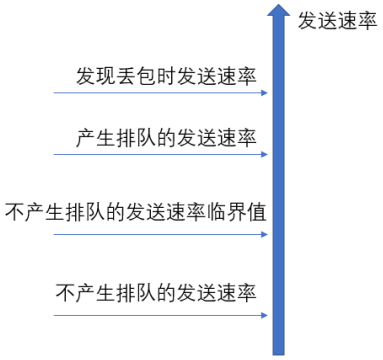
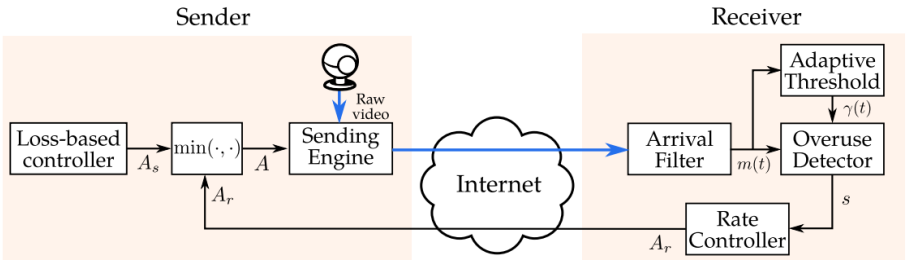
目录

一、介绍

经典的 TCP 拥塞控制算法使用丢包信息作为网络状态改变的条件，当发现传输有丢包，则调整发送速率(通过调整拥塞窗口实现)。在一般情况下，如果发生丢包，意味着中间路由缓存没有可用空间，中间路由产生了大量排队。理想情况下，我们希望数据传输过程没有路由排队[1]，而基于丢包的拥塞控制无法告知我们什么时候产生了网络排队。如下图所示，发生丢包时，发送速率已经超过了不产生排队时的发送速率。如果想要在传输过程中减少丢包或者排队的数据量，那么需要能够检测到网络瓶颈路由器的排队情况，然后对发送速率设置限制，而不是一直增加发送速率，直到丢包时才调整发送速率。



GCC(Google Congestion Control)是谷歌在 webrtc 上使用的拥塞控制算法。GCC 算法结构如下图[1]所示:



GCC 在发送端保留了基于丢包的带宽估计，在接收端使用基于延时的带宽估计(GCC 基于丢包的带宽估计和基于延时的带宽估计可以同时位于发送端[2]，本文不做介绍)。基于延时的带宽估计通过 RTT 变化估计瓶颈路由器缓存排队情况，得到一个发送速率值 Ar。接收端将 Ar 返回给发送端，发送端收到 Ar 之后发送速率不能超过 Ar。一般情况下，根据瓶颈路由排队情况得到的 Ar 小于丢包时的发送速率，GCC 通过给基于延时的带宽估计给 As 设置上限，来达到减少丢包和减少瓶颈路由器缓存排队的目的。当瓶颈路由排队正在大量增加，基于延时的带宽估计希望发送端减小发送速率，从而达到尽量避免丢包的目的；当瓶颈路由器排队正在大量减少，基于延时的带宽估计希望保持当前发送速率，使网络排队大量减少，从而达到尽量减少网络排队的目的；当瓶颈路由排队变化情况不大，那首先保持 Ar，然后增加 Ar，使发送端能够增加发送速率，探测新的可用带宽。

二、基于丢包的带宽估计

GCC 保留了基于丢包的带宽估计，根据丢包情况对带宽做出以下调整：

$$A_s(t_k) = \begin{cases} A_s(t_{k-1})(1 - 0.5f_l(t_k)) & f_l(t_k) > 0.1 \\ 1.05(A_s(t_{k-1})) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases}$$

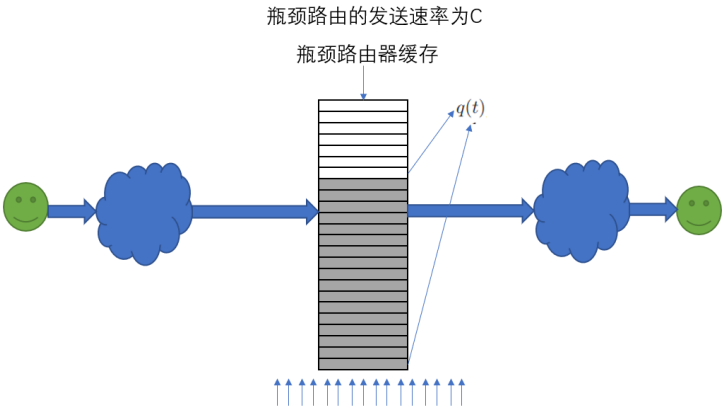
其中，$f_l(t_k)$当前时刻的丢包率，$As(t_{k-1})$是上一时刻发送速率，$As(t_k)$是调整后的发送速率。

丢包率小于 2%时，依然增加发送速率，意味着允许少量丢包，这个机制能够更好的与其他流竞争。

三、基于延时的带宽估计

3.1 网络建模

网络模型如下（Note:实际中，可能在多个中间路由器上产生排队，所有排队延时可以归并到单个路由器上面），



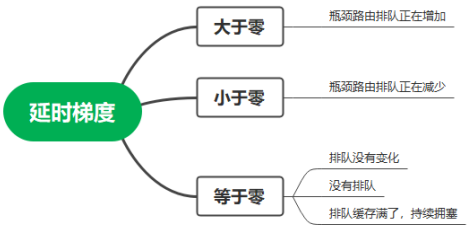
通信链路上有一个瓶颈路由器，经过瓶颈路由器的数据的排队延时满足[1]:

$$T_q(t) = q(t)/C$$

其梯度为:

$$\dot{T}_q(t) = \frac{\dot{q}(t)}{C}$$

排队延时梯度与排队情况满足如下关系:

