11-参考资料-WebRTC的拥塞控制和带宽策略

该部分知识在当前阶段以了解和理解为主。 后续章节再结合源码做进一步分析。

原文地址：WebRTC的拥塞控制和带宽策略 https://mp.weixin.qq.com/s/Ej63-FTe5-2pkxyXoXBUTw

网络的波动带来的卡顿直接影响着用户的体验，在WebRTC中设计了一套基于延迟和丢包反馈的拥塞机制（GCC）和带宽调节策略来保证延迟、质量和网路速度之间平衡，本文中重点是介绍基于trendline滤波的评估模型。本文来自学霸君资深架构师袁荣喜和萍乡学院辛锋的投稿，并由LiveVideoStack全文发布。

文 / 袁荣喜，辛锋

在视频通信的技术领域WebRTC已成为主流的技术标准，WebRTC包涵了诸多优秀的技术，譬如：音频数字信号处理技术（AEC, NS, AGC）、编解码技术、实时传输技术、P2P技术等，这些技术目的都是为了实现更好实时音视频方案。但是在高分辨率视频通信过程中，通信时延、图像质量下降和丢包卡顿是经常发生的事，甚至在WiFi环境下，一次视频重发的网络风暴可以引起WiFi网络间歇性中断，通信延迟和图像质量之间存在的排斥关系是实时视频过程中的主要矛盾。

分析WebRTC是如何解决这个矛盾之前，先来看看我们在在线教育互动的生产环境统计到的视频延迟和人感官的关系，大致如下：

0 ~ 400毫秒	人感觉不到视频在通信过程中的延迟
400 ~ 800毫秒	人能感觉到轻微延迟，但不影响通信互动
800毫秒以上	人能感觉到延迟而且影响通信互动

也就是说，通信过程中最好将视频延迟控制在800毫秒以内。除了延迟，视频图像质量也是个对人感官产生差异的关键因素，我们以640x480分辨率每秒24帧的H264编码情况下视频码率和人感官之间的关系（这组数据是我们通过小范围线上用户投票打分的数据）：

800kbps以上	人对视频清晰度满意,感觉不到视频图像中的信息丢失
480 ~ 800kbps	人对视频清晰度基本满意，有时能感觉到视频图像中的信息丢失
480kbps以下	人对视频清晰度不满意，大部分时候无法辨认图像中的细节信息

从上面的描述可以知道视频质量保持在一个可让人接受的质量范围是需要比较大的带宽码率支持的，如果加上控制延迟，则更需要网络有很好速度和稳定性。但是很不幸，我们现阶段的移动网络和家用WiFi并不是我们想象中的那么好，很难做到在实时视频通信中一个让人非常满意的程度。为了解决以上几个问题，WebRTC设计了一套基于**延迟和丢包反馈**的**拥塞机制**（Google Congestion Control，简称GCC）和**带宽调节策略**来保证**延迟、质量和网路速度**之间平衡，这是一个持续循环过程，如下图：

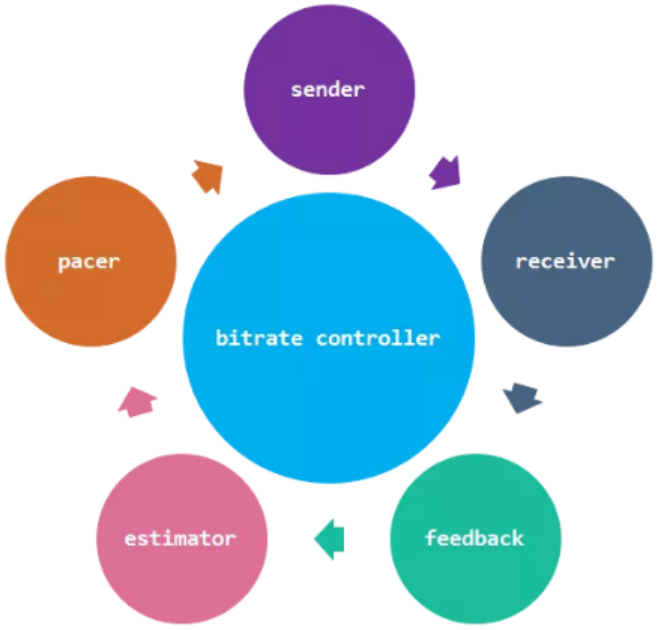


图1：拥塞控制循环示意图

1） **estimator（评估者）**通过RTCP的feedback反馈过来的包到达延迟增量和丢包率信息计算出网络拥塞状态并评估出适合当前网络传输的码率，根据这个码率改变视频编码器码率，然后改变pacer的码率

