目录结构

一、概述

1.1 介绍

在 1980 年代，链路 buffer 比较小，TCP 中的拥塞控制算法将丢包作为作为拥塞事件[8]。然而，随着链路 buffer 越来越大，使用基于丢包的拥塞控制算法会导致较大的排队，增加了传输延时。2016 年 google 提出 Bottleneck Bandwidth and Round Trip Time(BBR)算法，BBR 使用传输速率样本来估计网络可用带宽，使用 RTT 样本估计链路的实际 RTT，使用 Bandwidth Delay Product(BDP)控制飞行的数据，来达到尽量避免排队延时的目的。

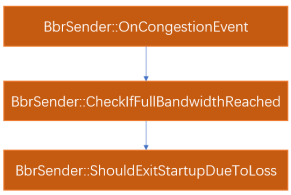
BBR v1 算法细节在[9]中已经描述的较为详细。本文重点如下:1.对于 Cubic 和 BBR v1 算法；2.描述 BBR v2；3.chrome 中 BBR v1 和 v2 代码解析；4.多种环境下 BBR 测试实验。

1.2 符号、缩写意义

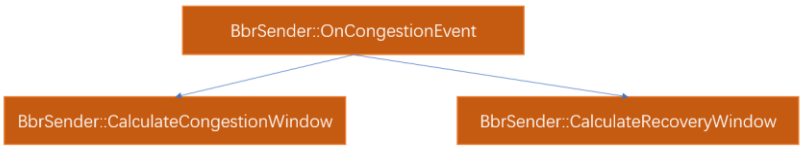
pacing rate：当前时刻发送数据速率，单位为 bps
inflight packets：已经发送但是没有收到 ack 的数据包
unacked packets：与 inflight packets 相同
cwnd：拥塞窗口
发送算法：BBR v1 以及 BBR v2
BBR：指 BBR v1
初始阶段：cubic 中的慢启动、BBR 中的 StartUp 阶段
稳定阶段：cubic 中慢启动之外的阶段、BBR 中 ProbeBw+ProbeRTT

函数调用图说明:

下图表示 BbrSender::OnCongestionEvent 调用 BbrSender::CheckIfFullBandwidthReached，BbrSender::CheckIfFullBandwidthReached 调用 BbrSender::ShouldExitStartUpDueToLoss



下图表示 BbrSender::OnCongestionEvent 从左到右依次调用 BbrSender::CalculateCongestionWindow 和 BbrSender::CalculateRecoveryWindow。



二、BBR v1 与 cubic

本节重点对比 BBR 算法和 cubic 算法，来说明 BBR 是怎样来减少传输延时的。Note：cubic 算法中，我们不考虑 cubic 中的快速收敛和 TCP 友好机制，与[10]中的版本进行对比。

我们将 cubic 中的慢启动、BBR 中的 StartUp 阶段叫做初始阶段。将 cubic 中慢启动之外的阶段、BBR 中 ProbeBw+ProbeRTT 叫做稳定阶段。

2.1 cubic 与 BBR 部分不同点

	Cubic	BBR
计算传输速率	无	使用[2]中方法
计算发送码率	窗口大小除以平滑后的 RTT[3]	根据传输速率调整发送码率
计算 RTT	计算指数加权平均 RTT、RTT 偏差	Cubic 基础上，还使用时间窗口计算窗口内最小 RTT

cubic 计算发送码率使用窗口(期望能够在一个 RTT 发出的数据量)除以 RTT。

2.2 初始阶段对比

	Cubic	BBR
初始阶段	每收到一个 ack，窗口增加 1，得到的效果是，每个 RTT 之后，窗口翻倍	发送速率始终等于最大传输速率的两倍，拥塞窗口始终等于 BDP 的两倍
退出初始阶段条件	发现丢包	超过三个 RTT 时间内，传输速率增长少于 1.25 倍
退出初始阶段后动作	cwnd=(1-β)cwnd，进入稳定阶段	进入 drain 阶段，抽干队列，发送速率等于可用带宽的 $1.0 / 2.885$。发现 inflight 小于 BDP 之后进入稳定阶段

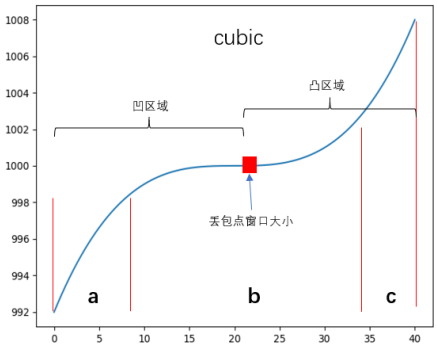
可以看出，在初始阶段 Cubic 与 BBR 都是指数增长，但是退出初始条件并不相同，cubic 发现丢包才退出初始条件，此时已经产生大量排队。与 cubic 相比，BBR 不等丢包才退出初始阶段，而是用传输速率的增长来判断排队情况，如果传输速率增长较慢(使用三个 RTT 来增强判断的鲁棒性)，说明在产生排队，应该退出初始阶段。

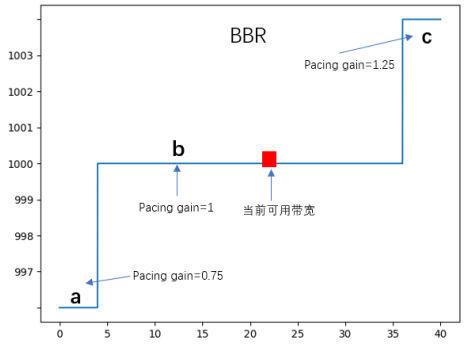
退出初始阶段之后，cubic 知道链路有拥塞，减少窗口，然后进入稳定阶段。BBR 在退出初始阶段之后，由于传

输速率增长变慢，推测链路有排队，从而将发送速率降低为测得带宽的 1.0 / 2.885 倍来抽干队列，进而减少在后面的传输中的排队延时。

2.3 稳定阶段对比

在稳定阶段，可以粗略地将 cubic 和 BBR 的稳定阶段分成三个部分：遇到拥塞之后使用较低的发送速率、保持当前发送速率、探测新的可用带宽。这三个阶段分别位于下图 a，b，c。注意，cubic 中 a、b、c 的划分是任意的，只做讲解使用。





下表是 cubic 和 BBR 在这三个阶段的对比

	Cubic	BBR
“探测新的可用带宽”行为	cwnd 在一个 RTT 内增加 cubic_function(Now+RTT)	pacing_gain = 1.25*Bandwidth
“保持当前发送速率”行为	cwnd 在一个 RTT 内增加 cubic_function(Now+RTT)	pacing_gain = Bandwidth
“遇到拥塞之后使用较低的发送速率”行为	cwnd 在一个 RTT 内增加 cubic_function(Now+RTT)	pacing_gain = 0.75*Bandwidth
退出“探测新的可用带宽”条件	发现丢包	一个 RTT 后，发现丢包或者 inflight>2*BDP
“保持当前发送速率”持续时间	根据 cubic 曲线变换	6 个 RTT
“遇到拥塞之后使用较低的发送速率”持续时间	根据 cubic 曲线函数计算，由于是三次曲线，这个时间非常短	1 个 RTT 或者 inflight<BDP

我们将上图中红色方块点称为基准点。可以看出，cubic 以丢包时窗口为基准点，BBR 以可用带宽为基准点。BBR 利用可用带宽和 BDP 理论来控制发送的数据，来达到尽量避免排队，同时得到高吞吐量的目的。

在 ProbeBw 阶段，cwnd_gain 始终大于 1，从而 inflight>BDP，这意味着链路始终存在排队。排队的存在意味着 RTT 样本比无排队时 RTT 样本大，为了得到无排队的 RTT 样本，BBR 使用 ProbeRTT 阶段来得到无排队时 RTT 样本。

ProbeRTT 阶段，pacing_gain=1，cwnd_gain=1，cwnd 始终等于 4。ProbeRTT 阶段以较低的发送窗口来发送数据，抽干队列，从而得到无排队时的 RTT 样本。这个阶段持续 200ms。

三、BBR v2

在 BBR v2 中，将 ProbeBw 分为三个阶段: Down、Cruise、Up，分别对应 BBR v1 中增益等于 0.75、1、1.25 的三个情况。这一节介绍 BBR v2 中对 BBR v1 中的问题的一些改进方案。

