且在哪里产生的有非常深刻的理解。

Congestion and Bottlenecks

在一个TCP连接中，每个传输方向都存在一个最慢的链路，或者说瓶颈链路（bottleneck）。

Bottleneck很重要！这是因为：

1. 它决定了该连接的最大传输速率（举个例子：如果高速公路上的某一段路发生了车祸，将会导致该道上的车速降低至那一段路的车速）。
2. 它是造成队列的元凶！因为只有一个链路的离开速率大于它的到达速率，队列才会缩短。对于一个尽力传输最大速率的连接来说，因为其他的链路速率都比bottleneck大，所以最后会造成bottleneck的队列。

无论一个TCP连接需要穿越多少链路，也无论这些链路的速率各自是多少，从TCP的角度来说，一个及其复杂的路径的行为终究跟一条与它拥有相同RTT和bottleneck速率的简单链路一样。这两个参数，RTprop（Round-trip propagation time）和BtlBw（bottleneck bandwidth），决定了传输的性能。（打个比方，如果将一条网络路径类比为一个管道，RTprop就是管道的长度，BtlBw就是管道中最短的半径。）

图1展示了RTT和发送速率与发送数据大小的关系。图中的蓝线受到了RTprop的约束，绿线受到BtlBw约束，红线受到瓶颈链路的缓存约束。注意阴影区域的部分是不可达的，因为这会至少违反一项约束。这三种约束形成了3不同的区域，分别为app-limited，bandwidth-limited，buffer-limited。在三种区域，流的行为是完全不同的。

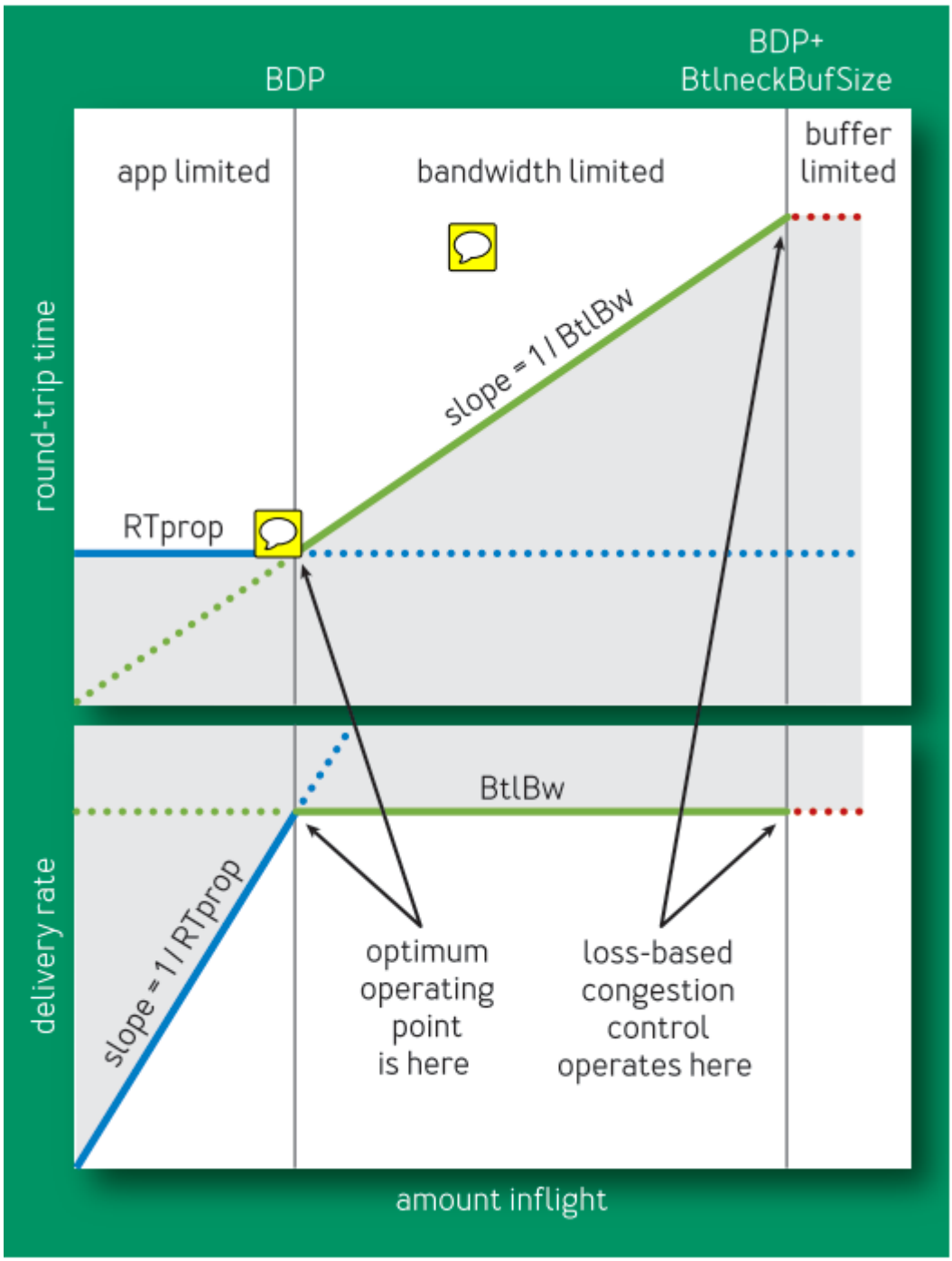

图1 发送速率和RTT vs 在外数据<\center>

当没有足够的数据来填满管道时，RTprop决定了流的行为；当有足够的数据填满时，那就变成了BtlBw来决定。这两条约束交汇在点inflight=BtlBw*RTprop，也就是管道的BDP（带宽与时延的乘积）。当管道被填满时，那些超过的部分（inflight-BDP）就会在瓶颈链路中制造了一个队列，从而导致了RTT的增大，如图1所示。当数据继续增加直到填满了缓存时，多余的报文就会被丢弃了。拥塞就是发生在BDP点的右边，而拥塞控制算法就是来控制流的平均工作点离BDP点有多远。

基于丢包的拥塞控制算法工作在bandwidth-limited区域的右边界区域，尽管这种算法可以达到最大的传输速率，但是它是以高延迟和高丢包率作为代价的。在存储介质较为的时候，缓存大小只比BDP大一点，此时这种算法的时延并不会很高。然而，当存储介质变得便宜之后，交换机的缓存大小已经是ISP链路BDP的很多很多倍了，这导致了bufferbloat，从而导致了RTT从毫秒级升到了秒级。

工作在bandwidth-limited区域的左边界比工作在有边界好。在1979年Leonard Kleinrock就展示了，无论是对流自身而言，或是对整个网络来说，工作在左边界是最优点，这个点在实现最大传输速率的同时，保持了低时延和低丢包。不幸的是，当时Jeffrey M.Jaffe也同时证明了不可能存在一个分布式算法可以收敛到这个边界点。这使得学术研究不再尝试去设计可以工作在该边界点的分布式算法。

我们谷歌工作组每天花了大量的时间来查看从世界各地发来的TCP报文头部，从而对流的行为机制有了很深的了解。我们通常首先计算出流最重要的两个参数，RTprop和BtlBw，这两个参数都可以从trace中推断出来。这表明Jaffe的结论可能不再适用。他当时做出这个结论是因为测量具有模糊性（例如，RTT的增加有可能是因为流的路径改变了，也有可能是瓶颈链路的带宽减少了，也有可能是因为别的流竞争而导致队列等等）。尽管不可能在单次测量中得到非常可靠的值，但是一个持续时间较长的连接会告诉你很多信息，从中或许可以设法来估计得到一个可靠的值。

使用最近在控制领域中提出的鲁棒伺服环来对这些观测结果进行处理，可以设计出一种分布式拥塞控制协议。该协议可以针对真实的拥塞进行反应，而不是基于丢包或者短暂的队列时延的，它可以大概率收敛到Kleinrock的最优边界点。因此这推动了我们近三年的研究——如何基于这两个观测参数bottleneck带宽及RTT（这就是BBR缩写的来源，bottleneck bandwidth and round-trip propagation time）来设计拥塞控制算法。

Characterizing the bottleneck

当一个连接满足以下两个条件时，它可以在达到最高的吞吐量的同时保持最低时延：

1. 速率平衡：瓶颈带宽的数据到达速率与BtlBw相等；

2. 填满管道：所有的在外数据（inflight data）与BDP（带宽与时延的乘积）相等

其中，第一个约束保证了瓶颈带宽可以得到100%利用。而第二个约束保证了流有足够的数据来填满瓶颈链路而且同时不会溢出（排队）。第一个条件本身并无法保证路径中不存在队列，它只能保证流的速率不发生改变（例如，考虑一个连接在一开始就发送了10个报文到一个BDP只有5个网络中，并且接下来一直保持瓶颈速率发送。这样子会导致在一开始就填满了管道，并且制造了5个报文的队列，并且这个队列永远不会被消灭）。相似地，full pipe条件也不能保证链路中没有队列。比如，一个TCP连接可以以burst方式发送BDP数量的报文，若该连接每次发二分之一BDP，并且发两次，此时full pipe条件得到满足了，然而网络中的平均队列长度是BDP/4。为了最小化网络中的队列长度，唯一的方式是同时满足以上两个条件。