2） **pacer(定速器)**会根据这个码率改变pacer的网络发送速度和padding比例，并用新的网络发送速度来定时触发发包事件。

3） **sender（发送者）**收到pacer的发送事件，进行RTP报文发送。

4） **receiver（接收者）**接收到RTP报文，进行arrival time统计和丢包统计

5） **feedback（反馈）**定时对receiver统计的信息进行RTCP编码，并反馈到发送端的estimator进行新一轮的码率评估。

> 注：padding包是为了避免码率波动过大，正常情况下静态的场景码率比较低，动态的场景码率比较高，静态的场景加入部分padding包避免码率波动过大。

以上是整个WebRTC拥塞控制和带宽调节过程，下面这个示意图是这个过程涉及到WebRTC内部模块关系。

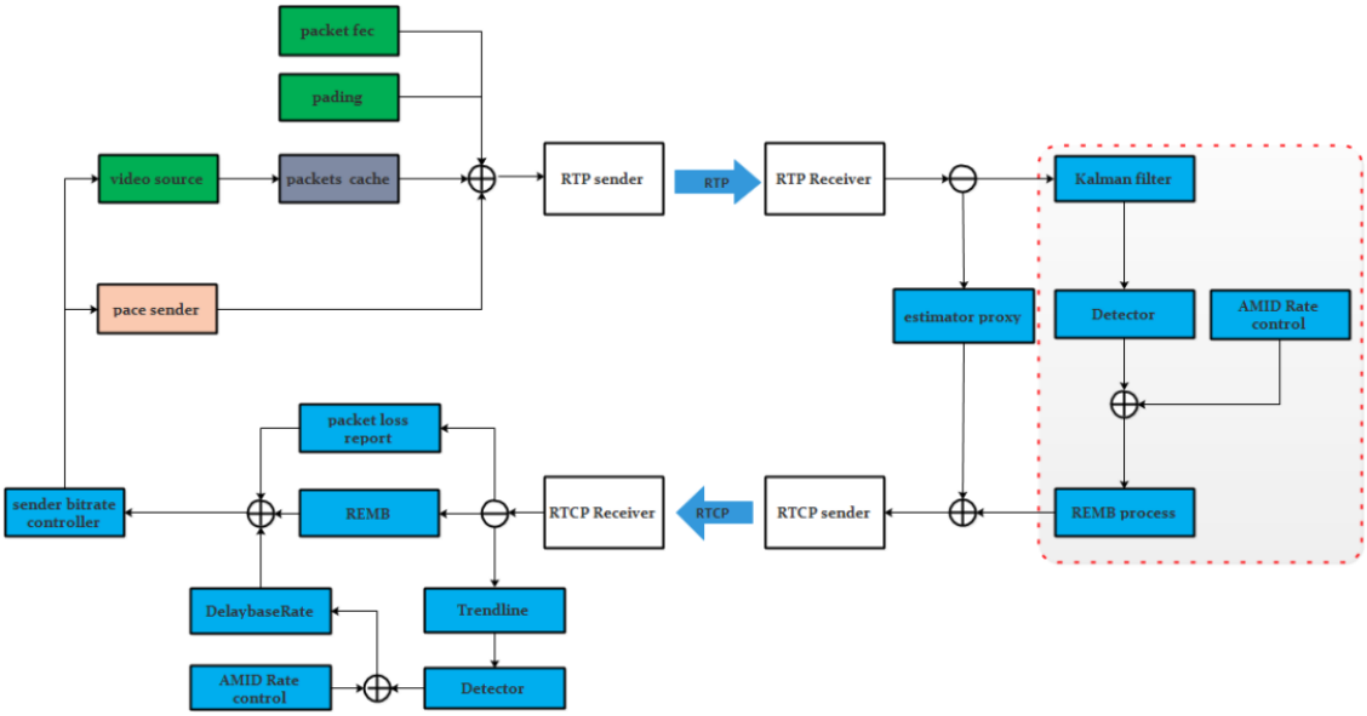


图2：WebRTC的拥塞控制模块关系图

需要说明的是红框中基于接收端的kalman filter带宽评估模型已经在新版本的WebRTC中不采用了，只做了向前版本兼容，新版本的WebRTC都是采用发送端的trendline滤波器来做延迟带宽评估，本文中重点是介绍基于trendline滤波的评估模型，下面依次来分析WebRTC的这五个过程。

1 estimator