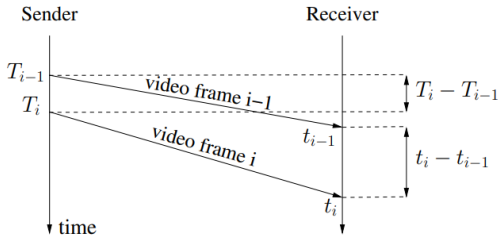


3.2 单向延时差(delay variation)

一般无法测得路由器上排队延时，在接收端，可以通过测量相邻包的单向延时差得到排队延时 m(t)。第 i 个数据包的单向延时等于:

$$D(i) = P(i) + T_q(i) + n(i)$$

其中 D(i)是单向延时，P(i)是无排队时单向延时，Tq(i)时路由器排队时长，n(i)是测量噪声。在接收端，$D(i) = t_i - T_i$，其中 t_i是收包时间，T_i是包发送时间。



GCC 通过测量相邻两个包的单向延时差 d(i)来估计$T_q(i)$ ，

$$d(i) = (t_i - T_i) - (t_{i-1} - T_{i-1}) = (t_i - t_{i-1}) - (T_i - T_{i-1})$$

测量的 d(i)含有测量噪声，需要对 d(i)滤波，论文[1][2][4]使用 kalman 滤波器对 d(i)进行动态滤波得到滤波后数据 m(i)，根据 m(i)来估计瓶颈路由缓存排队情况。Chrome 代码[3]中使 20 个包组(packet group)的排队延时差样本，用线性回归拟合出 20 个包组排队延时变化直线，利用直线斜率 m(i)估计单向延时的趋势。Kalman 滤波本文不做介绍，使用线性回归的预测见第五章。

3.3 自适应阈值的设计

GCC 将 m(i)与阈值γ(i)进行比较，然后产生响应的信号，状态转换机制收到信号后进行状态转换(见 3.4)。

m(i)>γ(i)：认为瓶颈路由排队在增加，发出 overuse 信号

m(i)<-γ(i)：认为瓶颈路由排队在减少，发出 underuse 信号

γ(i)>m(i)>-γ(i)：认为瓶颈路由排队基本稳定，发出 normal 信号

阈值γ(i)设计如下

$$\gamma(i) = \gamma(i-1) + \triangle t_i \cdot K(i) \cdot (|m(i)| - \gamma(i-1))$$

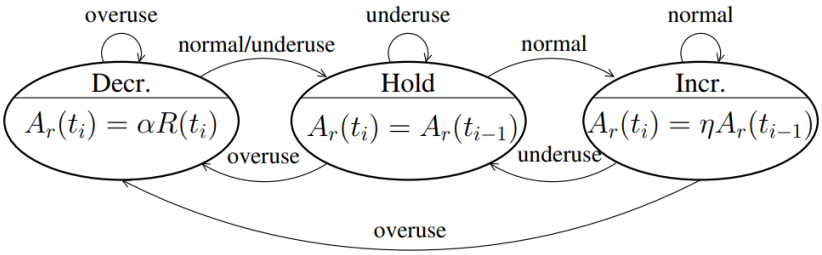
其中$\triangle t_i$ =t_i-t_{i-1}，K(i)满足：

$$K(i) = \begin{cases} K_d = 0.00018 & |m(i)| < \gamma(i-1) \\ K_u = 0.01 & Otherwise \end{cases}$$

3.4 状态转换机制

基于延时的带宽估计器有三个状态，Increase(Incr)、Decrease(Decr)、Hold。这三个状态分别对 Ar 做增加、减少、保持处理，处理后的 Ar 将作为发送端发送速率的上限。

当 m(i)与阈值做比较，发出信号之后，状态转换机制通过信号进入不同的状态。状态转换图如下所示：



解释：

1. 任意时刻收到 overuse 信号，说明排队正在增加，应该减少发送速率，减小排队；
2. 任意时刻收到 underuse 信号，说明排队正在减少，为了降低传输延时，在尽量低排队延时的情况下传输数据，保持 Ar，让排队继续减少；
3. 任意时刻收到 normal 信号，如果之前位于 Decrease 状态，说明现在排队增长相对稳定，保持 Ar；如果之前位于 Increase 状态，说明增加发送速率是安全的，继续增加 Ar；如果之前位于 Hold 状态，说明增加发送速率保持了一段时间后仍然正常，进入 Increase 状态，尝试增加发送速率，探测新的可用带宽。

3.4.1 Decrease

Ar 等于当前传输速率的 0.85 倍。

3.4.2 Increase

Increase 有两种方式，一种是加性增，一种是乘性增：

加性增:Ar 在一个 rtt 内增加一个包/rtt。

乘性增:Ar = 1.08*传输速率;

乘性增有两种情况:1.如果上一次状态是 Decrease，并且到传输速率小于平均传输速率-3*平均传输速率偏差;2.如果传输速率大于平均传输速率+3*平均传输速率偏差。

除了上面两种情况，其他情况加性增。

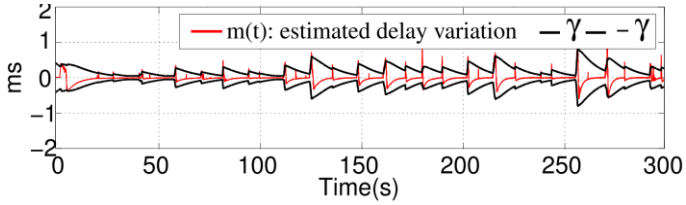
注意，得到的最终 Ar 的上限是 1.5 的传输速率。

3.5 理解自适应阈值的设计

将γ(i)做如下变换，可以看出，γ(i)是|m(i)|的指数加权平均，系数为$\triangle t_i \cdot K(i)$。

$$\gamma(i) = (1 - \triangle t_i \cdot K(i)) \cdot \gamma(i-1) + \triangle t_i \cdot K(i) \cdot (|m(i)|)$$