然而，BtlBw和RTprop可能是动态变化的，所以我们需要实时地对它们进行估计。目前TCP为了检测丢包，必须实时地跟踪RTT的大小。在任意的时间t，

$$RTT_t = RT_{prop_t} + \eta_t$$

其中，最后一项表示“噪声”。造成噪声的因素主要有：链路队列，接收方的时延ACK配置，ACK聚合等因素等待。RTprop是路径的物理特性，并且只有路径变化才会改变。由于一般来说路径变化的时间尺度远远大于RTprop，所以RTprop可以由以下公式进行估计：

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

即，在一个时间窗口中对RTT取最小值。一般将该窗口大小设置为几十秒至几分钟。

然而，bottleneck bandwidth的估计不像RTT那样方便，没有一种TCP spec要求实现算法来跟踪估计bottleneck带宽，但是，我们可以通过跟踪发送速率来估计bottleneck带宽。当发送方收到一个ACK报文时，它可以计算出该报文的RTT，并且从发出报文到收到ack报文这段时间的data Inflight。这段时间内的平均发送速率就可以以此计算出来：deliveryRate = delta delivered/ delta t。这个计算出的速率必定小于bottleneck速率（因为delta delivered是确定的，但是delta t会较大）。因此，BtwBw可以根据以下公式进行估计。

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

其中，时间窗口大小的值一般为6~10个RTT。

TCP必须记录每个报文的离开时间从而计算RTT。BBR必须额外记录已经发送的数据大小，使得在收到每一个ACK之后，计算RTT及发送速率的值，最后得到RTprop和BtlBw的估计值。

值得注意的是，这两个值是完全独立的：RTprop可以发生变化然而保持bottleneck不变（比如发生路由变化），或者BtlBw可以变化而路径不变（比如无线链路速率发生变化）。（This independence is why both constraints have to be known to match sending behavior to delivery path.）如图1所示，只有在BDP的左边，才能观测到RTprop，并且只有在BDP的右边才能观测到BtlBw，他们遵循了一个不确定性原则：当其中一个可以被观测的时候，另外一个并不能。直觉地，这是因为为了观测出管道的容量，必须填满管道，而这会创造出一个队列从而无法观测到管道的长度。例如，一个使用request/response协议的应用可能永远不会发送足够的数据来填满管道，并且观测到BtlBw。一个持续几个小时的大文件传输应用可能永远处在bandwidth-limited区域，而仅仅在第一个报文中的RTT采集到RTprop。这种不确定性机制意味着，在信息变得不明确的时候，必须要有机制来从当前的工作状态中学习到这两个值的其中一个，并且进入对应的工作区来重新学习这两个值。

Matching the packet flow to the delivery path

BBR核心算法包含两大部分：

- **当收到一个ACK报文时：**

每一个ACK提供了一个新的RTT和新的发送速率估计，BBR将以此为根据来更新RTprop和BtlBw。

```
function onAck(packet)
    rtt = now - packet.sendtime
    update_min_filter(RTpropFilter, rtt)
    delivered += packet.size
    delivered_time = now
    deliveryRate = (delivered - packet.delivered)
                   /(now - packet.delivered_time)
    if (deliveryRate > BtlBwFilter.currentMax
        || ! packet.app_limited)
        update_max_filter(BtlBwFilter,
                          deliveryRate)
    if (app_limited_until > 0)
        app_limited_until - = packet.size
```

伪代码中的if语句是对上一段所讲的不确定性进行估计的：发送方可能受限于应用，发送数据太少，而并没有将管道填满。当发送方采取的协议是request/response类时，这种现象是非常常见的。当没有数据可以发送时，BBR会将对应的带宽估计标志为是应用限制的（见下文伪代码中的send()）。这里的算法是为了决定流应该采集哪些数据，而避免采集那些被应用限制的而不是被网络限制的采集数据。BtlBw是发送速率的上界，所以如果计算出的发送速率大于当前所估计的BtlBw值，那么这表示这个估计值太小了（无论该发送速率是否是应用限制的），我们都应该更新它。否则，那些被应用限制的采集数据将会被丢弃。（如图1所示，在app-limited区域中，deliveryRate低估了BtlBw。这些if判断避免BBR低估了BtlBw而导致发送低速率）。

- **当发送数据时**：

为了使得bottleneck链路的报文到达速率和报文离开速率相等，BBR必须进行packet paced。BBR的发送速率必须与bottleneck的速率相等，这意味着BBR的实现需要pacing的支持——pacing_rate是BBR的一个主要控制参数！第二个参数，cwnd_gain，将inflight控制为比BDP稍大一些，从而处理常见的网络机制和接收方机制（参见后文Dealyed and Stretched ACKs）。TCP发送的时候做的事大概如下列伪代码所示（在Linux中，可以使用FQ/pacing qdisc来发送数据，这可以使得BBR在与几千个低速率的paced流在千兆链路上很好地共存，并且使用FQ这种机制也不会额外造成CPU的负担）。

```
function send(packet)
    bdp = BtlBwFilter.currentMax * RTpropFilter.currentMin
    if (inflight >= cwnd_gain * bdp) // wait for ack or timeout
        return
    if (now >= nextSendTime)
        packet = nextPacketToSend()
        if (! packet)
```

```
            app_limited_until = inflight
            return
        packet.app_limited = (app_limited_until > 0)
        packet.sendtime = now
        packet.delivered = delivered
        packet.delivered_time = delivered_time
        ship(packet)
        nextSendTime = now + packet.size / (pacing_gain * BtlBwFilter.currentMax)
    timerCallbackAt(send, nextSendTime)
```

Steady-state behavior

BBR的发送速率和发送数量完全就是一个关于BtlBw和RTprop的函数，所以BBR应该小心地估计这两个值。这创造了一种新型的闭环控制，如图2所示。图2显示了一个10Mbps，40ms的流在700ms中的RTT（蓝线），inflight（绿线）和发送速率（红线）变化过程。注意图中在发送速率上方的灰线是BtlBw最大化滤波器的状态。图中产生的三角形形状是因为BBR的pacing_gain周期性的变化产生的，因为BBR必须以此来探测BtlBw是否提高了。图中展示了该增益值在一个周期不同时间的变化和其影响的数据变化。

BBR将它的大部分时间的在外发送数据都保持为一个BDP大小，并且发送速率保持在估计得BtlBw值，这将会最小化时延。但是这会把网络中的瓶颈链路移动到BBR发送方本身，所以BBR无法察觉BtlBw是否上升了。所以，BBR周期性的在一个RTprop时间内将pacing_gain设为一个大于1的值，这将会增加发送速率和在外报文。如果BtlBw没有改变，那么这意味着BBR在网络中制造了队列，增大了RTT，而deliveryRate仍然没有改变。（这个队列将会在下个RTprop周期被BBR使用小于1的pacing_gain来消除）。如果BtlBw增大了，那么deliveryRate增大了，并且BBR会立即更新BtlBw的估计值，从而增大了发送速率。通过这种机制，BBR可以以指数速度非常快地收敛到瓶颈链路。如图3显示的，我们在1条10Mbps，40ms的流在20s稳定运行之后将BtlBw提高了1倍（20Mbps），然后在第40s又将BtlBw恢复至20Mbps。