3.1 BBR 问题

1.在 shallow buffer 场景，BBR 的 2 倍 BDP 探测机制会导致大量丢包;

2.ProbeRTT 阶段，直接将窗口降到 4，会导致吞吐量突然大量降低;

3.与 TCP 流共存时，BBR 流会被抑制;

4.RTT 公平性问题：当多个 BBR 流共享链路带宽，RTT 较大的 BBR 流会抑制 RTT 较小的 BBR 流。

3.2 shallow buffer 场景

对于第一个问题，BBR 将丢包情况也考虑在内。

具体来说在 StartUp 和 ProbeBw 的 Up 阶段，收到 ack 之后，检测丢包情况，如果丢包超过预设阈值，那么退出此阶段(详见 5.2.2.5 节)。

通过这个机制，在 shallow buffer 场景下，BBR v2 通过检测丢包来切换状态，这意味着再这种场景下，BBR v2 退化为基于丢包的方法。

3.3 ProbeRTT 阶段改进

对于第二个问题，BBR v2 进入 ProbeRTT 之后，目标窗口会设置为 BDP 的 0.5 倍，而不是将窗口设置为 4，避免吞吐量突降。

3.4 与 TCP 流共存问题

由于 BBRv1 的指数增长速率机制，BBRv1 在 StartUp 阶段并不会被 TCP 流抑制。BBR v2 的提升主要在 ProbeBw 阶段。

BBR v2 将一个 Down、Cruise、Up 过程称为一个 cycle（详见 5.2 节）。BBR v2 通过两个机制来避免被 TCP 流抑制：进入 Down 之后(即重新开始一个 cycle)，设置一个随机时间 probe_wait_time 以及 RTT 个数 probe_bw_probe_max_rounds，从这个 cycle 开始算，如果已经经过了 probe_bw_probe_max_rounds 个 RTT 或者以及过去了 probe_wait_time，那么进入 Up 阶段，增加发送速率。

BBR v2 通过设置两个阈值，避免太长时间没有增加发送速率，进而避免被 TCP 流抑制。

3.5 其他改进

3.5.1-3.5.4 这四个改进都是受 cubic 的启发。下面先介绍这四个改进，在 3.5.5 节将这几个机制与 cubic 对比。

3.5.1 带宽突降场景

例如，假设 RTT 不变，带宽从 12 Gbit/sec 降到 12 Mbit/sec，由于 BBR v1 使用 BDP 作为上限，当检测到带宽变化之后，发包可能会 1000 pkt/ms 从降为 1 pkt/ms。BBR v2 认为，发包不应该突降，因此 BBR v2 设置 inflight_lo，在带宽突降到检测到带宽突降这个短时间，作为窗口的上限，缓慢降低拥塞窗口。inflight_lo 在丢包后才更新(注意，如果不丢包，inflight_lo 对窗口无作用)，更新规则为 inflihgt_lo=max(inflight_latest,inflight_lo*0.7)，参见下图

3.5.2 inflight_hi 与快速收敛机制

BBR v2 还增加了 inflihgt_hi 变量，inflight_hi 变量更新如下：

1. 在退出 inflight_hi 时，inflight_hi 设置为退出 StartUp 时的拥塞窗口；
2. 在 ProbeBw 的 up 阶段，inflight_hi 指数增加。在 up 阶段检测到丢包之后，inflight_hi 设置为丢包时的窗口。

在 cubic 中，如果丢包时的窗口小于上一次丢包时的窗口，说明网络中有新的流进入，cubic 通过快速收敛机制让出一部分带宽，让新的流能够加入。快速收敛机制中，Wmax=0.9*丢包时带宽。

在 BBR v2 中，在 cruise 阶段，窗口的上限为 0.85*inflight_hi，通过这种机制来实现类似 cubic 中的快速收敛机制。

3.5.3 估计带宽方式

BBR v1 中使用窗口最大滤波器（Kathleen Nichols 算法），窗口大小为 10 个 RTT。BBR v2 中最大滤波器不再使

用算法。在 ProbeBw 阶段之前，估计的带宽是从 StartUp 到目前的传输速率最大值；进入 ProbeBw 阶段之后，估计的带宽是上上次进入 Down 阶段到现在的最大值，详见 5.2.2.2 节。

3.5.4 最小 RTT 更新规则

在 BBR v1 中，链路 RTT 使用时间窗口滤波器，窗口大小为 10s，如果 RTT 在 10s 内没有发生变换，强制更新链路 RTT。

在 BBR v2 中，在 ProbeBw 阶段之前，链路 RTT 是从 StartUp 到目前的 RTT 样本最小值；在 ProbeBw 之后，链路 RTT 是 ProbeBw 中 RTT 样本的最小值，同时，每进入 ProbeRTT 时，都要强制更新链路 RTT。

3.5.5 总结

	Cubic	BBR v1	BBR v2
带宽估计更新周期	不估计	始终是 10 个 RTT	StartUp 为一个阶段，probeBw 中每两个 cycle 更新一次
最小 RTT 更新周期	不估计	始终是 10s	StartUp 为一个阶段，ProbeBw 之后周期可以自定义，默认 10s
快速收敛机制	Wmax=0.9*Cwnd_at_lost	无	在 ProbeBw 的 Cruise 阶段中，headroom=0.85*inflihgt_hi(可以将 headroom 看作 cubic 中 Wmax，inflight_hi 看作 cubic 中 cwnd_at_lost)
TCP 友好机制	在丢包后的任意 RTT 中，cwnd 最小增长 $3\frac{\beta}{2-\beta}\frac{t}{RTT}$	无	从进入 ProbeBw Down 开始，如果探测超过一个随机时间，或者超过 63 个 RTT，则进入 Up 阶段
初始阶段丢包处理	设置 Wmax，进入 cubic 曲线周期变化区域	无	设置 inflight_hi，进入先进入 Drain 抽干队列，然后进入 ProbeBw 周期
稳定阶段丢包后处理	退出当前 cubic 曲线周期。令 ssthresh=0.8* Cwnd_at_lost,进入下一个 cubic 曲线周期	首先保存 cwnd，进入丢包恢复。cwnd=inflight+acked,然后在丢包后的第一个 RTT 内 cwnd=cwnd-packets_lost，第一个 RTT 之后 cwnd 增长速度不大于指数增长，直到没有丢包，然后恢复丢包前的 cwnd。	退出 ProbeBw 的 Up 阶段，设置 inflight_hi=当前 cwnd，进入 Down 阶段
带宽突降之后导致连续丢包后，cwnd 变化	ssthresh=0.8* ssthresh	同稳定阶段丢包处理	Inflight_lo=0.7*inflight_lo
探测可用带宽时窗口变化	根据 cubic 曲线变化	无特殊操作	Inflight_hi 指数增加