K(i)的选取意味着，当|m(i)|>=γ(i)时，γ(i)赋予更多的权重给当前样本 m(i)，从而γ(i)能够迅速赶上 m(i)；当|m(i)|<γ(i)，γ(i)赋予更多样本给过去的值，γ(i)都 m(i)的变化反映较慢。下图是γ随着 m(i)的变化图[1]，反映了这个现象。

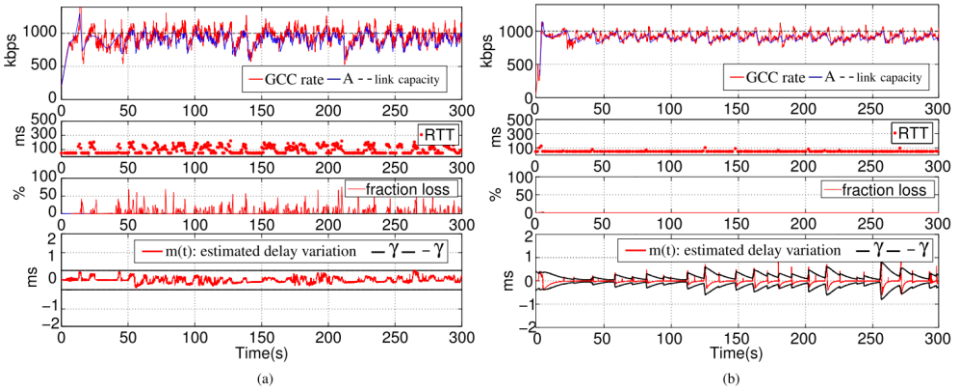


3.5.1 为什么不适用固定阈值

固定阈值带来两个问题[1]：1.当瓶颈路由缓存变化一直小于固定阈值γ时，基于延时的带宽估计不起作用；2.固定阈值会导致 GCC 无法与基于 Reno 的流量竞争。

3.5.1.1 瓶颈路由缓存变化一直小于固定阈值γ

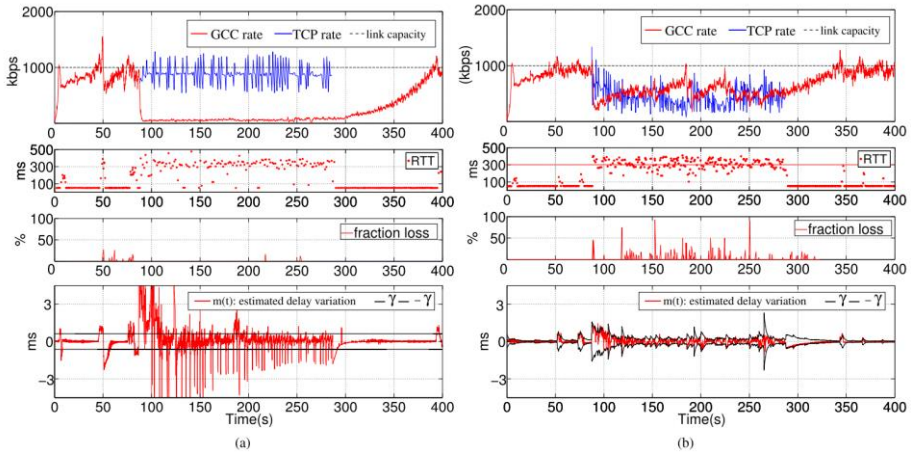
下图来自[1]，图 a 是固定阈值时发送速率，RTT，丢包率变化情况，图 b 是自适应阈值时情况。可以看出，固定阈值的发送速率、RTT 抖动比较大，而且丢包比较多。



出现这样现象的原因是：由于|m(i)|一直小于γ(i)，导致基于延时的带宽估计一直发出 normal 信号，从而 Ar 一直增加，最终 Ar 会大于发现丢包时的发送速率，从而 Ar 失效，基于延时的带宽估计失效。

3.5.1.2 无法与基于 Reno 的流量竞争

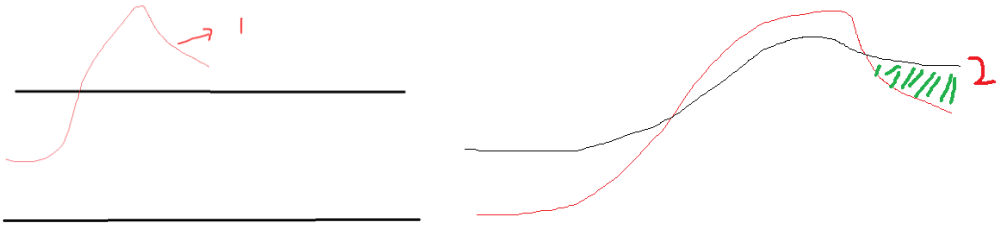
下图来自[1] ，图 a 是固定阈值时发送速率，RTT，丢包率变化情况，图 b 是自适应阈值时情况。图 a 与图 b 相比，使用固定阈值时，当有基于 Reno 的数据流到达，GCC 的带宽被抢占。