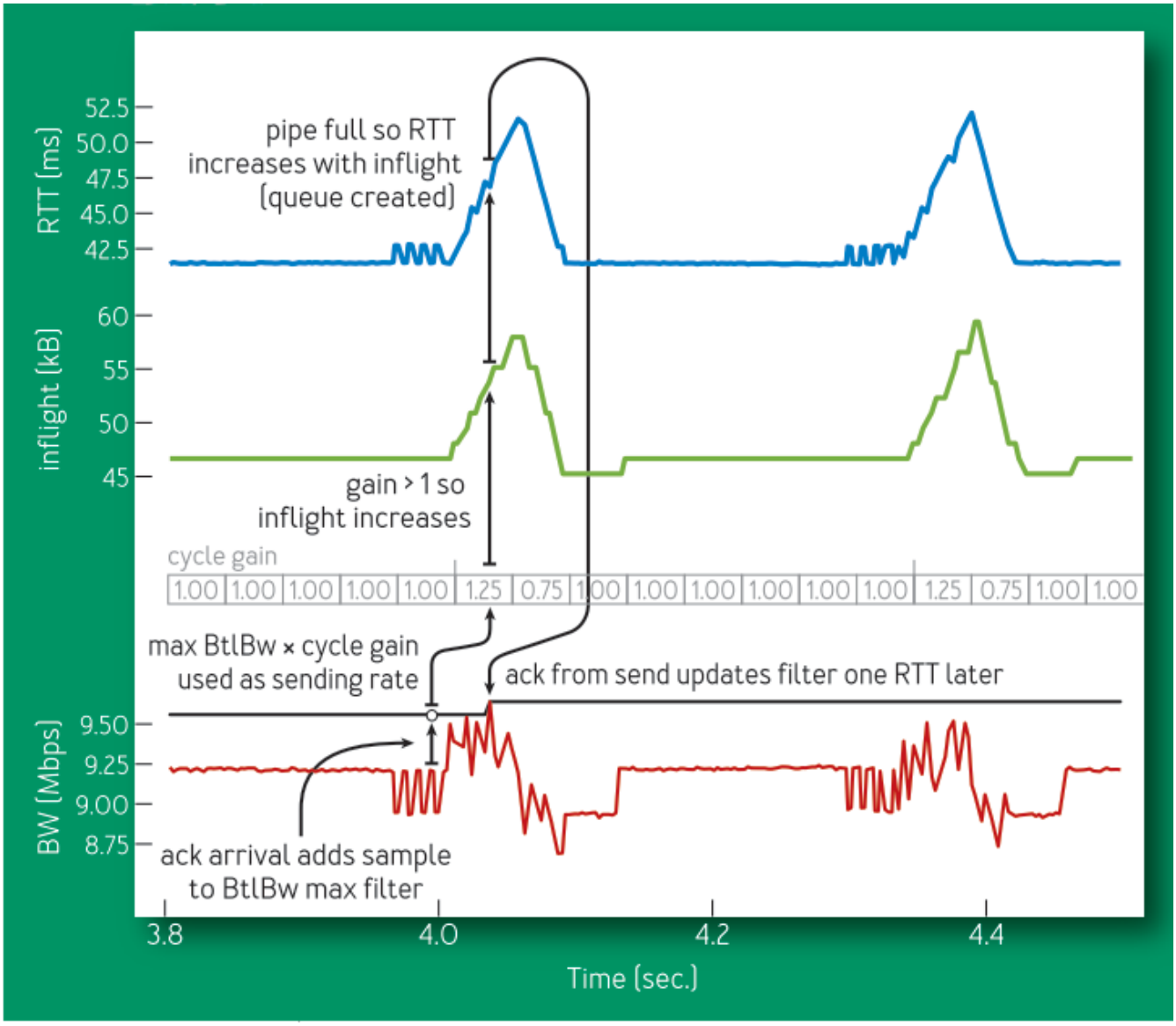

图2 RTT（蓝线），在外数据（绿线）和发送速率（红线）细节<\center>

(BBR is a simple instance of a Max-plus control system, a new approach to control based on nonstandard algebra.12 This approach allows the adaptation rate [controlled by the max gain] to be independent of the queue growth [controlled by the average gain]. Applied to this problem, it results in a simple, implicit control loop where the adaptation to physical constraint changes is automatically handled by the filters representing those constraints. A conventional control system would require multiple loops connected by a complex state machine to accomplish the same result.)

Single BBR flow Startup behavior

现有的TCP实现使用基于事件的算法来对事件如启动、关闭、丢包恢复（loss recovery）进行处理，这需要非常多的代码。BBR使用前文提到的代码（见Matching the Packet Flow to the Delivery Path一章）来应对一切。BBR通过一系列异步“状态”来处理事件，这些“状态”是根据一张包含多个固定增益和退出条件的表来定义的。BBR大多时候都出于ProbeBW阶状态（见Steady-state Behavior一章）。在连接启动的时候，BBR使用了Startup和Drain状态（见图4）。为了处理大小范围超过12个数量级的带宽，Startup对BtlBw进行二乘查找，它使用2/ln2的增益来对发送速率翻倍。这将会在log_2BDP的RTT时间内确定BtlBw的值，但会制造最多2个BDP的队列。一单Startup找到了BtlBw，BBR转变为Drain状态，该状态使用了Startup增益的倒数来消耗队列。消耗完队列之后，BBR进入ProbeBW阶段来维持1个BDP的在外数据。

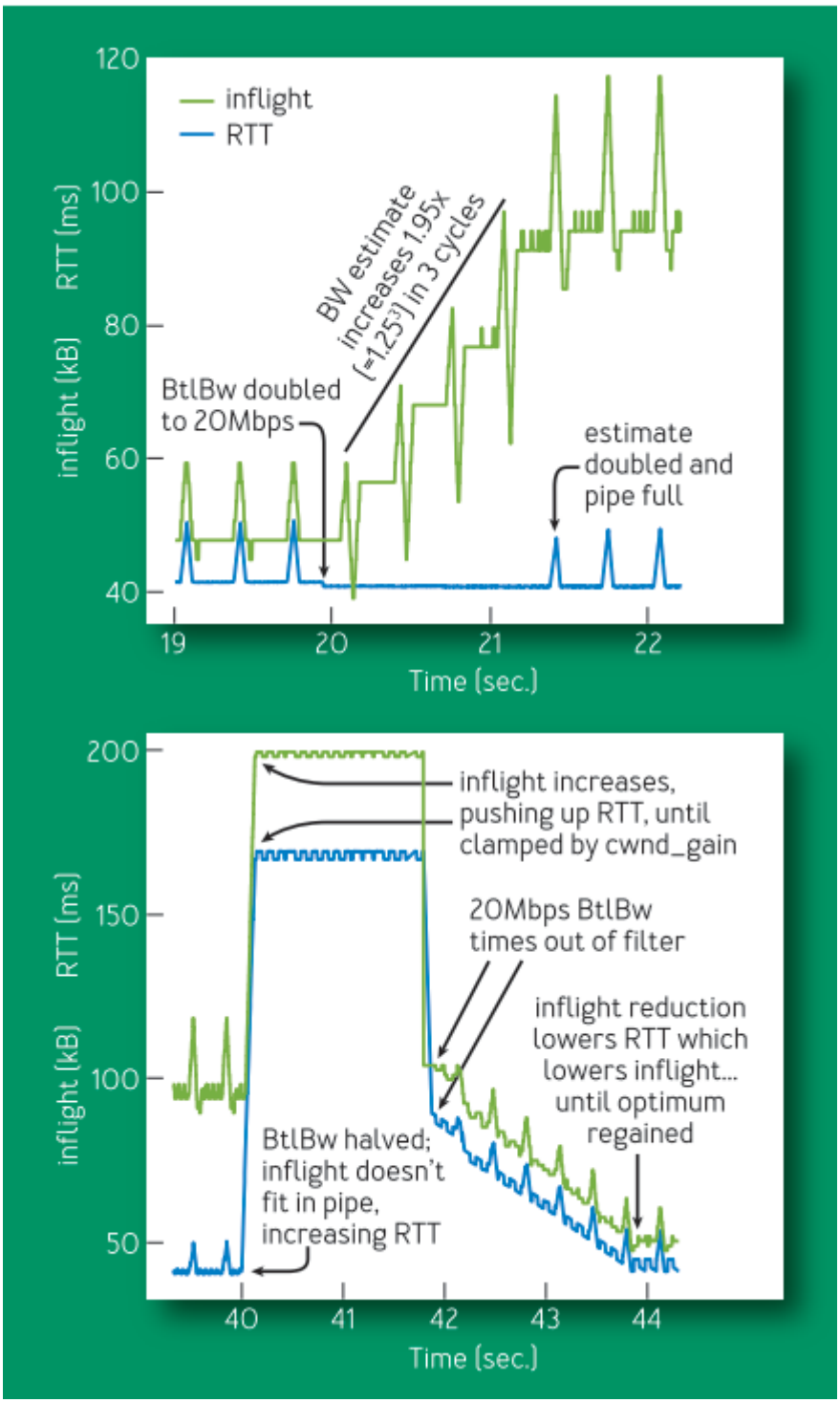

图3 带宽变化<\center>

图4展示了1个10Mbps，40ms的BBR流在一开始的1秒内，发送方（绿线）和接收方（蓝线）的过程。红线表示的是同样条件下的CUBIC发送。垂直的灰线表示了BBR状态的转换。下方图片展示了两种连接的RTT在这段时间的变化。注意，只有收到了ACK（蓝线）之后才能确定出RTT，所以在时间上有点偏移。图中标注了BBR何时学习到RTT和如何反应。