estimator的功能就是通过接收端反馈过来的**包到达时刻信息**、**丢包信息**和**REMB信息**进行当前网络状态的码率评估，WebRTC拥塞控制有两部分：**基于延迟的拥塞控制**和**基于丢包的拥塞控制**，它是一个尽力而为的拥塞控制算法，牺牲了拥塞控制的公平性换取尽量大的吞吐量(尽量自己多占用带宽)。从设计结构来描述向它**输入延迟和丢包信息**，它就会输出一个适应当前网络状态的码率值。示意图如下：

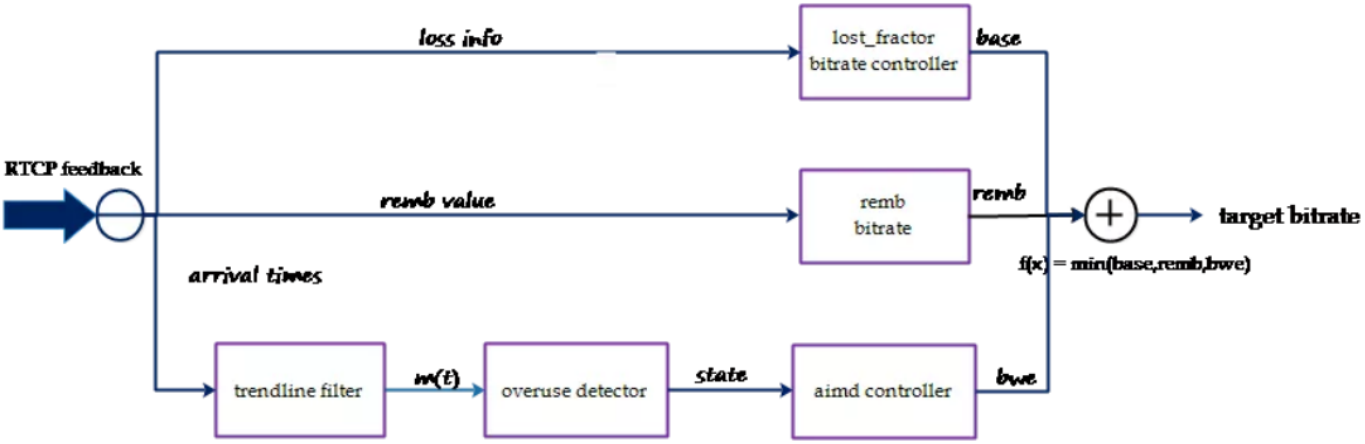


图3：WebRTC的CC estimator输入与输出

从上图可以看出，estimator基于延迟的拥塞控制是通过trendline滤波再进行过载判断，最后根据过载情况进行aimd（Additive Increase Multiplicative Decrease）码率调控评估出一个**bwe（Bandwidth Estimation）** bitrate码率，这个码率会合丢包评估出来的码率和remb（Receiver Estimated Maximum Bitrate）来决定**最后的码率**。

1.1 基于延迟的拥塞控制

基于延迟的拥塞控制是通过**每组包**的到达时间的**延迟差（delta delay）**的增长趋势来判断网络是否过载，如果过载进行码率下调，如果处于平衡范围维持当前码率，如果是网络承载不饱满进行码率上调。
这里有几个关键技术：包组延迟评估、滤波器趋势判断、过载检测和码率调节。