出现这样现象的原因是：对于自适应阈值，当 m(i)比较大，ku 比较大意味着γ(i)可以快速跟进，从而可以快速发送 normal 信号，当 m(i)从减小时，kd 比较小意味着γ(i)跟进的比较慢，在更多情况下γ(i)>m(i)>-γ(i)，从而发送 normal 信号，normal 信号导致 Ar 增加，最后允许发送端增加发送速率，竞争带宽；对于固定阈值，Reno 流到达之后，导致|m(i)|>γ(i)，发出 overuse 信号，由于阈值不变，发送 overuse 信号次数相对自适应阈值较高，从而 Ar 减小比较

多，从而抑制基于发送端发送速率，从而与基于 Reno 的流的竞争中落于下风。

下图是固定阈值与自适应阈值变化，对发出 overuse 信号的影响，左图是固定阈值，右图是自适应阈值，黑色线是阈值，红色线是 m(i):



固定阈值在 1 仍然发送 overuse 信号，从而 Ar 仍然降低，自适应阈值在 2 的绿色阴影部分保持进入 normal 状态。可以看出，自适应阈值发出 normal 信号更多。使用自适应阈值的极限情况是，一直发出 normal 信号，从而基于延时的带宽估计不停地增加 Ar，这最终使得基于延时的带宽估计失效，回退为基于丢包的带宽估计，从而能够与基于 Reno 的流竞争。

总体来说，为了与基于 Reno 的流更好的竞争，自适应阈值保证了基于带宽的估计大部分时间都发射 normal 信号，从而可以增加 Ar，使发送端逐渐不不收基于延时的带宽估计的影响，而逐渐变成基于丢包的带宽估计(Reno 也是基于丢包的带宽估计)，从而可以与 Reno 竞争。

3.5.2 自适应阈值系数为什么要与$\triangle t_i$有关

当相邻两个包组的到达间隔比较大，即$\triangle t_i$比较大，那么意味着当前传输速率比较小，可能是由于带宽竞争中处于下风。$\triangle t_i$的存在使得系数指数加权平均系数比较大，从而提升带宽竞争能力(如 3.5.1.2 中所述)。

3.6 遗留问题

Q1.自适应阈值中 ku 和 kd 是怎样选出来的?

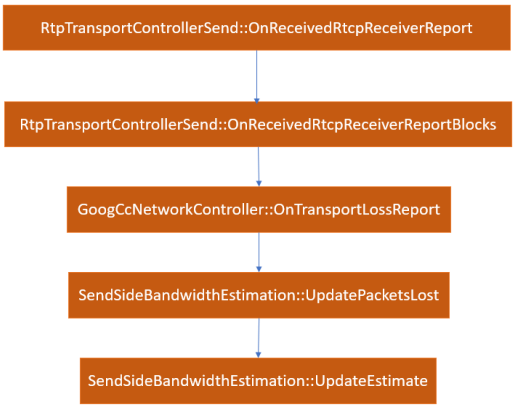
A1. 参考[1]，目前未完全理解

Q2. GCC 基于延时的带宽估计是否只有当其他流涌入时才有效？当只有单个流时，GCC 会失效?

A2. 当只有 GCC 时，m(i)受发送端发送速率变化的影响，这种情况下，发送端发送速率变化比较快的话会影响 m(i)。那么理论上会出现锯齿状的发送速率变化。

四、基于丢包的带宽估计代码

调用栈如下：发送端每间隔 1s，发送一个 report，接收端每 1s 更新一次基于丢包的带宽估计



基于丢包的带宽估计代码[3]位于: `SendSideBandwidthEstimation::UpdateEstimate(Timestamp at_time)`。chrome 使用 RTP/RTCP 协议，如果要得到丢包率，那么需要将 RTCP 反馈的 fraction_loss 除以 256.

4.1 小于 2%

1. 如果丢包率小于 2%
2. 那么基于丢包的发送速率立即增加 8%
3. 增加 1k，避免发送速率太小

```
if (time_since_loss_packet_report < 1.2 * kMaxRtcpFeedbackInterval) {
  // We only care about loss above a given bitrate threshold.
  float loss = last_fraction_loss_ / 256.0f;
  // We only make decisions based on loss when the bitrate is above a
  // threshold. This is a crude way of handling loss which is uncorrelated
  // to congestion.
  if (current_target_ < bitrate_threshold_ || loss <= low_loss_threshold_) {
    // Loss < 2%: Increase rate by 8% of the min bitrate in the last
    // kBweIncreaseInterval.
    // Note that by remembering the bitrate over the last second one can
    // rampup up one second faster than if only allowed to start ramping
    // at 8% per second rate now. E.g.:
    //   If sending a constant 100kbps it can rampup immediately to 108kbps
    //   whenever a receiver report is received with lower packet loss.
    //   If instead one would do: current_bitrate_ *= 1.08^(delta time),
    //   it would take over one second since the lower packet loss to achieve
    //   108kbps.
    DataRate new_bitrate = DataRate::BitsPerSec(
        min_bitrate_history_.front().second.bps() * 1.08 + 0.5);

    // Add 1 kbps extra, just to make sure that we do not get stuck
    // (gives a little extra increase at low rates, negligible at higher
    // rates).
    new_bitrate += DataRate::BitsPerSec(1000);
    UpdateTargetBitrate(new_bitrate, at_time);
    return;
```