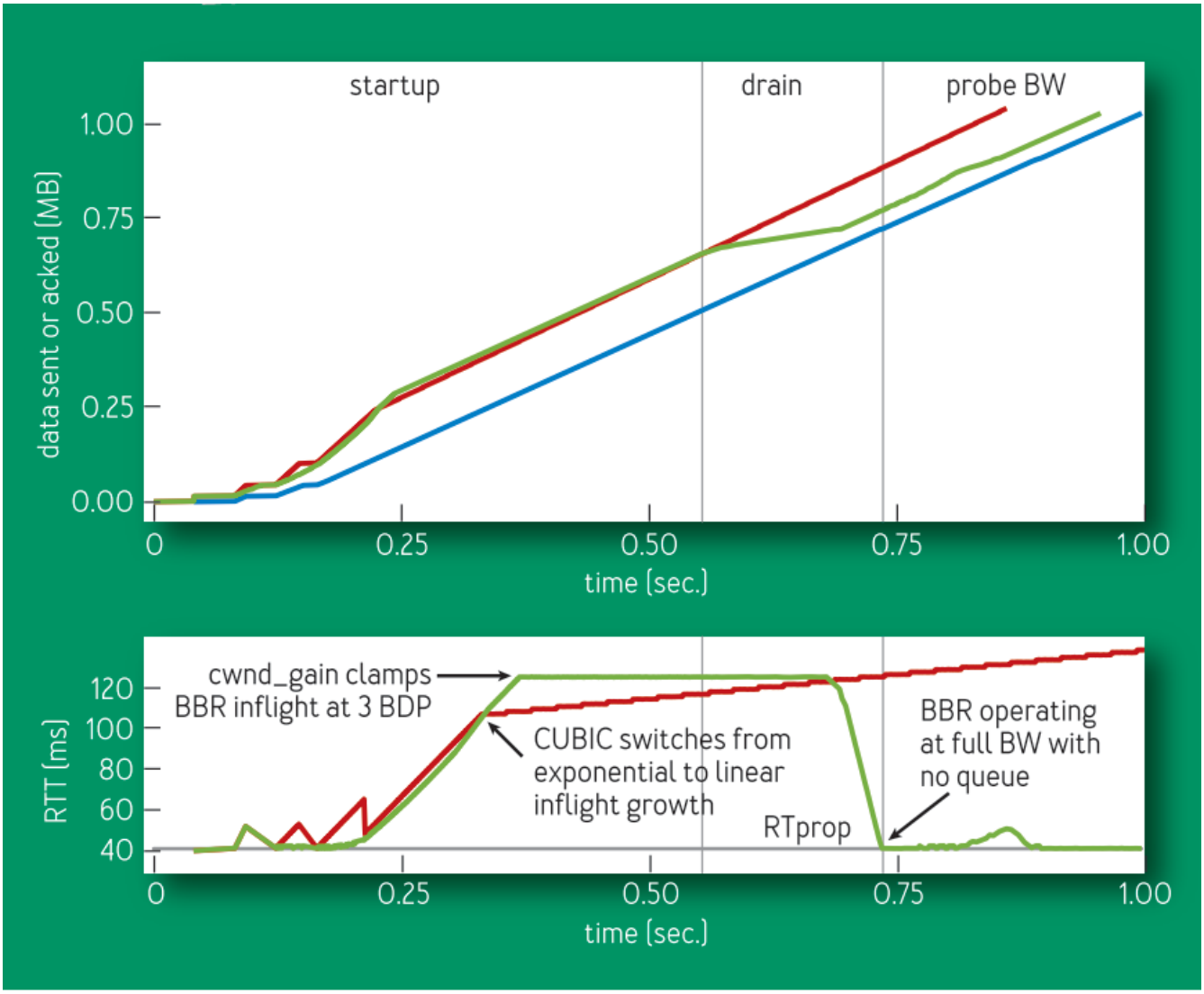

图4 10Mbps、40ms链路上BBR流的第一秒<\center>

图4的下图展现了BBR和CUBIC行为的巨大不同。他们的初始行为都是相似的，但是BBR可以完全地消耗掉它启动阶段产生的队列，而CUBIC并不能。CUBIC没有一个路径模型来告诉它应该发送多少报文，所以CUBIC一直在缓慢地增加他的在外报文，直到瓶颈链路缓存填满而丢包、或者接收方的缓存（TCP接受窗口）满了。

图5展示了在图4中展示的BBR和CUBIC流在开始8秒的行为。CUBIC（红线）填满了缓存之后，周期性地在70%~100%的带宽范围内波动。然而BBR（绿线）在启动过程结束后，就非常稳定地运行，并且不会产生任何队列。

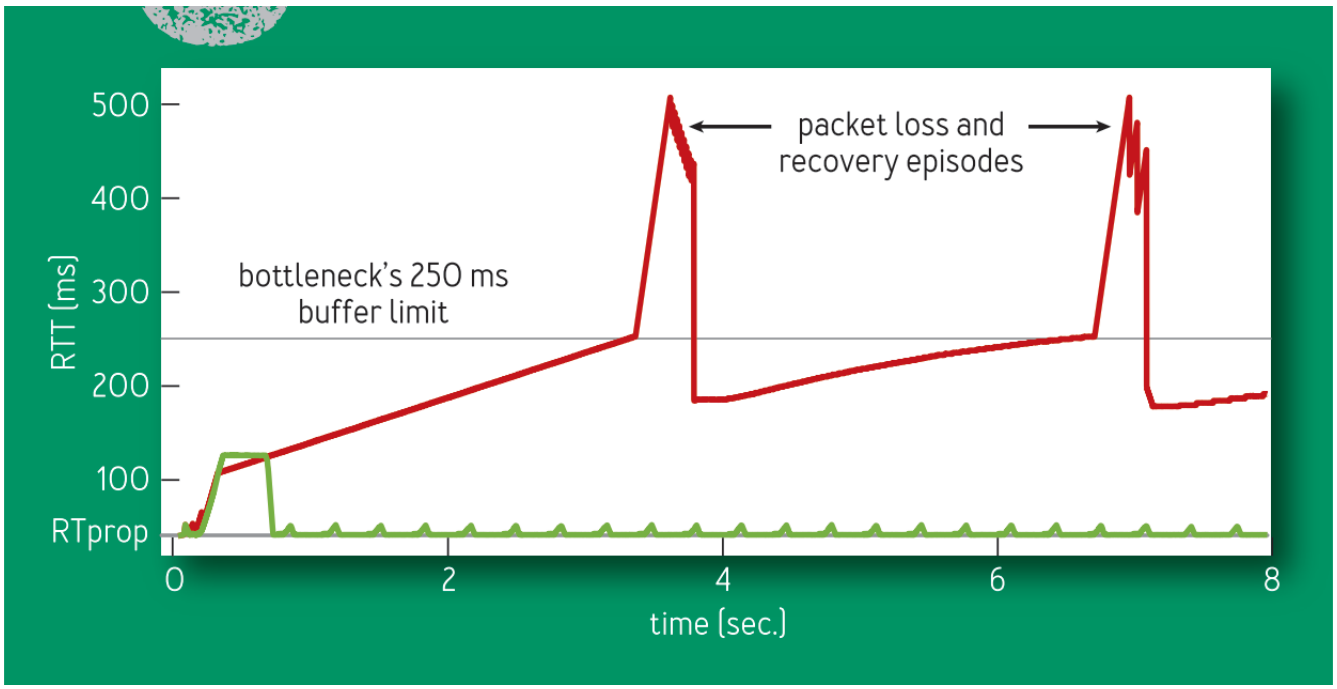

图5 在10Mbps、40ms链路上的BBR流和CUBIC流的前8秒对比<\center>

Multiple BBR flows sharing a bottleneck

图6展示了在一个100Mbps、10ms的瓶颈链路上共享的多个BBR流是如何收敛到公平值的。他们之所以会减小带宽是因为他们进入了ProbeRTT阶段，使得他们快速地收敛。

ProbeBW机制（见图2）会使得大流会让渡带宽给小流，最终使得每个流都学习到自己的fair share。尽管后发送者可能会因为别的流暂时地创造了队列，而高估了RTprop，从而导致不公平性，但最终都会在几个ProbeBW周期内收敛到公平值。

为了得到真实的RTprop值，一个流会使用ProbeRTT状态来将自己的工作点移动到BDP的左边：当好几秒内都没有更新RTprop时，BBR进入ProbeRTT状态，这会使得inflight在至少一个RTT内降低为4个报文，然后BBR再重新进入正常阶段。大流进入ProbeRTT阶段会消耗队列中的许多报文，所以会让其他的流学习到一个更小的RTprop值。这会同时让他们的RTprop估计值过期，并同时一起进入ProbeRTT阶段。这会更快地消耗掉队列中的报文，并使得更多的流学习到一个新的RTprop值并一直循环。这个分布式的合作机制是实现公平性和稳定性的重点！