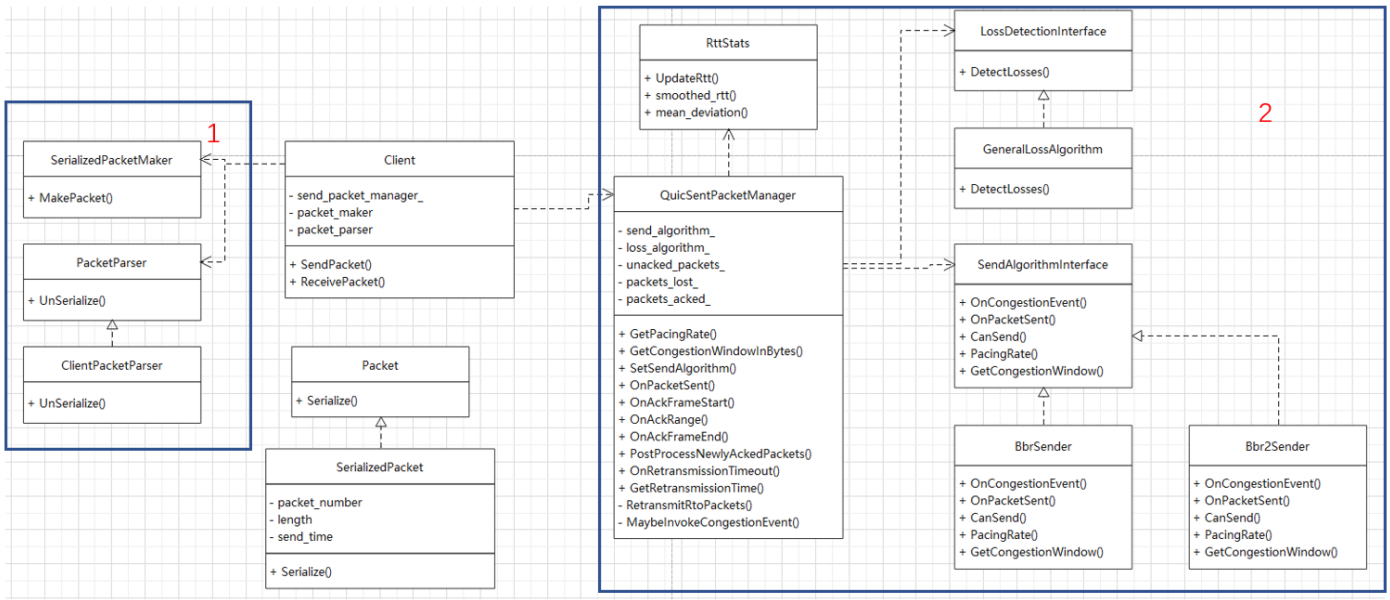
总体可以看出，chrome 的 BBR v2 中的很多修改，都是借鉴了 cubic 中的机制。可以近似认为，chrome BBR v2=BBR v1+cubic。

四、chrome 测试代码结构

测试代码中服务器的主要功能是:收到包之后无条件回包。客户端收到回包之后认为这个数据已经到达了服务器。比较简单，这里不做介绍，下面内容都针对客户端。

4.1 代码总体结构

代码总体结构如下图所示:



客户端包含以下模块：

1. 制作包(SerializedPacketMaker)和解析包(ClientPacketParser)
2. 发包管理模块(QuicSendPacketManager)

制作包只使用简单的 ascii 数据作为数据包内容，每个包中携带包号信息。解析包解析收到的服务器发送的包中的包号。下面重点介绍 QuicSendPacketManager 模块，其中 SendAlgorithm、BbrSender、Bbr2Sender 放在下一章介绍。

4.2 Client

客户端每收到一个包之后，调用 send_packet_manager，后者运行 BBRv1 或者 BBRv2 更新 BBR 算法状态并反馈发送速率和拥塞窗口大小。

客户端根据 QuicSendPacketManager 反馈得到 pacing rate 和 cwnd。当前时刻是否发送包同时受这两个参数影响，pacing rate 控制发包速率，设置下一个包的发送时间。

计算公式如下：

$$Next\_packet\_send\_time = Last\_packet\_send\_time + \frac{Pacing\ rate}{Packet\ size}$$

需要注意的是，发包也需要一定的时间。在实际中，发 1500byte 的包消耗的时间最多为 50us，因此可以发包的最大速率为 $1500*8/(5*10^{-5}) \approx 240M/s$。

cwnd 是拥塞窗口，只有当当前 packets_inflight<cwnd 才能发送下一个包。任意时刻可以发包的条件为：

Now>next_packet_send_time and packets_inflight<cwnd

4.3 RTTStats 与 LossDetectionInterface

4.3.1 RTTStats

在每次收到 ack 之后，调用 RTTStats 更新指数加权平均 RTT(smooth_RTT)，以及的指数加权平均偏差(mean_deviation)。这个信息在重传超时中有用。

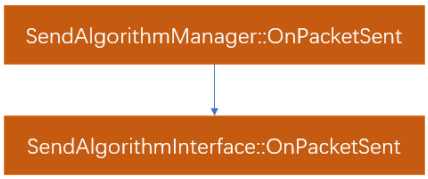
4.3.2 LossDetectionInterface

LossDetectionInterface 定义了丢包检测的一般接口，在 chrome 中，有两个对其的实现：GeneralLossAlgorithm 和 UberLossAlgorithm。前者的实现使用包阈值和时间阈值，方法同 RFC9002。后者本文不做阐述。

4.4 QuicSendPacketManager

4.4.1 发送包

客户端每发送一个包之后，会调用 send_manager->OnPacketSent()，发包管理器处理发包事件。



send_manager 接下来做两件事情:1.调用发送算法的 OnPacketSent；2.将刚发送的包的信息加入到 unacked_packets_中。

```
bool QuicSentPacketManager::OnPacketSent(
    SerializedPacket* mutable_packet,
    QuicTime sent_time) {
  const SerializedPacket& packet = *mutable_packet;
  QuicPacketNumber packet_number = packet.packet_number;

  bool in_flight = true;

  send_algorithm_->OnPacketSent(sent_time, unacked_packets_.bytes_in_flight(),
                                packet_number, packet.encrypted_length);

  unacked_packets_.AddSentPacket(mutable_packet, sent_time,in_flight);
  // Reset the retransmission timer anytime a pending packet is sent.
  return in_flight;
}
```

4.4.2 收到 ack 后操作

确定收到了回包之后，做两件事情:

1. 首先调用 send_manager 的三个收包函数 OnAckFrameStart、OnAckRange、OnAckFrameEnd(google 团队在设计这三个动作时是针对 quic 协议的，测试代码中没有修改接口的名字，其实在测试代码中这三个操作可以合成一个动作)，通知 send_manager 收包事件，并进行处理。
2. 根据发送算法的反馈得到新的 cwnd 和 pacing_rate;

```
void Client::OnPacketReceived(){
    SerializedPacket* packet = (SerializedPacket*)packet_parser_.UnSerialize(receive_buf_);
    QuicTime time =  clock_->Now();
    send_manager_->OnAckFrameStart(packet->packet_number,time);
    send_manager_->OnAckRange(packet->packet_number,packet->packet_number+1);
    send_manager_->OnAckFrameEnd(time,packet->packet_number);

    // Get feed back
    send_control_.pacing_rate = send_manager_->GetPacingRate().ToBitsPerSecond();
    send_control_.cwnd = send_manager_->GetCongestionWindowInBytes()/DEFAULT_PACKET_SIZE;
    // debuger_->PrintPacketInfo(packet);
}
```

4.4.2.1 OnAckFrameStart:

更新 RTT(RTTStats)

4.4.2.2 OnAckRange:

将新 ack 的包加入到 packet_acked

4.4.2.3 OnAckFrameEnd:

1.将新 ack 的包从 packet_uacked 中去除；2.检测是否有丢包；3.通知发送算法收包事件

注意：packet_acked 和 packets_lost 都只针对此次 ack 事件，处理完此次 ack 之后会清空，而 uacked_packets 是全局的

4.5 丢包与超时事件

4.5.1 丢包

使用丢包检测算法检测到丢包之后，会记录到 packets_lost，在 quic 中，会将 quic 中可重传帧选出来，放在后面的包中进行重传，本测试代码忽略这部分功能，每次发包的内容都是相同的。