4.2 丢包率位于 2%到 10%

基于丢包的带宽估计不变

4.3 丢包率大于 10%

如基于丢包的带宽估计所述。

$$\left(\frac{512 - lost\_fraction\_loss}{512}\right) \cdot current\_bitrate$$

$$= \left(1 - \frac{1}{2}\frac{lost\_fraction\_loss}{256}\right) \cdot current\_bitrate$$

$$= \left(1 - \frac{1}{2} loss\_ratio\right) \cdot current\_bitrate$$

```
    return;
  } else if (current_target_ > bitrate_threshold_) {
    if (loss <= high_loss_threshold_) {
      // Loss between 2% - 10%: Do nothing.
    } else {
      // Loss > 10%: Limit the rate decreases to once a kBweDecreaseInterval
      // + rtt.
      if (!has_decreased_since_last_fraction_loss_ &&
          (at_time - time_last_decrease_) >=
              (kBweDecreaseInterval + last_round_trip_time_)) {
        time_last_decrease_ = at_time;

        // Reduce rate:
        //   newRate = rate * (1 - 0.5*lossRate);
        //   where packetLoss = 256*lossRate;
        DataRate new_bitrate = DataRate::BitsPerSec(
            (current_target_.bps() *
             static_cast<double>(512 - last_fraction_loss_)) /
            512.0);
        has_decreased_since_last_fraction_loss_ = true;
        UpdateTargetBitrate(new_bitrate, at_time);
        return;
      }
    }
  }
}
```

4.4 相关的参数

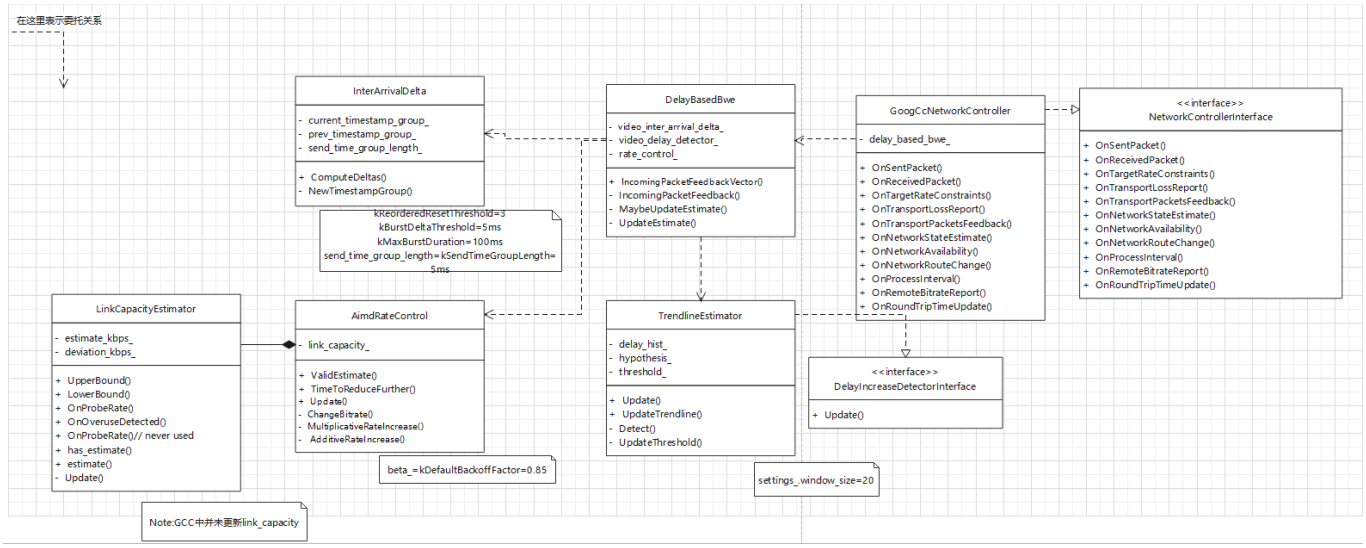
kMaxRtcpFeedbackInterval=5s：收到 RTCP 包的最大区间，理想情况下为 [0.5, 1.5]s，但是可能由于网络抖动而变化；丢包估计最多只统计 kMaxRtcpFeedbackInterval *1.2=6s 内的丢包情况。

kLimitNumPackets =20：更新基于丢包的带宽估计的最小包的数量

kBweDecreaseInterval=300ms：相邻两次降低发送速率的间隔为 300ms+rtt，避免降低 Ar 太频繁。

五、基于延时的带宽估计代码

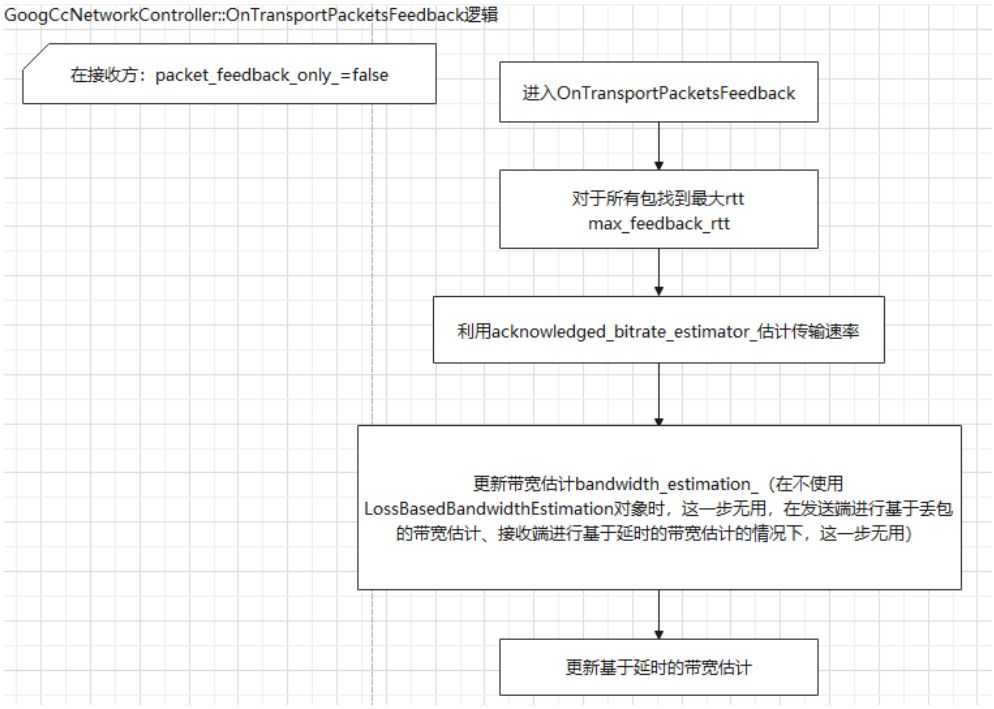
chrome 代码整体框架如下图所示，这里只显示 DelayBaseBwe。当 DelayBaseBwe 收到 GccCcNetworkController 传递的多个包信息 msg vector 之后，对于每个包，使用 InterArrivalDelta 将包分组并计算发送时间差、接收时间差，得到计算后的值之后，更新预测 trend，也就是应该产生 overuse、normal、underuse 中的哪个信号。