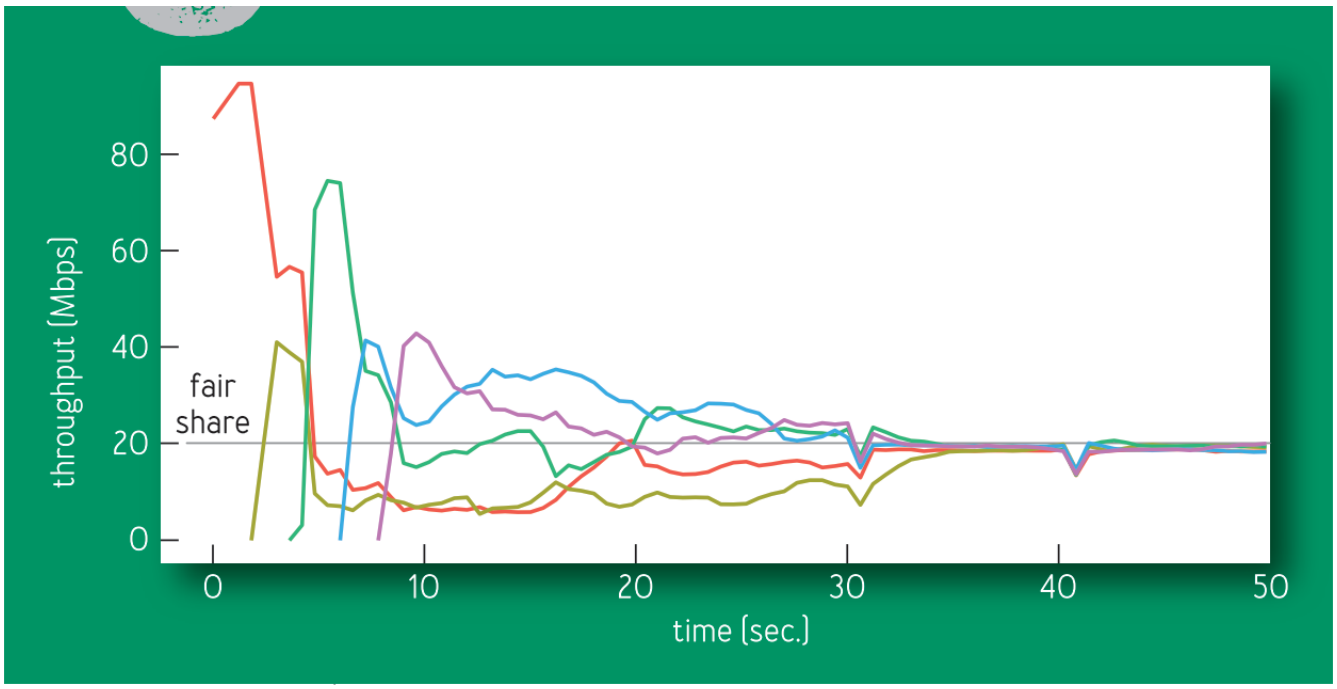

图6 共享同一个瓶颈链路的5个BBR流的吞吐量<\center>

BBR可以使得多个流共享一个链路而不会造成队列，而基于丢包的拥塞控制算法会造成队列周期性地增长然后再溢出，导致了高时延和丢包。

Google B4 WAN deployment experience

谷歌B4网络是一个使用商业交换机的高速广域网。在这个网络中，丢包经常是因为这些只有少量缓存的交换机没办法缓存大量突发性的报文。在2015年谷歌开始将B4的生产流量从CUBIC换成BBR。在这个过程中没有出现任何问题，一切都很顺利。直到2016年，所有B4的TCP流量都使用BBR。图7展示了做这种改变的一个原因：BBR的吞吐量能够稳定地维持在CUBIC的2到25倍。我们本来期望能够提高得更多，但是最终发现75%的BBR连接都被内核的TCP接收缓存限制了，这是因为网络运维组故意将缓存设成比较低的值（8MB）来避免CUBIC发送太多的inflight。手动地在一条从美国到欧洲的路径上的接收缓存调高，会立即将BBR的吞吐量提高至2Gbps，然而CUBIC还是维持在15Mbps（Mathis语言可以达到133倍的提高）。

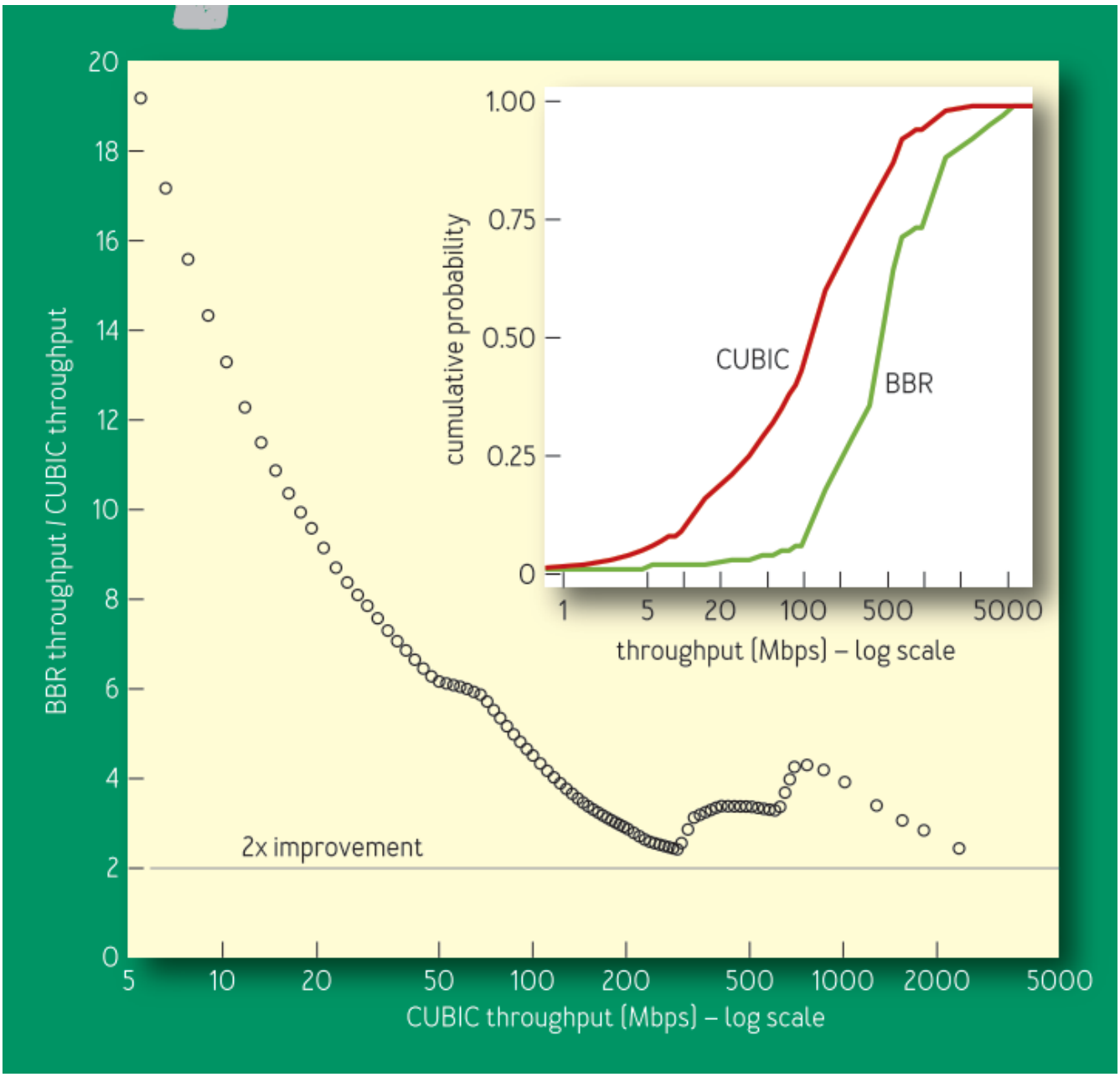

图7 BBR的吞吐量相对CUBIC的吞吐量提高比例<\center>

图7展示了BBR相比CUBIC的吞吐量提升，子图显示的是吞吐量的CDF（累积分布函数）。图中的数据来源于一个探测服务，该探测服务每分钟都会打开一个BBR连接和一个CUBIC连接到远端数据中心，然后传输8MB数据。这些连接包括了B4的很多路径，如在北美、欧洲、亚洲之间或者在洲内传输路径等等。

BBR不再使用丢包作为拥塞信号是我们的一大贡献！为了达到最大带宽，现有的基于丢包的拥塞控制算法需要丢包率小于BDP的倒数的平方（例如，在1Gbps/100ms的链路上，丢包率需要小于三千万分之一）。图八比较了BBR和CUBIC在不同丢包率的吞吐量。可以看到，CUBIC很受丢包影响，而BBR的影响并不大。当BBR的丢包率接近ProbeBW的增益时，估计到实际BtlBw的概率急剧下降，这导致了BBR低估了BtlBw。

图8展示了在一条100Mbps，100ms的链路上，BBR和CUBIC在60秒内的吞吐量与随机丢包率（从0.001%~50%）的关系。在丢包率只有0.1%的时候，CUBIC的吞吐量就已经下降了10倍，并且在丢包率为1%的时候就几乎炸了。而理论上的最大吞吐量是链路速率乘以（1-丢包率）。BBR在丢包率为5%以下时还能基本维持在最大吞吐量附近，在15%丢包率的时候虽然有所下降但还是不错。