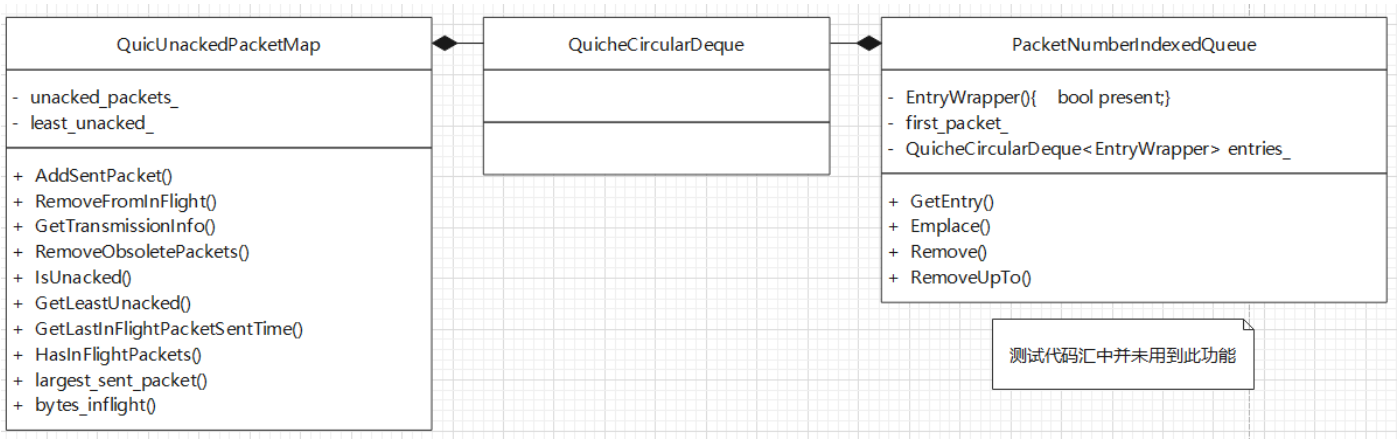
4.5.2 超时事件

当网络环境非常差，客户端可能长时间无法收到服务器的回包(当 unacked_packets 中的包全部丢失，那么永远收不到 ack，而此时客户端也无法发包，因为 uacked_packets=cwnd)，因此当客户端很长时间没有收到 ack，就认为 uacked_packet 全部丢失，从而可以接着往下发包(unacked_packets<cwnd)。

```
void Client::ReceivePacket(){
    memset(receive_buf_,'\0',SEND_BUF_SIZE);
    socklen_t addrlen = sizeof(struct sockaddr_in);
    int num =recvfrom(sender_socket_,receive_buf_,SEND_BUF_SIZE,MSG_DONTWAIT,(struct sockaddr*)&remote_addr_,&addrlen);
    if (num > 0){
        OnPacketReceived();
    }
    else if(clock_->Now()>send_manager_->GetRetransmissionTime()){
        send_manager_->OnRetransmissionTimeout();
    }
}
```

4.6 使用的数据结构

这里介绍 chrome bbr 模块中的数据结构是 QuicUnackedPacketMap、QuicheCircularDeque、PacketNumberIndexedQueue。它们的关系如下所示:



QuicheCircularDeque 非常类似 std::deque，接口基本相同。内存分配上的不同地方是，std::deque 分配的内存用完之后，内存翻倍[7]。QuicheCircularDeque 内存使用完之后，内存增加原来的 1/4。

QuicUnackedPacketMap、PacketNumberIndexedQueue 都是在 QuicheCircularDeque 上面的一层封装，增加两个功能：1.O(1)时间访问位置元素，类似 map(利用 deque 顺序存储的特性)；2.从任意位置删除元素。

对于第一个功能，QuicUnackedPacketMap、PacketNumberIndexedQueue 实现类似，都维护一个基准包号(QuicUnackedPacketMap 是 leaset_unacked_，PacketNumberIndexedQueue 是 first_packet_)，从而对于任意包号可以用包号减去基准包号来访问。

对于第二个功能，PacketNumberIndexedQueue、QuicUnackedPacketMap 实现有所不同，PacketNumberIndexedQueue<T>对每个 T 都放入 EntryWrapper 这个结构体，再入 deque，EntryWrapper 有一个 present 位，指示 T 是否仍然有效，如果 present 为假，那么认为元素已经被删除(尽管还在内存中)。真正从内存中去除元素是操作是 RemoveUpTo()，会调用 deque 的 pop_front()操作。

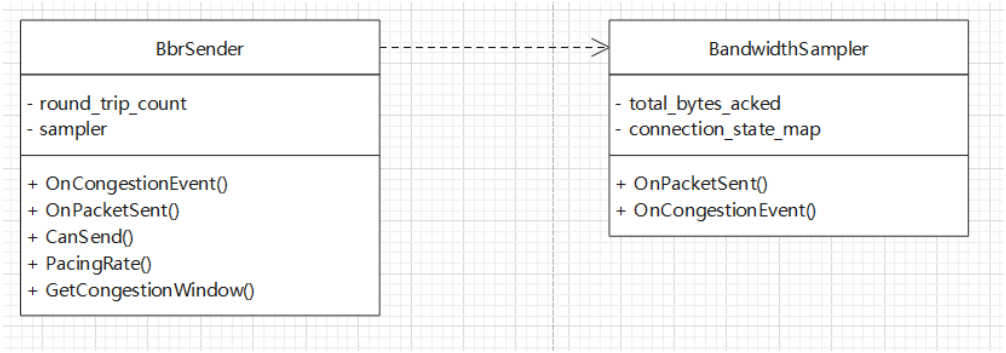
QuicUnackedPacketMap<QuicTransmissionInfo>中 QuicTransmissionInfo 本身有类似 present 位的东西来判断某个 QuicTransmissionInfo 对象是否仍然有效，这里不做介绍。

五、BBR 代码

这一节主要介绍 chrome 中 SendAlgorithmInterface BBR v1 v2 代码。SendAlgorithmInterface 提供了两个主要接口 OnCongestionEvent()、OnPacketSent()分别处理发送包和收到 ack 的事件来更新发送算法的状态。提供 GetCongestionWindow()和 PacingRate()两个接口返回当前 BBR 希望的 cwnd 和发送速率大小。

5.1 BbrSender

下面只列出了主要的接口



5.1.1 一些参数

这里只介绍[1]中没有介绍的参数。

int FLAGS_quic_bbr2_default_StartUp_full_loss_count = 1;

int FLAGS_quic_bbr2_default_loss_threshold = 1;

这两个变量用于 STARTUP 阶段丢包检测，如果再 STARTUP 阶段丢包比较多，那么退出 STARTUP。这两个变量的具体值都是我自己设置的，flag 名称中含有 bbr2，意味着这是对 bbr v1 的改进，因为 bbr v1 中没有 STARTUP 阶段丢包检测。

```
bool BbrSender::ShouldExitStartupDueToLoss(
    const SendTimeState& last_packet_send_state) const {
  if (num_loss_events_in_round_ <
          GetQuicFlag(FLAGS_quic_bbr2_default_startup_full_loss_count) ||
      !last_packet_send_state.is_valid) {
    return false;
  }

  const QuicByteCount inflight_at_send = last_packet_send_state.bytes_in_flight;

  if (inflight_at_send > 0 && bytes_lost_in_round_ > 0) {
    if (bytes_lost_in_round_ >
        inflight_at_send *
            GetQuicFlag(FLAGS_quic_bbr2_default_loss_threshold)) {
      stats_->bbr_exit_startup_due_to_loss = true;
      return true;
    }
    return false;
  }

  return false;
}
```

