问题出在对“拥塞”的定义上。在1980年设计拥塞控制算法时，认为拥塞等同于丢包。当瓶颈缓冲区很大时，基于丢包的拥塞避免算法会把缓冲区填满，直到出现了丢包，这样的话就会导致缓冲膨胀（bufferbloat, 包在缓冲区的时间很长, 长达几秒）的问题。虽然时延很大，但没有丢包，发送方并不认为网络出现了拥塞。

因此我们需要更换这种基于丢包检测的拥塞控制方法。

每一个TCP连接的每个方向上都会有一条最慢的路径，成为这个链路上的瓶颈。

1. 瓶颈决定了这个连接最大发送速率。
2. 瓶颈导致了排队。只有当链路上包离开的速率大于包到达的速率才会排空队列。

传输性能的两个决定因素：整个链路往返的传播时延Rtprop(round trip propagation time) 和 瓶颈带宽BtlBw(bottleneck bandwidth)

RTT与inflight的大致关系图：

Target:拥塞控制算法理论最优起作用点

基于丢包的拥塞控制算法开始起作用

RTprop

有丢包

满带宽

RTT + BufferSize

BDP

RTT

B

A

C

Inflight

BDP = BtlBw \* Rtprop

当缓存比BDP大不了多少时，处于B阶段时排队带来额外的时延并不明显。而现在缓存大小是BDP的好几个数量级，处于B阶段排队的时延就由几毫秒增加到几秒。

所以目标是：找到Kleinrock’s最佳作用点——BDP这个点，使得inflight = BDP。

目前TCP里面测得的RTT是：一个包发出去的时刻和收到这个包的ACK的时刻之差。

其中δ表示由链路上排队、接收方延时确认、累积确认等带来的额外时间开销；Rtprop是由链路的物理属性决定的，等于路径长度 / 信号传播速率，只有当路由路径改变了才会变。而路径改变的时间窗口远远大于传播时延Rtprop, 因此可以对Rtprop作如下近似，在任意时刻T：

其中是路径改变的时间窗口大小，定为10s。

对BtlBw的一个近似是送达速率（delivery rate）。当ACK到达发送方后，它可以告诉发送方RTT和有多少数据到到了接收方。

因此对BtlBw的一个近似是， 在任意时刻T，有：

其中是6-10个RTT。

由于Rtprop只有当inflight在BDP 线的左边才能测到，而BtlBw只能在右边才能测到，所以它们俩不能同时得到。因此除了维护这个两个参数的估计值外，我们还需要知道当前处于什么状态，我们可以得到哪一个值。如果当前的值已经过期了，又如何回到可测得点来重新获得Rtprop或BtlBW的值。

BBR算法：

1. 收到确认后

每收到ACK后，都计算出一个新的RTT和deliveryRate, 用于更新Rtprop和BtlBW的估计值。

void onAck(packet){

rtt = now - packet.sendtime;

update\_min\_filter(RTpropFilter, rtt);

delivered += packet.size;

deliverd\_time = now;

deliveryRate = (deliverd - packet.deliverd) / (now - packet.deliverd\_time) // 相邻两个ACK期间送达了多少数据/ 相邻两个ACK的时间间隔

if(deliveryRate > BtlBwFilter.currentMax || ! packet.app\_limited){

update\_max\_filter(BtlBwFilter, deliveryRate);

}

if(app\_limited\_util > 0){

app\_limited\_util -= packet.size; //how much data left inflight

}

如果可以发送数据但是没有数据可发，那么app\_limited 为真。这时候不能用deliveryRate来更新BtlBw。由于BtlBw始终都会大于deliveryRate，所以如果新算到的deliveryRate比Btlbw的估计值要大，那么就可以直接更新BtlBw。

1. 发送数据时

cwnd\_gain: 是一个来调节inflight的系数

bdp = BtlBwFilter.currentMax \* RtpropFilter.currentMin;

if(inflight >= **cwnd\_gain** \* bdp){

// already sent enough data, can not send packet now

return;

}

if(now >= nextSendTime){

// can send data now

packet = nextPacketToSend();

if(!packet){

app\_limited\_util = inflight;

return;

}

packet.app\_limited = (app\_limited\_util > 0);

packet.sendtime = now;

packet.delivered = delivered;

packet.delivered\_time = delivered\_time;

sendPacket(packet);

nextSendTime = now + packet.size / (**pacing\_gain** \* BtlBwFilter.currentMax); // now + needed delivered\_time

}

timerCallbackAt(send, nextSendTime);

}

Client

Server

T4

T3

T1

T2

ack = 4

ack = 3

Ack = 2

ack = 1

Seq = 3

Seq = 0

Seq = 1

Seq = 2

Seq = 4

对deliveryRate的理解：以ack=3为例，接收方收到了seq=2后给发送方回复了ack = 3, 那么now = T3, packet.delivered\_time = T2, 即收到ack = 2的时刻。那么 packet.delivered是在收到ack = 2时接收方收到了多少数据，为2MSS。那么在T2-T3期间，接收方新接收了seq=2, 为1MSS. 故deliveryRate = 1MSS / (T3 – T2). 更新delivered = 3MSS, delivered\_time = T3.在发送seq=4的时候，就会用这个两个值来填充packet的属性。