YOUTUBE edge deployment experience

我们将BBR部署在Google.com和YouTube的视频服务器上。我们随机挑选一部分用户来使用CUBIC或BBR来进行实验。使用BBR算法的playback基本上在所有的YouTube的QoE参数上都有所显著提高，这可能是因为BBR的行为是更加一致的并且是可预测的。由于YouTube已经能很好地自适应将视频传输速率调节到BtlBw之下来避免bufferbloat和重缓存，所以BBR在吞吐量上只比CUBIC提高了一点点。尽管如此，在全世界范围内，BBR将RTT中位数平均降低了53%，而这个数值在发展中国家是80%。图9统计了在一星期中，在五大洲的超过2亿YouTube的playback连接中，BBR的RTT中位数相对于CUBIC的提高幅度。

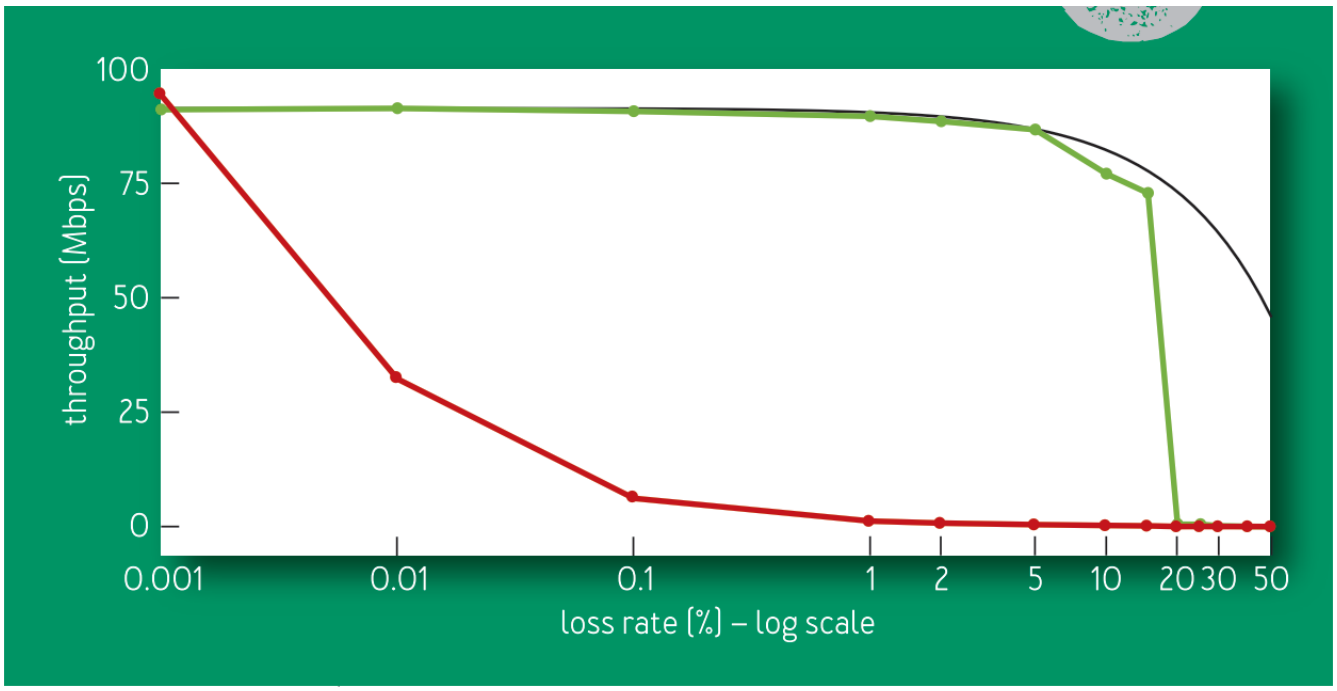

图8 BBC和CUBIC的吞吐量与丢包率的关系<\center>

在全球范围内的70亿互联网移动终端中，超过一半设备使用8至114kbps的2.G系统。因为基于丢包的拥塞控制算法填满buffer的特点，这些连接已经暴露了太多太多的问题。他们的bottleneck链路一般情况下都是无线终端与基站之间的链路。图10展示了使用BBR和使用CUBIC在基站与用户终端见的延迟的对比。其中的水平线标志了一个很严重的结果：TCP在连接建立阶段没有办法忍受太长的RTT时延，在该阶段，TCP等待SYN的时延是一个与操作系统相关的timeout。当移动设备在接收大量的数据而此时的基站缓存又足够大时，在基站的队列耗尽之前，移动终端将没有办法打开新的连接到互联网！（这是因为移动终端等到花儿都谢了都没有等到SYN ACK）

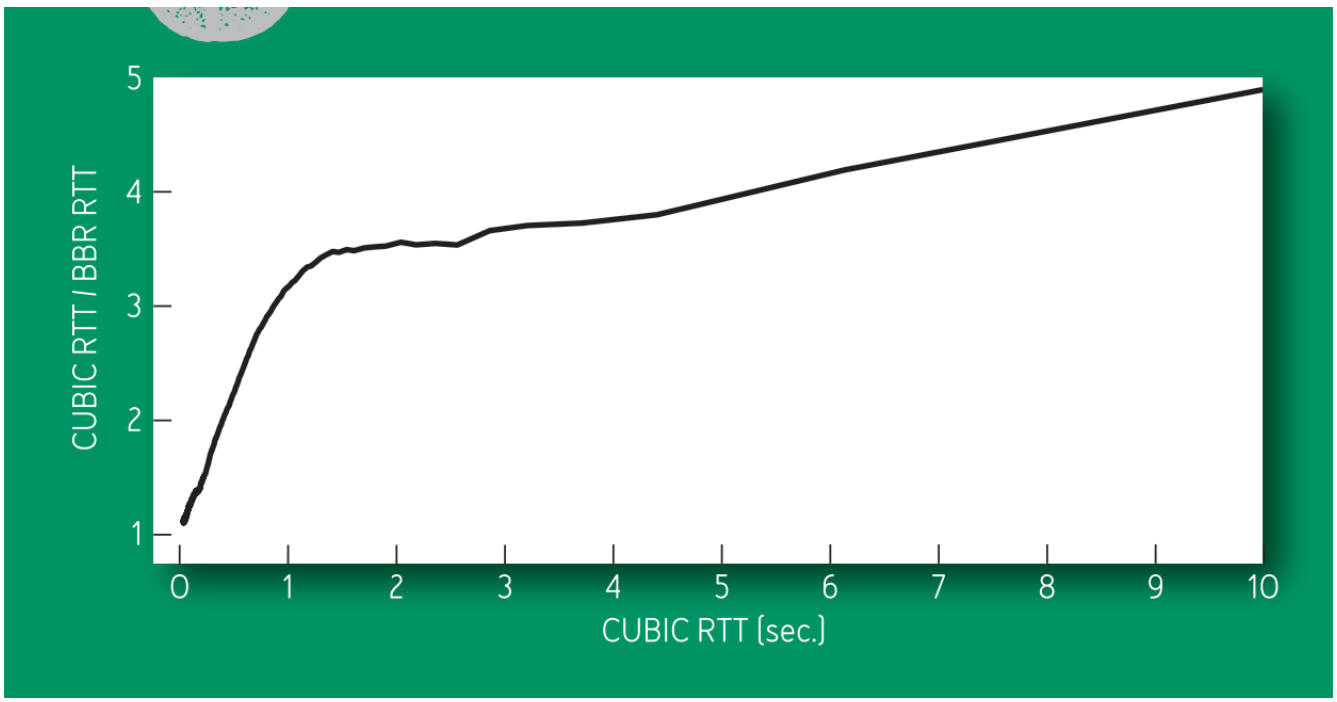

图9 BBR的RTT中位数相比CUBIC的提高比例<\center>

图10展示了在稳定状态下，在一个具有8个BBR（绿线）或CUBIC流（红线）的128Kbps，40ms的链路上，RTT中位数与链路缓存大小的关系。由图中可以看到，无论链路缓存如何变化，BBR都可以保持队列为空。而由于CUBIC总会把缓存填满，所以CUBIC的曲线随着缓存的增大而线性增长。