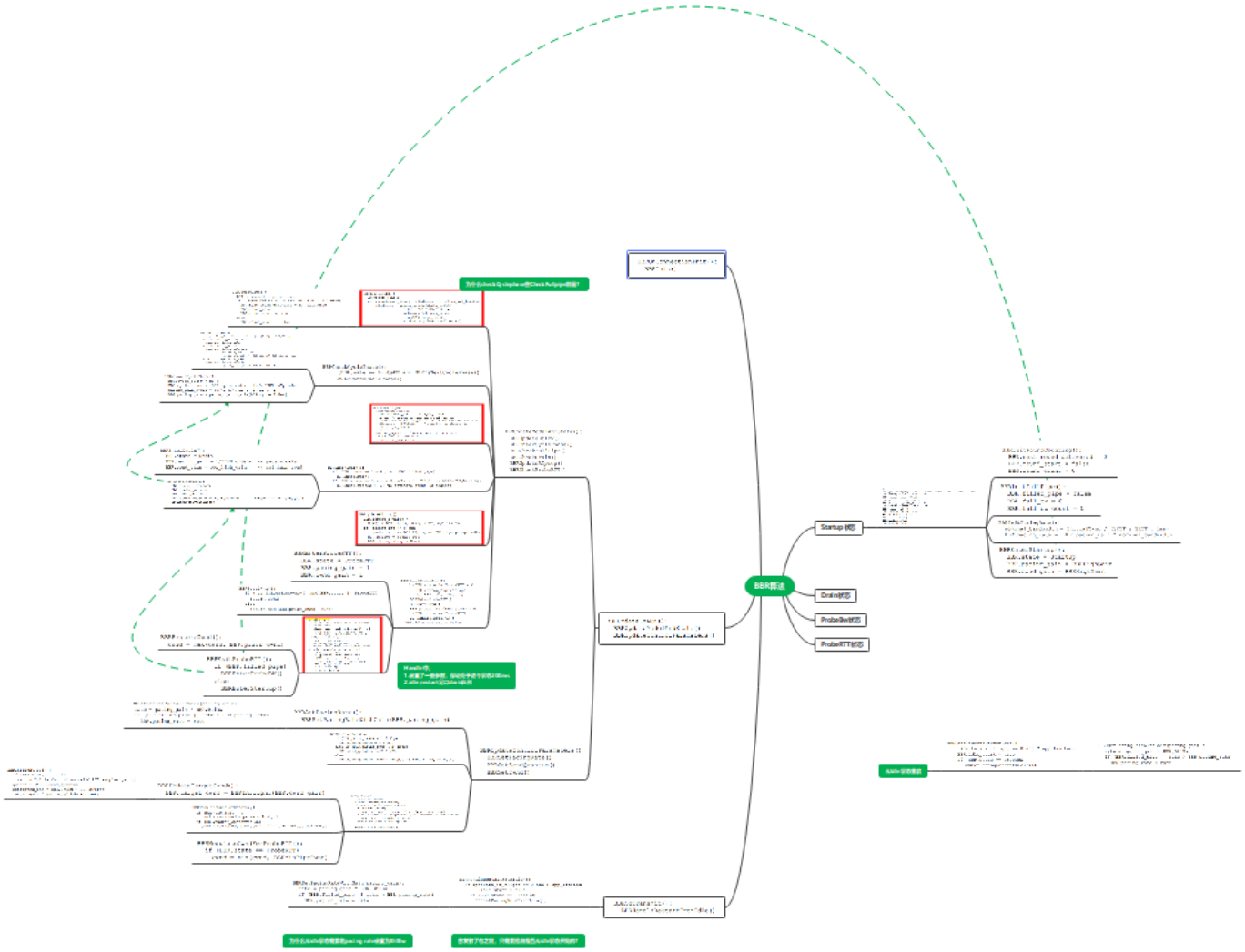
5.1.2 OnPacketSent()

每发送一个包，将这个包加入到 BandwidthSampler 中，BandwidthSampler 记录用于计算传输速率样本的信息：这个包发送时总的 ack 数量。

```
void BbrSender::OnPacketSent(QuicTime sent_time,
                             QuicByteCount bytes_in_flight,
                             QuicPacketNumber packet_number,
                             QuicByteCount bytes) {

  if (stats_ && InSlowStart()) {
    ++stats_->slowstart_packets_sent;
    stats_->slowstart_bytes_sent += bytes;
  }

  last_sent_packet_ = packet_number;

  if (bytes_in_flight == 0 && sampler_.is_app_limited()) {
    exiting_quiescence_ = true;
  }
  sampler_.OnPacketSent(sent_time, packet_number, bytes, bytes_in_flight);
}
```

5.1.3 OnCongestionEvent()

每次收到 ack 或者检测到丢包事件，调用 BbrSender::OnCongestionEvent，首先调用 BandwidthSampler::OnCongestionEvent 得到传输速率样本，传输速率样本的计算方法见[2]。得到传输速率样本后，剩余代码与[1]中伪代码基本类似，这里做了如下整理：

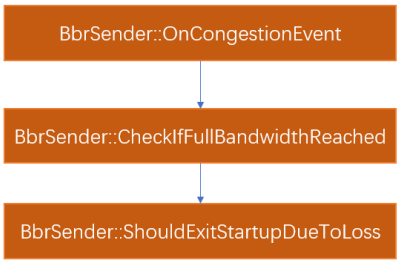


5.1.4 丢包处理

接下来我们看一下 BBR v1 丢包时做的一些处理。

5.1.4.1 在 StartUp 阶段丢包

当丢包超过一定数量，则退出 STARTUP 阶段，认为发送速率已经达到了可用带宽。注意，丢包数以 RTT 为单位。调用栈如下

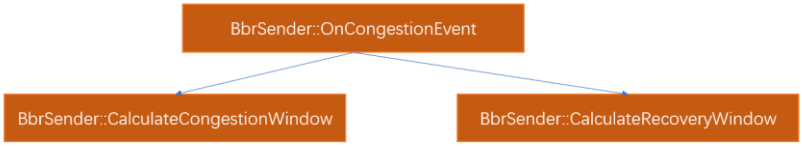


5.1.4.2 丢包对发送速率的影响

由于测试代码中并不检测 over_shooting，因此丢包对测试代码中发送速率无影响

5.1.4.3 丢包对拥塞窗口的影响

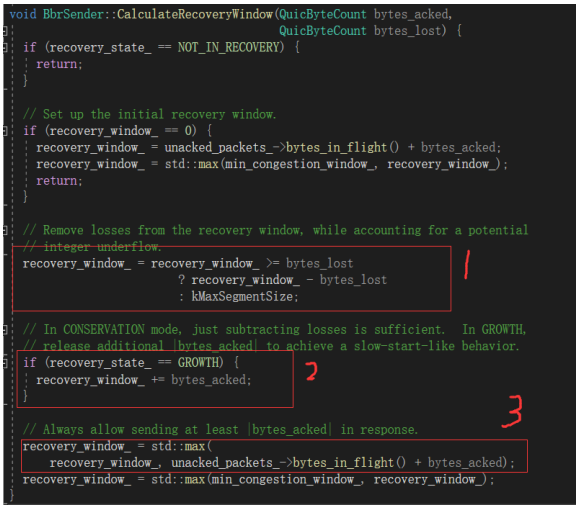
调用栈如下



在没有丢包的情况下 BbrSender::CalculateCongestionWindow()之后起作用(congestion_window_)，在有丢包的情

况下 BbrSender::CalculateRecoveryWindow()起作用(recovery_window_)。

第一次检测到丢包之后，恢复状态进入 CONSERVATION 状态，拥塞窗口每次减少丢包的数量(网络目前无法承受这些数据)，并且不增加拥塞窗口大小(缓解网络压力)，这个状态持续一个 RTT。在一个 RTT 之后，进入 GROWTH 状态，在 GROWTH 状态中，会将拥塞窗口增加 bytes_acked(这些包可以收到，因此拥塞窗口增加。注意，在恢复阶段，窗口最小值为处理这次 ACK 前的 inflight。