Note：chrome 代码中基于延时的带宽估计和基于丢包的带宽估计都在发送端，而我们测试代码将基于延时的丢包估计放在接收端，下面代码分析部分都按照基于延时的带宽估计在接收端的方式分析。

5.1 GoogCcNetworkController::OnTransportPacketsFeedback

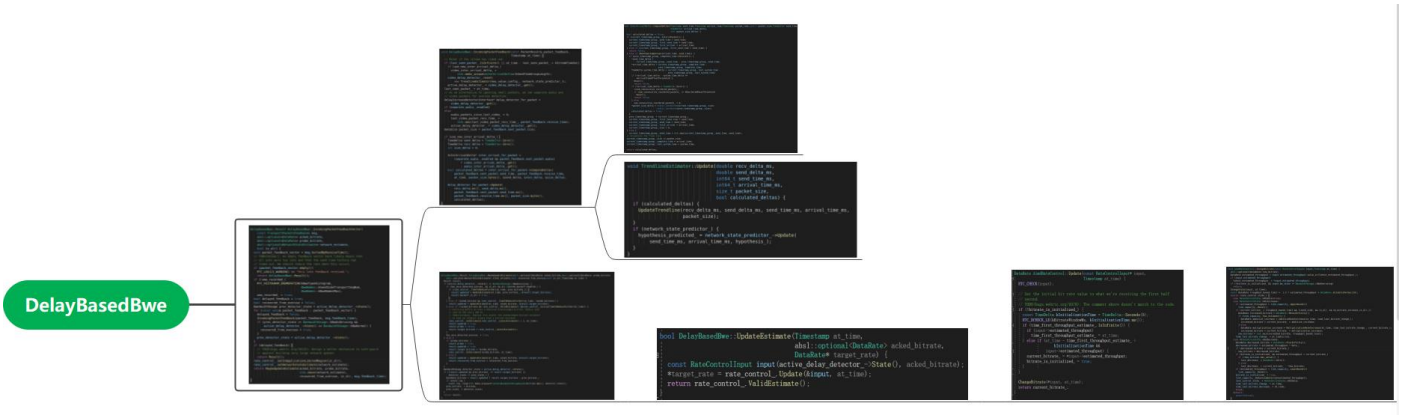
GoogCcNetworkController::OnTransportPacketsFeedback()函数处理逻辑如下：



关键步骤为：计算传输 rtt（在测试代码中其实是单向延时，用于 aimd_controler）、计算传输速率、更新基于延时的带宽估计

5.2 DelayBaseBwe:: IncomingPacketFeedbackVector

DelayBaseBwe 调用关系如下图所示



对于所有包消息，进入 IncomingPacketFeedback 更新发送信号，然后根据最终的信号，调用 MaybeUpdateEstimate 更新来 Ar

5.2.1 IncomingPacketFeedback

inter_arrival_for_packet：分组计算 delta

delay_detector_for_packet：根据 delta 更新信号

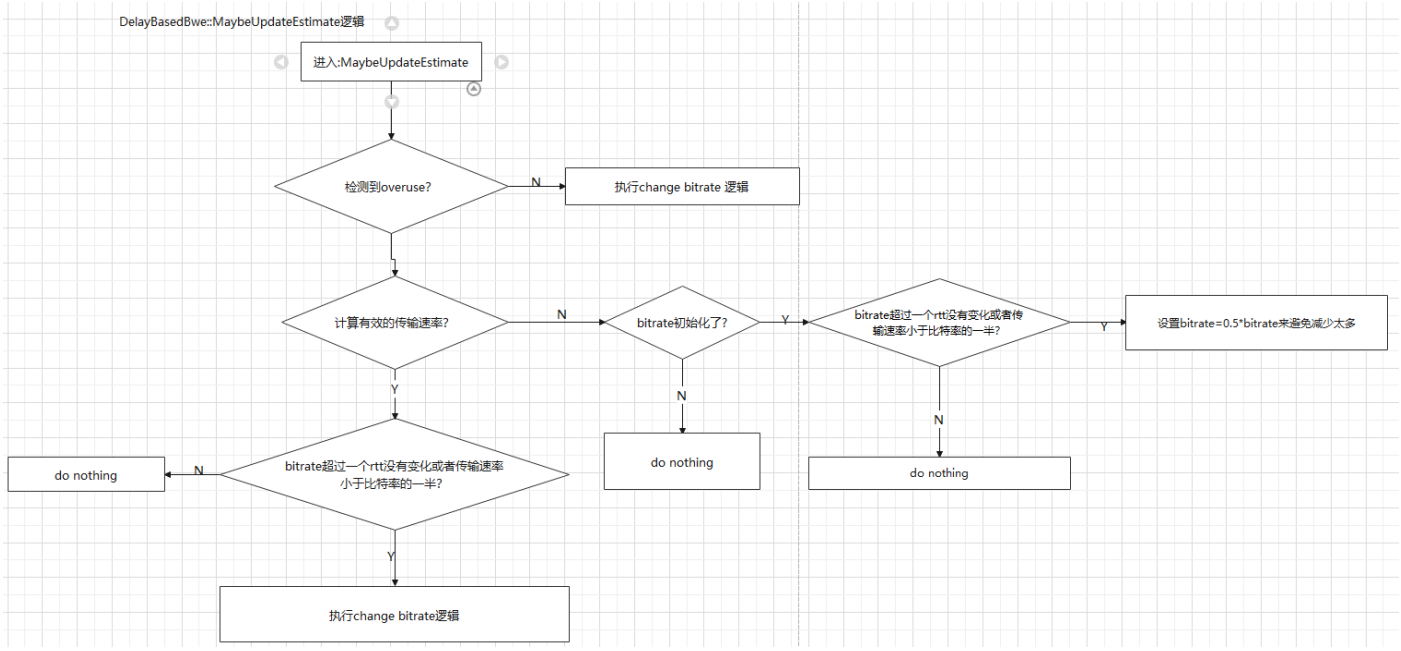
注意 1.如果接收端很长时间没有收到包，那么重置 inter_arrivval_delta 和 delay_detector，如代码图 1 所示。

