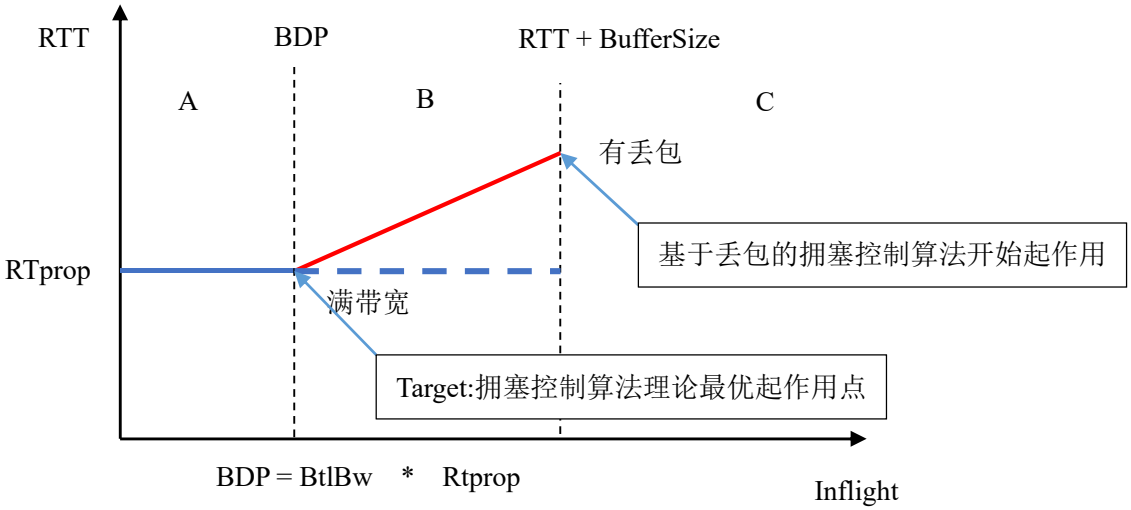
问题出在对“拥塞”的定义上。在 1980 年设计拥塞控制算法时，认为拥塞等同于丢包。当瓶颈缓冲区很大时，基于丢包的拥塞避免算法会把缓冲区填满，直到出现了丢包，这样的话就会导致缓冲膨胀（bufferbloat，包在缓冲区的时间很长，长达几秒）的问题。虽然时延很大，但没有丢包，发送方并不认为网络出现了拥塞。
因此我们需要更换这种基于丢包检测的拥塞控制方法。

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：



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。

$$\mathrm{RTT_t} = RTprop_t + \ \delta_t$$

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：

$$\widehat{\mathrm{RTprop}} = \mathrm{RTprop} + \min(\delta_t) = \min(RTT_t), \forall t \in [T - W_r, T]$$

其中$\mathrm{W_r}$ 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。

$$\mathrm{deliveryRate} = \frac{\Delta \mathrm{delivered}}{\Delta \mathrm{t}} \leq \mathrm{BtlBw}$$

因此对 BtlBw 的一个近似是， 在任意时刻 T，有：

$$\widehat{\mathrm{BtlBw}} = \max(deliveryRate), \quad \forall t \in [T - W_b, T]$$

其中w_b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。

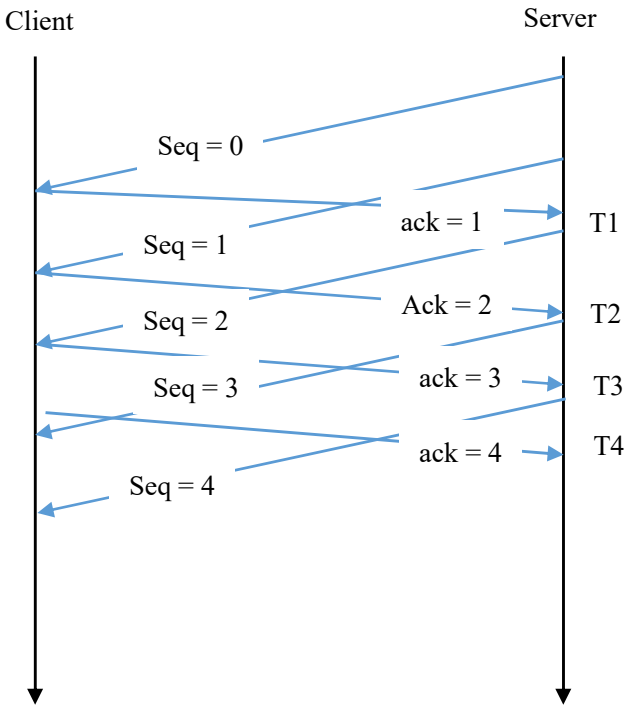
2. 发送数据时

   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);
}
```



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$\log_2 BDP$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 = $\frac{2}{\ln 2} = 2.885$. 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。

2． Drain 状态：

Pacing_gain = $\frac{\ln 2}{2} = 0.3465$. 当 inflight = BDP， 离开 Drain 状态，进入 ProbeBW 状态。

3． Steady 状态:

BBR 流绝大部分时间（98%）都处于 ProbeBW 和 ProbeRTT 状态。

a. 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。

b. 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α * 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。