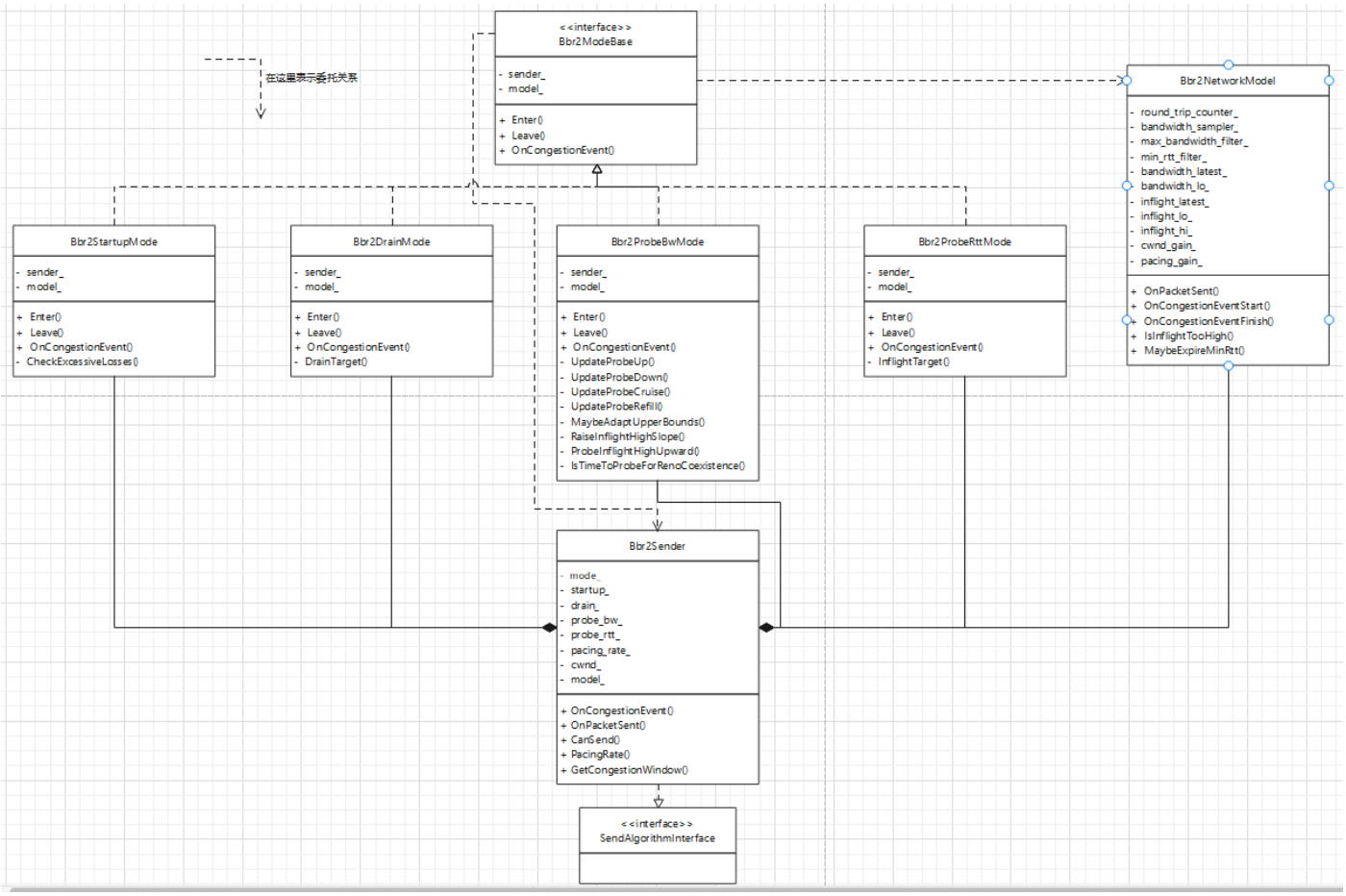


在 GROWTH 状态中，如果不再丢包，那么进入 NOT_IN_RECOVERY 状态，congestion_window_重新生效。

5.2 BBR v2

同样的，BBR v2 最重要的两个接口时 OnCongestionEvent 和 OnPacketSent，下面是代码整体思维导图

BBR v2 将 StartUp、drain、ProbeBw、ProbeRTT 四个状态分开，同时，将网络模型与这四个状态也分离开来。BBR v2 由网络模型四个状态组成。



5.2.1 用到的参数 (Bbr2Params in bbr2_misc.h)

StartUp_full_loss_count:最大丢包事件，如果 StartUp 阶段丢包数超过这个数，就退出 StartUp 阶段。

always_exit_StartUp_on_excess_loss=true:是否在 StartUp 阶段检测丢包

uint32_t probe_bw_probe_max_rounds = 63;在 ProbeBw 阶段，当相邻两次 ProbeBw_UP 阶段的最大 round 间隔，这个间隔是为了与 reno 竞争。下图参考[4]

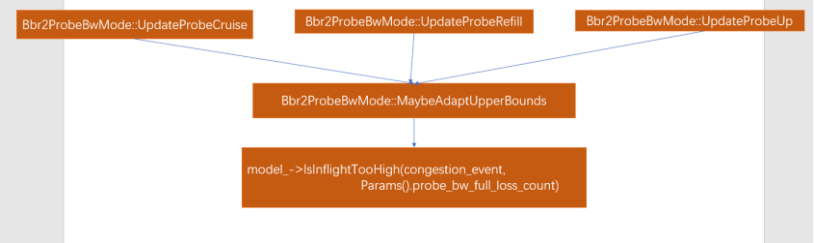
```
 /* How long do we want to wait before probing for bandwidth (and risking
* loss)? We randomize the wait, for better mixing and fairness convergence.
*
* We bound the Reno-coexistence inter-bw-probe time to be 62-63 round trips.
* This is calculated to allow fairness with a 25Mbps, 30ms Reno flow,
* (eg 4K video to a broadband user):
*  BDP = 25Mbps * .030sec /(1514bytes) = 61.9 packets
*
```

QuicTime::Delta probe_bw_probe_base_duration:500ms，probe_bw_probe_max_rand_duration=200ms。ProbeBw down 阶段最少时间和最大的随机时间。probe_bw down 阶段探测时间是如下随机整数:

```
cycle_.probe_wait_time =
    Params().probe_bw_probe_base_duration +
    QuicTime::Delta::FromMicroseconds(sender_->RandomUint64(
        Params().probe_bw_probe_max_rand_duration.ToMicroseconds()));
```

int64_t probe_bw_full_loss_count:在 ProbeBw 阶段，更新在更新 inflight_hi 时，判断 upperbound 的依据

调用栈如下:



float probe_RTT_inflight_target_bdp_fraction = 0.5:ProbeRTT 时，拥塞窗口减半，而不直接设置为 4

QuicTime::Delta probe_RTT_period=2.5s：ProbeRTT 周期，在 v1 版本中为 10s

float inflight_hi_headroom = 0.15 ：在 bbr v2 中会留 headroom,headroom=inflight_hi* inflight_hi_headroom,有两个作用:1.是拥塞窗口的上限;2.ProbeBw down 阶段 inflight 要小于 headroom 才退出

float loss_threshold=0.3:值是我自己设置的，用在 IsInflightTooHigh 中，判断是否过度丢包，见 5.2.2.2

float beta = 0.3;用于设置 inflight_lo 和 bandwidth_lo 和 inflight_hi，

5.2.2 BBR2NetModel

网络模型主要负责的事情是:更新最小 RTT,更新带宽估计,计算传输速率样本,维护增益、inflight_lo、inflight_hi、bandwidth_lo。

其中计算传输速率样本与 v1 中相同。

根据[5]中说法，后缀 lo 是指下界，是一个最近值；后缀 hi 是一个上界，是一个相对 lo 来说长期保持的值。下面分别介绍他们。

5.2.2.1 更新最小 RTT

v1 是用 10s 的时间窗口。v2 将时间窗口减小到 2.5s。另外，在进入 ProbeBw 之前，最小 RTT 是从 StartUp 到 now 的最小 RTT，这段时间并不使用时间窗口。

5.2.2.2 更新带宽估计

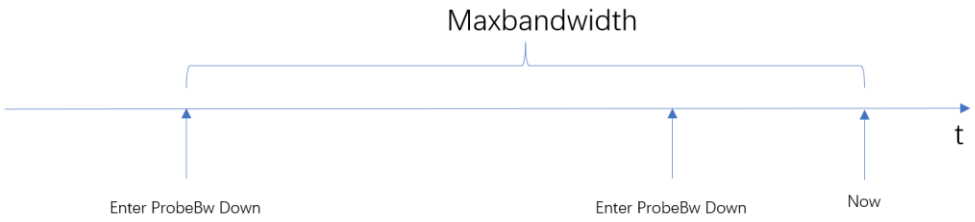
带宽估计是最大滤波器和 bandwidth_lo 二者间的最小值。这一节介绍最大滤波器 Bbr2MaxBandwidthFilter

在 BBR v1，最大滤波器估计带宽使用 Kathleen Nichols 算法，估计的带宽是最近 10 个 RTT 得到的传输速率的最大值。在 BBR v2 中，使用下面的带宽滤波器:

```cpp
class QUIC_EXPORT_PRIVATE Bbr2MaxBandwidthFilter {
public:
  void Update(QuicBandwidth sample) {
    max_bandwidth_[1] = std::max(sample, max_bandwidth_[1]);
  }

  void Advance() {
    if (max_bandwidth_[1].IsZero()) {
      return;
    }

    max_bandwidth_[0] = max_bandwidth_[1];
    max_bandwidth_[1] = QuicBandwidth::Zero();
  }

  QuicBandwidth Get() const {
    return std::max(max_bandwidth_[0], max_bandwidth_[1]);
  }
```