2. 代码中，当(separate_audio_.enabled=true 且 packet_feedback.sent_packet.audio=true)为假时，delay_detector_for_packet 使用的 video,而 inter_arrival_for_packet 使用的 audio,不知道是不是命名上的 bug,实际上，这两个对象是相同的，使用谁都可以。

```
void DelayBasedBwe::IncomingPacketFeedback(const PacketResult& packet_feedback,
                                           Timestamp at_time) {
  // Reset if the stream has timed out.
  if (last_seen_packet_.IsInfinite() || at_time - last_seen_packet_ > kStreamTimeOut)
    if (use_new_inter_arrival_delta_)
      video_inter_arrival_delta_ = std::make_unique<InterArrivalDelta>(kSendTimeGroupLength);
      audio_inter_arrival_delta_ = std::make_unique<InterArrivalDelta>(kSendTimeGroupLength);
    video_delay_detector_.reset( new TrendlineEstimator(key_value_config_, network_state_predictor_));
    active_delay_detector_ = video_delay_detector_.get();
  last_seen_packet_ = at_time;
  // As an alternative to ignoring small packets, we can separate audio and
  // video packets for overuse detection.
  DelayIncreaseDetectorInterface* delay_detector_for_packet = video_delay_detector_.get();
  if (separate_audio_.enabled)
    if (packet_feedback.sent_packet.audio)
      delay_detector_for_packet = audio_delay_detector_.get();
    else
      audio_packets_since_last_video_ = 0;
      last_video_packet_recv_time_ =
          std::max(last_video_packet_recv_time_, packet_feedback.receive_time);
      active_delay_detector_ = video_delay_detector_.get();

  if (use_new_inter_arrival_delta_)
    TimeDelta send_delta = TimeDelta::Zero();
    TimeDelta recv_delta = TimeDelta::Zero();
    int size_delta = 0;

    InterArrivalDelta* inter_arrival_for_packet =
        (separate_audio_.enabled && packet_feedback.sent_packet.audio)
            ? video_inter_arrival_delta_.get()
            : audio_inter_arrival_delta_.get();
    bool calculated_deltas = inter_arrival_for_packet->ComputeDeltas(
        packet_feedback.sent_packet.send_time, packet_feedback.receive_time,
        at_time, packet_size.bytes(), &send_delta, &recv_delta, &size_delta);

    delay_detector_for_packet->Update(
        recv_delta.ms(), send_delta.ms(),
        packet_feedback.sent_packet.send_time.ms(),
        packet_feedback.receive_time.ms(), packet_size.bytes(),
        calculated_deltas);
}
```

5.2.2 MaybeUpdateEstimate

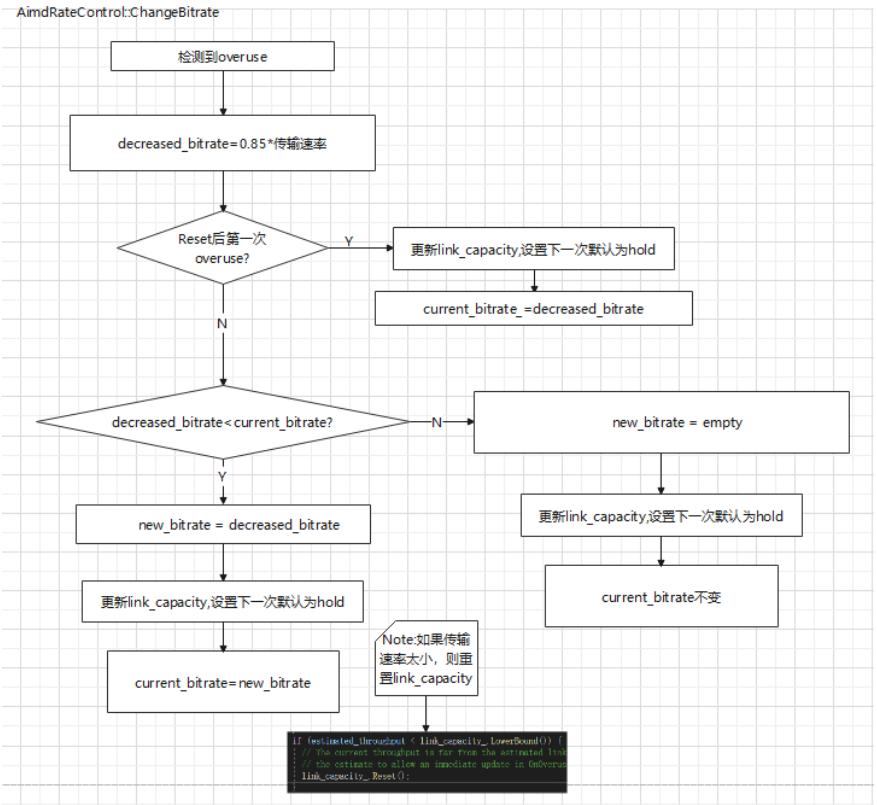
逻辑如下：当检测到 overuse，bitrate 超过一个 rtt 没有变化或者传输速率小于比特率的一半时，才调用 UpdataEstimate 减少。