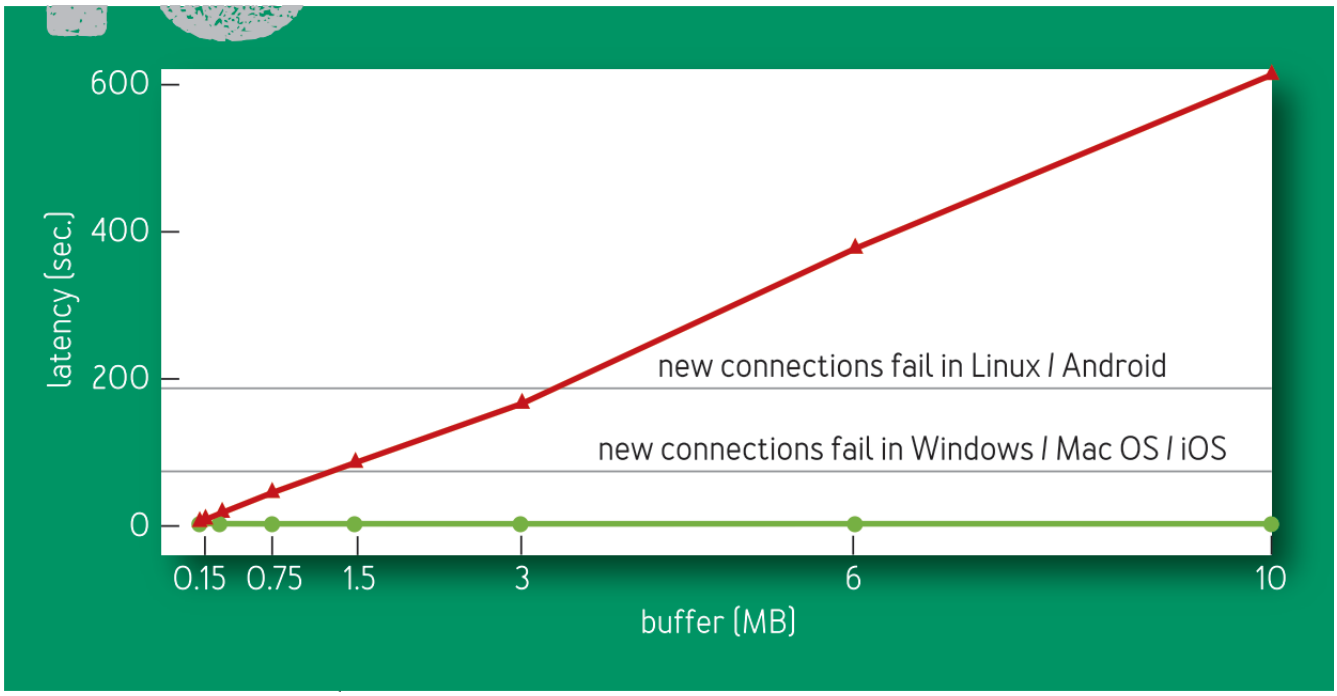

图10 稳定状态下RTT中位数随着链路缓存大小的变化<\center>

Mobile cellular adaptive bandwidth

蜂窝系统对每一个用户使用一个队列来统计报文数量，并以此为根据来评估用户的需求带宽，从而自适应调整每个用户的带宽。早期我们为了不创造太深的队列，调整了BBR，但是这导致了连接速率非常慢。提高ProbeBW的pacing_gain会创造较长的队列，但会减少低速连接的数量。从这里我们学到了，BBR可能对某些网络太过仁慈了！目前我们将BtlBw的增益设置为1.25，经过测试，BBR不会在任何一种网络中输给CUBIC了！

Delayed and stretched acks

蜂窝、WiFi以及有限宽带网络经常会将ACK进行延时或者聚合。当BBR将带宽控制为1个BDP时，这种行为可能会导致吞吐量不够高。可以将ProbeBW的cwnd_gain提高为2从而允许BBR持续地以估计的速率平滑地发送数据，即使ACK被延时了1个RTT。这可以很好地避免BBR产生低吞吐量。

Token-bucket policers

早期我们将BBR部署在YouTube的时候，我们就发现了全球范围内大多数ISP都会使用令牌桶策略将流量进行限速。

大多数情况下，在连接的开始阶段，bucket都是满的，所以BBR可以学习到网络的Btlbw，然而一旦bucket空了，所有发送速率大于bucket补充速率（该速率远小于BtlBw）的报文都会被丢弃。最终BBR会学习这个新的发送速率，然而ProbeBW仍然会导致持续丢包。为了避免浪费带宽，为了降低这些丢包带来的时延，我们为BBR添加了一个流量管控检测模块和一个显式的流量管控模块。我们也正在寻找更好的办法来降低这种流量管控带来的影响。

Competition with loss-based congestion control

无论竞争流的拥塞控制算法是BBR还是其他基于丢包的算法，BBR都能最终收敛到一个公平值。即使基于丢包的算法会将缓存填满，ProbeBW仍然会鲁棒地将BtlBw估计为公平值，ProbeRTT仍会估计到一个较高的RTProp来保持公平性。然而，有时候，路由器的缓存太大，大小超过了几个BDP，这会使得持续时间较长的基于丢包的竞争流填满了队列，而得到了一个较大的份额。未来的研究方向包括如何降低这种流的影响！

Conclusion

重新对拥塞控制进行思考会带来很大的好处。BBR并不是基于事件如丢包、或者发生缓存等，这些事件并没有跟拥塞发生非常紧密的联系。BBR起源于Kleinrock关于拥塞的模型，它工作在那个最优点上。通过异步观测估计时延和带宽，BBR绕过了那个不确定魔咒，即时延和带宽没办法同时被估计。我们使用了控制和估计理论最近几年的发展，设计了一个简单的分布式控制环，它可以近似地工作在最优点，在完全利用网络带宽的同时并不会制造长的队列。目前开源的Linux内核TCP已经集成了谷歌的BBR实现，关于更多的细节，请查看附录。

我们将BBR部署在谷歌的B4骨干网，它的吞吐量相比CUBIC提高了几个数量级。我们也将BBR部署在谷歌和YouTube的Web服务器上，它减小了五大洲的传输时延，特别是在发展中地区。BBR仅工作在发送端，它无需对协议、接收方、网络作任何修改，这使得它非常容易部署。BBR只关心RTT和数据传输速率，所以它能够被部署在大多数的传输层协议上。

联系我们：https://googlegroups.com/d/forum/bbr-dev

Acknowledgments

The authors are grateful to Len Kleinrock for pointing out the right way to do congestion control. We are indebted to Larry Brakmo for pioneering work on Vegas2 and New Vegas congestion control that presaged many elements of BBR, and for advice and guidance during BBR's early development. We would also like to thank Eric Dumazet, Nandita Dukkipati, Jana Iyengar, Ian Swett, M. Fitz Nowlan, David Wetherall, Leonidas Kontothanassis, Amin Vahdat, and the Google BwE and YouTube infrastructure teams for their invaluable help and support.

References

1. Abrahamsson, M. 2015. TCP ACK suppression. IETF AQM mailing list; https://www.ietf.org/mail-archive/web/aqm/urrent/msg01480.html.
2. Brakmo, L. S., Peterson, L.L. 1995. TCP Vegas: end-to-end congestion avoidance on a global Internet. IEEE Journal on Selected Areas in Communications 13(8): 1465–1480.
3. Chakravorty, R., Cartwright, J., Pratt, I. 2002. Practical experience with TCP over GPRS. In IEEE GLOBECOM.
4. Corbet, J. 2013. TSO sizing and the FQ scheduler. LWN. net; https://lwn.net/Articles/564978/.
5. Ericsson. 2015 Ericsson Mobility Report (June); https:// www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf.
6. ESnet. Application tuning to optimize international astronomy workflow from NERSC to LFI-DPC at INAF-OATs; http://fasterdata.es.net/data-transfer-tools/case- studies/nersc-astronomy/.
7. Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., Govindan, R. 2016. An Internet-wide analysis of traffic policing. In ACM SIGCOMM: 468–482.
8. Gail, R., Kleinrock, L. 1981. An invariant property of computer network power. In Conference Record, International Conference on Communications: 63.1.1- 63.1.5.
9. Gettys, J., Nichols, K. 2011. Bufferbloat: dark buffers in the Internet. acmqueue 9(11); http://queue.acm.org/detail.cfm?id=2071893.
10. Ha, S., Rhee, I. 2011. Taming the elephants: new TCP slow start. Computer Networks 55(9): 2092–2110.
11. Ha, S., Rhee, I., Xu, L. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. ACM SIGOPS Operating Systems Review 42(5): 64–74.
12. Heidergott, B., Olsder, G. J., Van Der Woude, J. 2014. Max Plus at Work: Modeling and Analysis of Synchronized Systems: a Course on Max-Plus Algebra and its Applications. Princeton University Press.

13. Jacobson, V. 1988. Congestion avoidance and control. ACM SIGCOMM Computer Communication Review 18(4): 314–329.
14. Jaffe, J. 1981. Flow control power is nondecentralizable. IEEE Transactions on Communications 29(9): 1301–1306.
15. Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M.,et al. 2013. B4: experience with a globally-deployed software defined WAN. ACM SIGCOMM Computer Communication Review 43(4): 3–14.
16. Kleinrock, L. 1979. Power and deterministic rules of thumb for probabilistic problems in computer communications. In Conference Record, International Conference on Communications: 43.1.1-43.1.10.
17. Mathis, M., Semke, J., Mahdavi, J., Ott, T. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. ACM SIGCOMM Computer Communication Review 27(3): 67–82.
18. Wikipedia. GPRS core network serving GPRS support node; https://en.wikipedia.org/wiki/GPRS_core_network#Serving_GPRS_support_node_.28SGSN.29.