任意时刻调用 Get 时，得到的值时上上次 Advance()时间点到 Now 的传输速率的最大值。在 BBR v2 中，最大带宽的逻辑如下：

1. 在进入 ProbeBw Down 之前，最大带宽是 StartUp 到现在的最大值。
2. 第一次进入 ProbeBw Down 之后，最大带宽是“进入 ProbeBw Down 之前的最大带宽“和“从进入 ProbeBw Down 到现在的最大带宽”二者的最大值；第二次进入 ProbeBw Down 之后，最大带宽是上上次 ProbeBw Down 现在的最大值，示意图如下：(并不明白为什么要这样做，为什么要将最大带宽窗口从 10 个 RTT 扩展为两个 ProbeBw cycle？)



5.2.2.3 bandwidth_lo、inflight_lo

传输速率、inflight 的上限分别是 bandwidth_lo，inflight_lo。

bandwidth_lo 是最新带宽样本和 0.7 倍 bandwidth_lo 二者间的最大值。

```cpp
if (bandwidth_lo_.IsInfinite()) {
  bandwidth_lo_ = MaxBandwidth();
}
bandwidth_lo_ =
    std::max(bandwidth_latest_, bandwidth_lo_ * (1.0 - Params().beta));
```

inflight_lo 估计类似：

```cpp
if (inflight_lo_ == inflight_lo_default()) {
  inflight_lo_ = congestion_event.prior_cwnd;
}
inflight_lo_ = std::max<QuicByteCount>(
    inflight_latest_, inflight_lo_ * (1.0 - Params().beta))
```

为什么要有 inflight_lo 和 bandwidth_lo?

[5]中给的解释是，如果带宽骤降，那么这个机制能够保证窗口和发送速率缓慢减小，而不是骤降。

5.2.2.4 inflight_hi

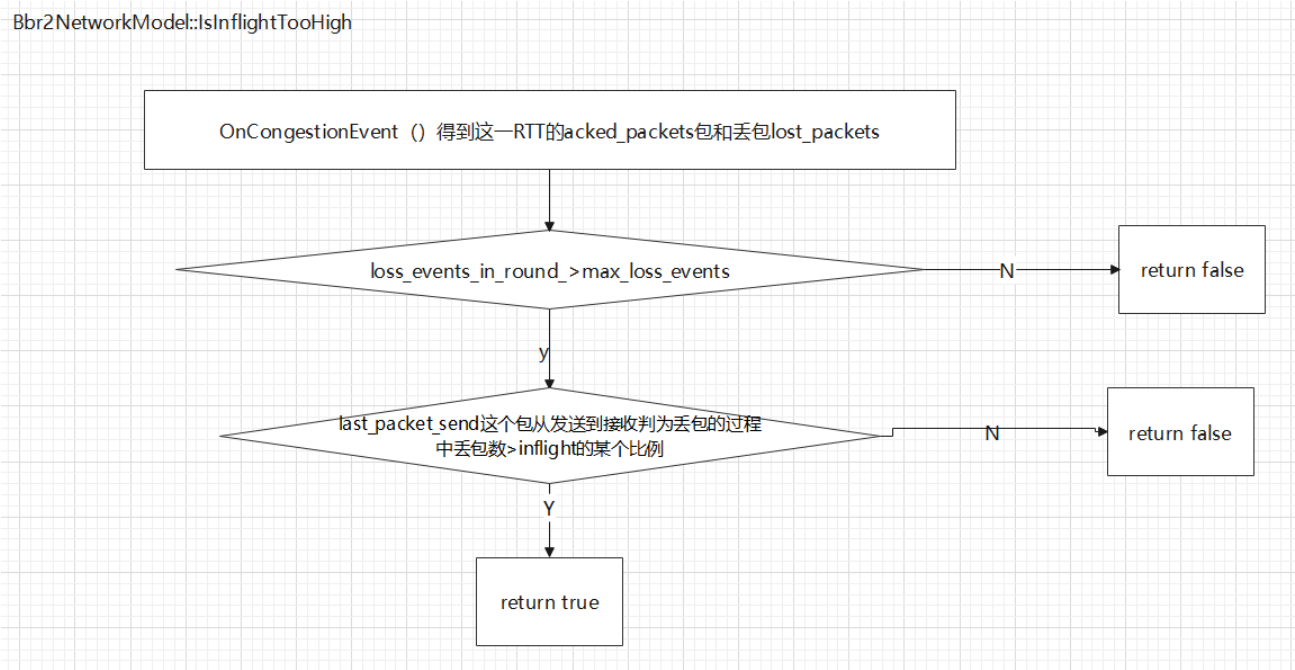
任意时刻如果发现 IsInflightTooHight（见 5.2.2.5）为真，那么就设置 inflight_hi，作为发送窗口的另外一个上限。

在 ProbeBw up 阶段，由于在探测新的可用带宽(pacing gain=1.25)，inflight_lo 会逐渐增大，此时 inflight_hi 也

需要增大，bbr v2 用的方法是指数增加 inflight_hi(见 5.2.5.3)。

5.2.2.5 IsInflightTooHigh 逻辑

IsInflightTooHigh 用来判断某一时刻 infligh 的数量是否太多了。



5.2.3 bbr2_StartUp

StartUp 阶段的判断以 RTT 为单位。与 bbr v1 相比，增加了 StartUp 阶段检测丢包的过程:

5.2.4 bbr2_drain

与 bbr v1 相比没有变化

5.2.5 bbr2_ProbeBw

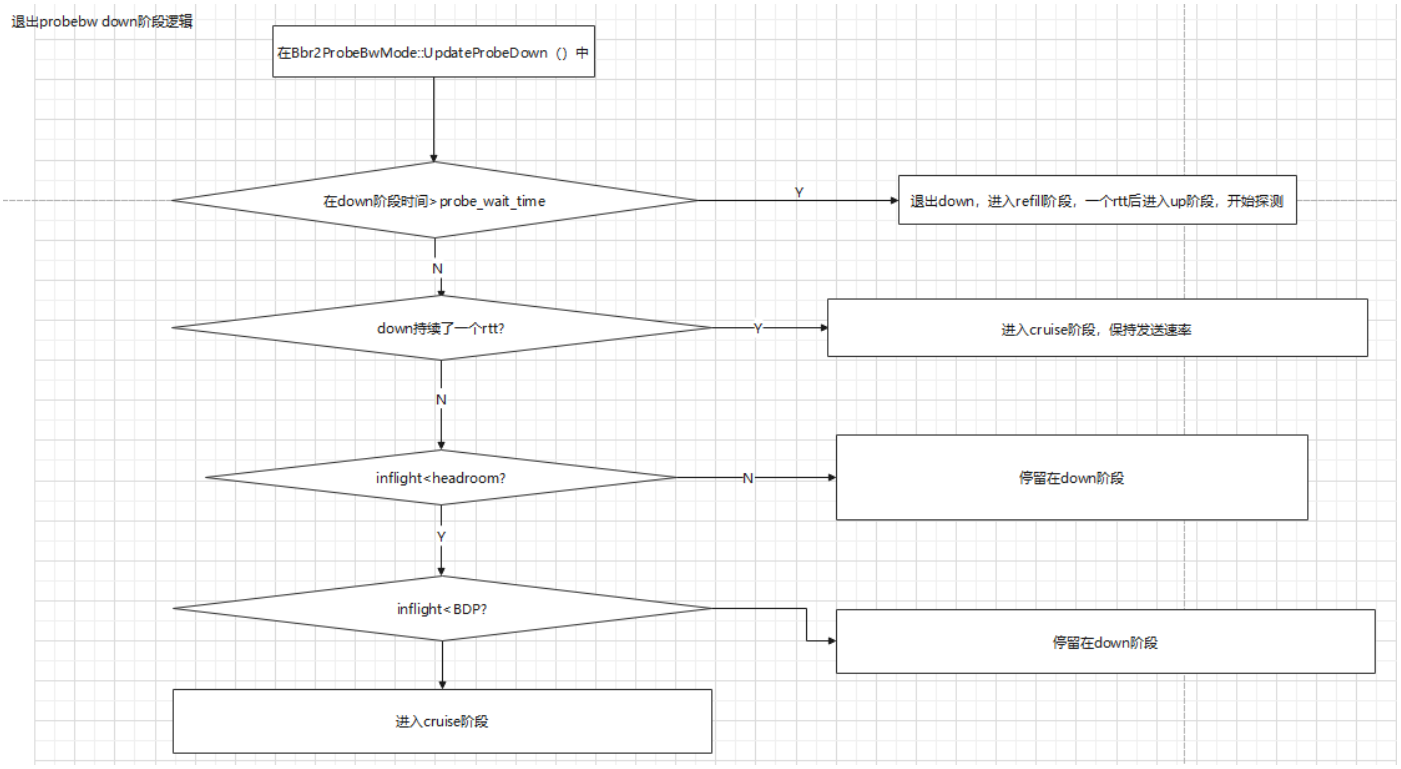
v2 的 ProbeBw 中，不再有 pacing_gain 数组，而是将不同的 pacing gain 分成 up、down、cruise 这三个阶段。在每个阶段中，如果有丢包比较多，会重新设置 inflight_hi（根据[5]中介绍，inflight_hi 为可以达到的最大 inflight，称为 long term inflight，但是留有一定的 headroom，从而允许其他流加入网络)。