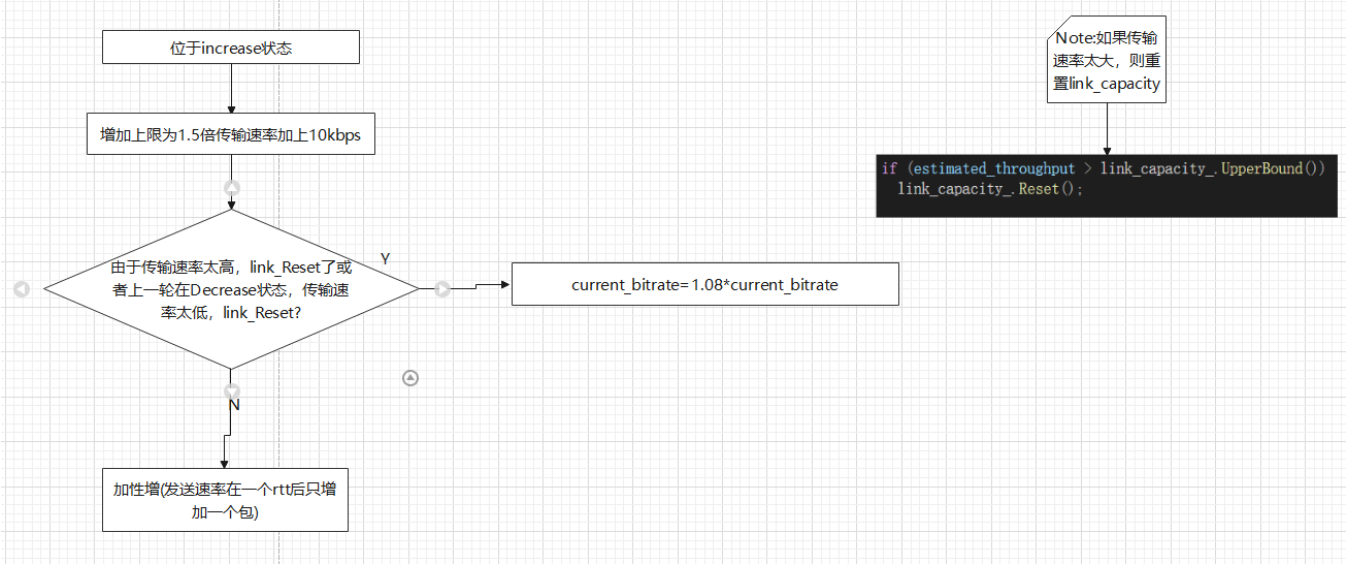
5.2.3 UpdateEstimate

UpdateEstimate 根据最新信号来更新 Ar，调用 AimdRateControl 的 ChangeBitRate，更新 Ar.

5.2.3.1 检测到 overuse 的更新流程



5.2.3.2 在 Decrease 状态的更新流程



5.3 相关的参数

TimeDelta kSendTimeGroupLength = TimeDelta::Millis(5);包组的间隔

unsigned kDefaultTrendlineWindowSize = 20；计算 trend 的最少包组的数量

统计数据包(msg)的周期=100ms：代码中没有体现，100ms= kSendTimeGroupLength * kDefaultTrendlineWindowSize

传输速率统计周期：[4]中建议[0.5s，1s]，chrome 代码中：kMinRateWindowMs=150ms；kMaxRateWindowMs=500ms。

5.4 其他

1.recovered_from_overuse：用于 probe_controller_,本文不介绍

2.计算传输速率方法:与 BBR 计算一个 rtt 内的传输速率不同，gcc 使用固定计算传输速率区间 150ms,然后对 150ms 内得到的传输速率样本进行贝叶斯估计。

六、GCC 测试以及结果

测试场景与[1]相同

6.1 可用带宽变化场景

6.1.1 场景描述

可用带宽从 500kbps 开始，每 50s 可用带宽增加 500kbps，RTT 设为恒定值 50ms。

6.1.2 实验结果

参考文献

1. G. Carlucci, L. De Cicco, S. Holmer and S. Mascolo, "Congestion Control for Web Real-Time Communication," in IEEE/ACM Transactions on Networking, vol. 25, no. 5, pp. 2629-2642, Oct. 2017, doi: 10.1109/TNET.2017.2703615.
2. M.L. Guerrero Viveros. Performance Analysis of Google Congestion Control Algorithm for WebRTC
3. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/modules/congestion_controller/goog_cc/
4. A Google Congestion Control Algorithm for Real-Time Communication draft-ietf-rmcat-gcc-02
5. Delivery Rate Estimation draft-cheng-iccrg-delivery-rate-estimation-00