Related Articles

Sender-side Buffers and the Case for Multimedia Adaptation Aiman Erbad and Charles “Buck” Krasic http://queue.acm.org/detail.cfm?id=2381998

You Don’t Know Jack about Network Performance Kevin Fall and Steve McCanne http://queue.acm.org/detail.cfm?id=1066069

A Guided Tour through Data-center Networking Dennis Abts and Bob Felderman http://queue.acm.org/detail.cfm?id=2208919

Appendix - Detailed Description

A state machine for sequential probing

pacing_gain用于控制相对于BtlBw的数据发送速率，它是BBR的关键！当pacing_gain大于1时，它增加了在外报文数量，减少了数据到达间隔时间，在图1中，它将工作点右移了。当pacing_gain小于1时，它的效果刚好相反，它将工作点往左移。

BBR使用pacing_gain来实现一个简单的异步探测状态机，它不停地探测高可用带宽和低时延。（没有必要去探测带宽是否变低了，因为BtlBw滤波器会自动地做这件事：当一个估计值迟迟没有更新时，它就会丢弃掉这个值重新估计带宽了。类似地，RTprop滤波器也会自动地丢掉过时的估计值）。

如果链路带宽提高了，BBR必须发送得快一些来发现它。相似地，如果实际的往返传播时延变化了，从而改变了BDP，因此BBBR必须将发少一些，使得在外传输报文数量低于BDP从而估计新的RTprop值。因此，为了发现这些动态变化，唯一的办法就是做探测：发快点来检测BtlBw是否提高，发慢点来检测RTprop是否降低。做这些探测的频率、强度、持续时间和结构都取决于已知（在startup和稳定状态）和发送应用的行为（断续的或持续的）。

Steady-state behavior

BBR大多数的时间都处于ProbeBW状态，它使用一种名为gain cycling的方法来探测带宽，从而取得较高的吞吐量，较低的队列实验，并且最终收敛到一个公平带宽值。使用这个方法，BBR周期性地调整pacing_gain值。它使用一个有八个阶段的周期，值分别为：5/4,3/4,1,1,1,1,1,1。每个阶段一般持续时间为估计的RTprop值。这种设计使得BBR首先使用一个大于1的pacing_gain值来探测高可用带宽，并在下一阶段使用一个低于1的值来消耗队列，然后接下来都稳定工作在值为1的阶段。因为ProbeBW旨在调整平均发送速率与可用带宽相等来保持对网络的高利用率，并且同时不制造一个长队列，所以该增益值的平均值为1。注意，尽管pacing_gain的值随着时间变化，但是cwnd_gain的值并不变化，它保持为2，这是因为ACK会被时延（详见章节Delayed and Stretched Acks）。

而且，为了提高公平性，并且在多个BBR流共享同一个瓶颈链路的时候能保持队列较短，BBR随机挑选一个值来作为gain cycling周期的开始值来进入ProbeBW状态（除了3/4）。为什么3/4不能被挑选为初始值呢？因为3/4是为了消耗在它之前的那个5/4产生的队列。当BBR从Drain状态或ProbeRTT状态进入ProbeBW状态时，并不需要消耗队列。使用3/4只会使得在该阶段，链路的利用率为3/4，而不是1。既然3/4没有好处只有坏处，我们干嘛还要使用它呢?

多个BBR流可以周期地进入ProbeRTT状态从而消耗掉瓶颈链路的队列。当BBR的状态不是ProbeRTT时，只要RTProp估计值一段时间内（超过10秒）没有得到更新（如，得到一个更低的RTT观测值），那么BBR就会进入ProbeRTT状态，并且将cwnd值减小为一个非常小的值（4个包）。在保持这个非常小的cwnd值运行至少200ms和一个往返时间后，BBR离开ProbeRTT状态，进入Startup或者ProbeBW阶段（这取决于它对当前管道是否为满的评估）。

我们将BBR设计为将绝大多数的时间（约98%）处于ProbeBW阶段，并且其余的时间处于ProbeRTT阶段。ProbeRTT会至少持续200ms来适应不同的RTT，这对整个BBR流的性能惩罚已经足够小了（大约在2%，200ms/10ms）。）对于流量变化或路由变化来说，RTprop滤波器的窗口大小（10s）足够小，它可以快速收敛。另一方面对于那些交互性的应用（如网页浏览、RPC、视频传输等）来说，这个窗口大小是足够大的，它可以允许这些应用在一小段时间内传输很少的数据，而且可以消耗掉这段瓶颈链路的队列。由于RTprop滤波器会时常更新值，所以BBR并不必经常进入ProbeRTT阶段。通过这种方式，典型地，当多个大流持续地在整个RTprop窗口内发送大量报文时，BBR只需要受到2%的惩罚。

Startup behavior

当一个BBR流开始启动的时候，它开始它的第一个异步探测过程。网络链路的带宽范围非常大（10^12），从几个位每秒到100kMb每秒。为了在这个这么大的范围内学习到BtlBw，BBR的查找是幂次的。这可以非常快地在确定BtlBw的值（在log_2BDP时间内），但这会在最后一步时制造出2BDP的队列。在BBR的Startup阶段后，Drain阶段会消耗掉这些队列。

一开始，Startup以幂次方增加发送速率，每次将它乘以2。为了平滑地快速地进行这个探测的过程，在Startup阶段，BBR的pacing_gain和cwnd_gain被设为2/ln2，这个值是可以使得发送速率每次乘2的最小值。一旦管道满了，cmwd_gain会保证队列的长度不超过(cwnd_gain-1)*BDP。

当BBR处于Startup状态时，若BtlBw估计值在一段时间内不变，那么BBR判断当前管道已满。如果它注意到在几个（3）往返时间内，当它尝试对发送速率进行翻倍并没有引起很大的提升（低于25%），那么它判断它已经到达了BtlBw，所以BBR会离开Startup状态，然后进入Drain阶段。之所以BBR要等待3个往返时间，是因为它要确保BtlBw估计值不发生改变不是因为接收方的接收窗口导致的。BBR等待3个往返时间使得接收方可以调节接收窗口，从而使得BBR发送方可以察觉到BtlBw可以更高：在第一个往返中，接收方窗口自动调节的算法会将接收窗口增大；第二个往返中，发送方将该接收窗口填满；所以，第三个往返中，发送方可以取得更高的发送速率估计值。我们通过YouTube的实验数据来验证了这个阈值（3个往返）。

BBR的Drain状态旨在快速消耗掉Startup状态产生的队列，它是通过将pacing_gain的大小设为其在Startup状态时的大小的倒数来实现的，这会在一个往返时间内就消耗完队列。当在外数据大小与估计的BDP值相等时，这意味着BBR已经将网络中的队列清空了，并且此时的管道仍然是满的。此时BBR离开Drain状态，进入ProbeBW状态。

注意，BBR的Startup阶段和CUBIC的慢启动都是以指数探测瓶颈带宽，它们都在每个往返将发送速率翻倍。然而它们是不同的。首先，BBR在发现可用带宽这方面更加鲁棒，因为它并不是根据丢包或者时延的增加（如CUBIC的Hystart）来离开Startup状态的。第二，BBR平滑地加速它的发送速率，而CUBIC在每一次往返都是突发地发送数据，然后在接下来一段时间内就不发了（就算设置了pacing也一样）。图4展示了BBR和CUBIC在每次收到ack的在外报文数量和RTT。

Reacting to transients

网络路径和网络流量都可能会发生突然变化。为了平滑地、鲁棒地处理这些变化，并且降低在这些情况下的丢包，BBR使用一些列策略在实现它的核心模型。首先，BBR的在外数据目标是cwnd_gain与BDP的乘积，而BBR的cwnd是小心地增大的，cwnd每次增大的大小都不会超过被确认的数据。其次，当发生重传超时时，这意味着发送端会认为所有的在外报文都被丢弃了，BBR保守地将cwnd减小为1，并且仅发送1个报文（就像CUBIC等那些基于丢包的拥塞控制算法一样）。最后，当发送方发现丢包，但此时仍存在在外数据时，在第一个往返时，BBR会暂时地将发送速

率减小为当前的发送速率；在第二个往返以及之后，它会确保发送速率不会超过当前发送速率的两倍。当BBR遇到流量管控或者与其他的流竞争一个缓存只有一个BDP的链路时，这会显著地降低暂时的丢包。

The authors are members of Google's make-tcp-fast project, whose goal is to evolve Internet transport via fundamental research and open source software. Project contributions include TFO (TCP Fast Open), TLP (Tail Loss Probe), RACK loss recovery, fq/pacing, and a large fraction of the git commits to the Linux kernel TCP code for the past five years.