5.2.5.1 down 阶段

首次进入 ProbeBw 阶段时，首先进入 down 阶段，down 阶段就是 v1 中增益等于 0.75 的情况，v2 的改变为，将 down 阶段时长随机化，设计者的解释为:for better mixing and fairness convergence[4]。

```
cycle_.probe_wait_time =
    Params().probe_bw_probe_base_duration +
    QuicTime::Delta::FromMicroseconds(sender_->RandomUint64(
        Params().probe_bw_probe_max_rand_duration.ToMicroseconds()));
```

退出 down 阶段的逻辑如下

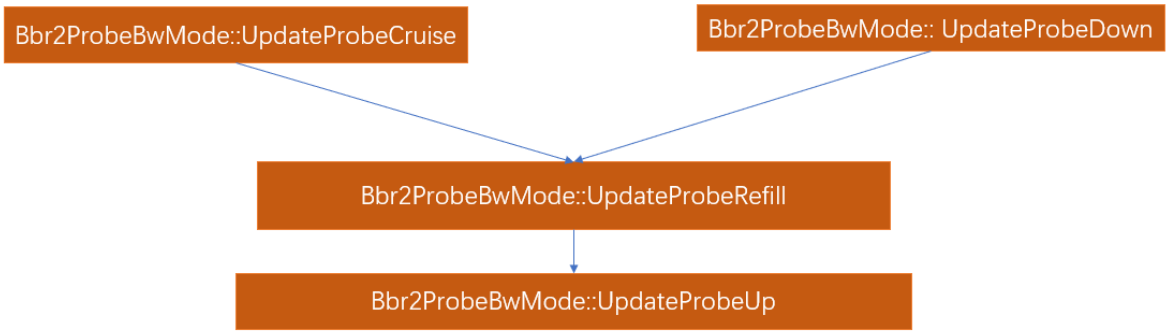


5.2.5.2 cruise 阶段

类似 bbr v1 中 pacing_gaing=1 的情况，不过在 bbr2 中，退出 cruise 阶段的条件不再是 6 个 RTT，而是持续了 wait_time 或者探测了 63 个 round，不过代码中 wait_time 并未设值，而是使用 down 阶段的 wait_time，不知道这是不是一个 bug。(不是 bug，wait_time 针对整个 ProbeBw 的所有阶段，wait_time 在次进入 down 的时候随机，这与 v1 中，每个 phase 持续一个 RTT 不同，v2 中 down 和 cruise 总时长最多持续 wait_time 时间)

5.2.5.3 up 阶段

1.up 阶段类似 bbr v1 中 pacing_gain=1.25 的情况。不过在 bbrv2 中，进入 up 阶段前首先要进入 refill 这个中间阶段。refill 阶段类似 cruise，pacing_gain=1，不过 refill 阶段只持续一个 RTT，随后立即进入 Up 阶段。(为什么要 refill 阶段?作为一个过渡阶段？)



2.up 阶段与其他阶段的主要不同是，探测新的可用 inflight_hi:

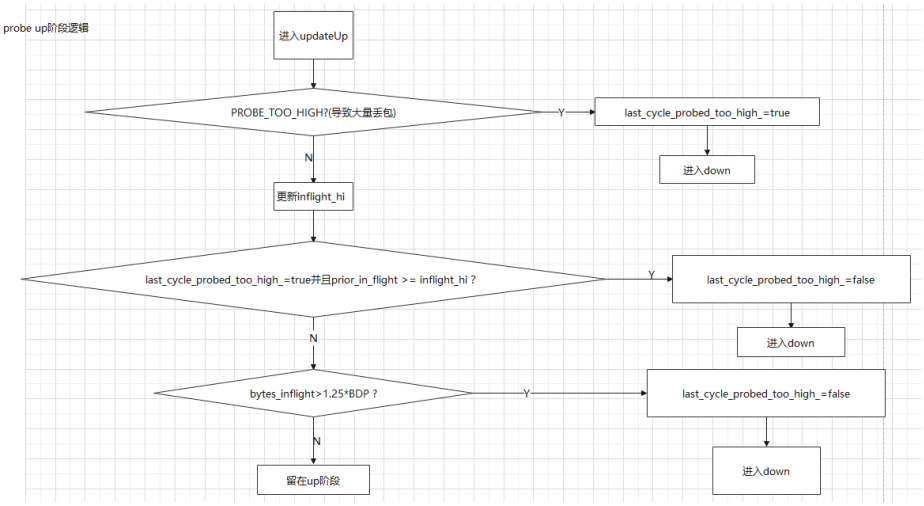
基本思想是，每一个 round 按照指数增加窗口:1,2,4,8,16。算法如下[6]:(还需进一步理解)

Algorithm 2 ProbeInflightHighUpward

Input:
bytes_acked, is_round_end

```
probe_up_acked+ = bytes_acked
if probe_up_acked ≥ probe_up_bytes then
    delta = ⌊probe_up_acked / probe_up_bytes⌋
    probe_up_acked− = delta ∗ probe_up_bytes
    inflight_hi ← inflight_hi + delta ∗ MSS
end if
if is_round_end then
    growth ← 1 << probe_up_rounds
    probe_up_rounds ← min(30, probe_up_rounds + 1)
    probe_up_bytes ← ⌊cwnd / growth⌋
    probe_up_bytes ← max(MSS, probe_up_bytes)
end if
```

3.up 阶段退出条件:



为什么要 last_cycle_probe_too_high?

5.2.6 丢包对 BBR v2 的影响

在 StartUp 阶段中，丢包过多会提前退出 StartUp。

在 ProbeBw 阶段中，丢包过多会提前退出 Up 阶段，然后进入 Down 阶段。

在这两种情况下，都会重新设置 inflight_hi。

5.3 其他内容

1. QuicConnectionStats connection_stats_作用:用于在不用发送算法中间进行切换。比如 BBR v1 算法切换为 BBR v2 算法，需要知道当前网络状态，从而无缝进行切换发送算法。测试代码中未设计到这方面内容。

2.目前 BBR v2 中 chrome 代码问题：丢包情况没有像 v1 中那样处理。

参考文献

1. BBR Congestion Control draft-cardwell-iccrg-bbr-congestion-control-00
2. Delivery Rate Estimation draft-cheng-iccrg-delivery-rate-estimation-00
3.https://source.chromium.org/chromium/chromium/src/+/main:net/third_party/quiche/src/quic/core/congestion_control/
4. https://groups.google.com/g/bbr-dev/c/8qh82UnBakc
5. https://www.youtube.com/watch?v=cJ-0Ti8ZlfE&t=210s
6. Songyang Zhang. An Evaluation of BBR and its variants
7. https://github.com/ldy921227/SGI-STL-GCC2.91/blob/master/sgi-stl-master/g%2B%2B/stl_deque.h
8. Y. -J. Song, G. -H. Kim, I. Mahmud, W. -K. Seo and Y. -Z. Cho, "Understanding of BBRv2: Evaluation and Comparison With BBRv1 Congestion Control Algorithm," in IEEE Access, vol. 9, pp. 37131-37145, 2021, doi: 10.1109/ACCESS.2021.3061696.
9. https://blog.csdn.net/dog250/article/details/52879298
10. Ha, Sangtae & Rhee, Injong & Xu, Lisong. (2008). CUBIC: a new TCP-friendly high-speed TCP variant. Operating Systems Review. 42. 64-74. 10.1145/1400097.1400105.