为了防止速率过大造成排队增加，需要进入到ProbeRTT状态：如果RTProp很久(好几秒)没有更新(没有测到更低的RTT)，那么会把发送速率降低到每一个或多个RTT发4个包， 然后在回到原来的状态。通过这种机制来保证公平性和稳定性。

关键参数：pacing\_gain

Pacing\_gain > 1, 增加发送速率也就增加inflight，往BDP的右边移动；pacing\_gain < 1, 减小发送速率也就减小inflight，往BDP的左边移动。BBR用pacing\_gain的增减来探测链路上是有更多带宽还是有更低的RTT。（不需要探测是否有更小的带宽，因为BtlBw用max filter:如果新的测量值小于当前的BtlBw估计值，当老的估计值过期后就会用新的值来更新BtlBw估计值。同理，也不需要探测是否有比当前更大的RTT， 因为Rtprop使用min filter： 如果路由路径变长了使得RTT增加了，那么当前的RTT过期后，就会用这个新的值来更新Rtprop.）如果瓶颈带宽增加了，BBR要通过加快发包速率来发现这一点。同样，如果往返传播时延减小了，BDP会减小，BBR就要减慢发包速率使得inflight小于BDP从而测得实际的RTprop。因此，为了检测是否有这两种变化，就要做实验，也就是发快一点来检测BtlBW增加或者是发慢一点检测RTprop减小。

BBR状态机

1. Startup状态：

为了短时间内探测到带宽大小，采用二分查找。这样的话用个往返时间就可以达到BW， 但是在最后一步会inflight = 2BDP， 会产生队列。因此需要进入Drain状态排空队列。在此期间，pacing\_gain = cwnd\_gain = . 为了检测什么时候应该进入Drain状态，在Startup期间，如果有3个连续RTT内，期望 deliveyRate加倍实际上deliveryRate没有增加多少（小于25%），那么就认为已经inflight已经到了BDP，于是退出startup, 进入Drain状态。

1. Drain状态：

Pacing\_gain = . 当inflight = BDP， 离开Drain状态，进入ProbeBW状态。

1. Steady 状态:

BBR流绝大部分时间（98%）都处于ProbeBW和ProbeRTT状态。

1. ProbeBW状态

ProbeBW状态采用gain cycling方法来探测带宽。一个gain cycling是一个8阶段的循环，这8个阶段pacing\_gain分别设置为这一些值：5/4, 3/4, 1,1,1,1,1,1，每个阶段持续一个RTprop. 先设一个大于1的值来看看有没有更多带宽，然后设置一个小于1的值排空产生的队列，之后一直为1。在这期间cwnd\_gain 始终为2。

1. ProbeRTT状态

当多个BBR流处于ProbeRTT状态时，瓶颈路径上的队列会排空。如果处于其他状态时，Rtprop在10s内没有更新（没有测得一个更小的RTT）， 那么BBR就会进入ProbeRTT状态，并把cwnd降为很低的值（4个包）。持续至少200ms加一个RTprop的时间后，BBR离开ProbeRTT状态，进入到Startup或ProbeBW状态。

ProbeRTT的持续时间需要满足以下条件：1.足够长（200ms）使得具有不同RTT的不同流的ProbeRTT状态可以重叠，也就是多个流同时处于ProbeRTT状态；2. 不能太长，ProbeRTT状态会使得性能降低，ProbeRTT差不多占2%（200ms/10s）。另外还需要合理设置更新Rtprop的时间窗口大小Wr（10s）。Wr不能太长，因为它需要保证当路由改变时能够迅速收敛到正确的值；同时也不能太短，因为这个时间窗口应该包含交互式应用没多少数据要发的时候，这样就会排空瓶颈路径上的队列，不需要额外地进入ProbeRTT状态。

其他策略：

1. Cwnd = cwnd\_gain \* BDP
2. 如果出现RTO，cwnd = 1
3. 如果有丢包，但是inflight > 1, 第一个RTT， sending rate = deliveryRate;之后sending rate <= 2 \* deliveryRate

总结：

BBR通过探测瓶颈链路的带宽（bottleneck bandwidth）和整个链路的往返传播时延(round trip propagation time)来刻画链路的传输能力。

BBR主要用pacing rate来控制发送速率，cwnd成为次要控制因素。Pacing rate控制着发送速率比瓶颈带宽快或者是慢。Cwnd 设为 \* BDP（bandwidth delay product）。

当BBR连接开始时，进入STARTUP模式，为了快速探测到瓶颈带宽，采用指数增长方式（每个RTT发送速率加倍，和slow start一样）。不同之处在于它如何判断何时应该退出该状态，它不是到填满了缓存（比如基于丢包的拥塞控制算分cubic，reno），或者是时延到某一个阈值（比如Hystart），而是这样判断的：发现估计带宽不再增加。这时候BBR会退出STARTUP模式，进入DRAIN模式，降低pacing rate。

然后进入稳定状态，在PROBEBW模式，先pacing快些来探测有没有更多带宽，然后pacing慢些把可能存在的队列清空；然后按照估计带宽发送。然后在有必要的情况下，进入PROBE\_RTT来检测往返传播时延是否发生变化。