1.1.1 包组与延迟

WebRTC在评估延迟差的时候不是对每个包进行估算，而是采用了包组间进行延迟评估，这符合视频传输（视频帧是需要切分成多个UDP包）的特点，也减少了频繁计算带来的误差。那么什么是包组呢？就是距包组中第一个包的发送时刻t0小于5毫秒发送的所有的包成为一组，第一个超过5毫秒的包作为下一个包组第一个包。为了更好的说明包组和延迟间的关系，先来看示意图：

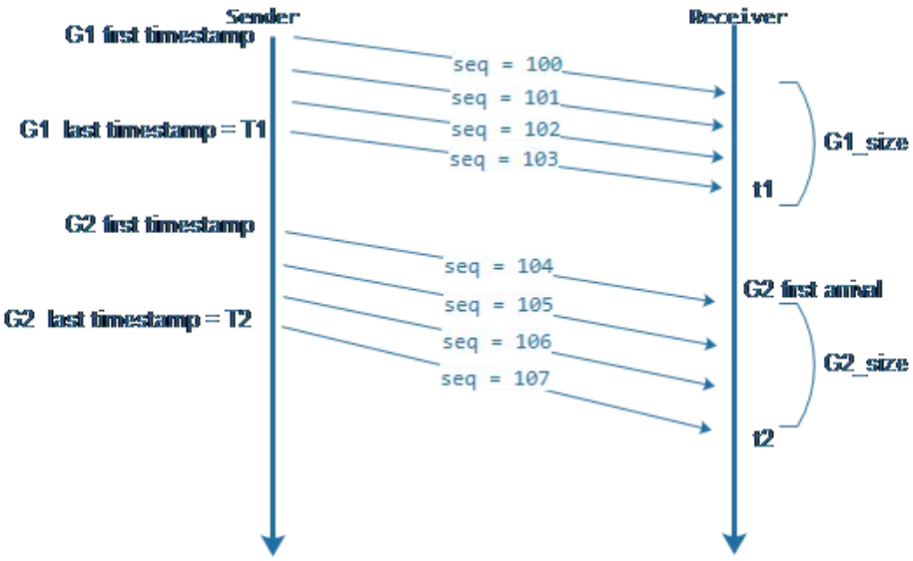


图4：包组与延迟示意图

上图中有两个包组G1和G2，其中第100号包与103号包的时间差小于5毫秒，那么100 ~ 103被划作一个包组。104与100之间时间超过5毫秒，那么104就是G2的第一个包，它与105、106、107划作一个包组。知道了包组的概念，那么我们怎么通过包组的延迟信息得到滤波器要的评估参数呢？滤波器需要的三个参数：

- 发送时刻差值（delta_timestamp）
- 到达时刻差值（delta_arrival）
- 包组数据大小差值（delta_size）。

从上图可以得出：

$$\mathrm{delta}_{\mathrm{timestamp}} = T2 - T1$$
$$\mathrm{delta}_{\mathrm{arrival}} = \mathrm{t2} - \mathrm{t1}$$
$$\mathrm{delta}_{\mathrm{size}} = \mathrm{G2}_{\mathrm{size}} - G1_{size}$$

1.1.2 滤波器

我们通过包组信息计算到了delta_timestamp、delta_arrival和delta_size,那么下一步就是进行数据滤波来评估延迟增长趋势。在WebRTC实现了两种滤波器来进行延迟增长趋势的评估，分别是：kalman filter和trendline filter，从图2中我们知道kalman filter是运行在接收端的，我在这里以不做介绍，有兴趣的可以参考https://www.jianshu.com/p/bb34995c549a。

这里介绍trendline filter(运行在发送端)，我们知道如果平稳网速下**传输数据的延迟时间就是数据大小除以速度**，如果这数据块很大，超过恒定网速下延迟上限，这意味着要它要占用其他后续数据块的传输时间，那么如此往复，网络就产生了延迟和拥塞。Trendline filter通过**到达时间差、发送时间差和数据大小**来得到一个趋势增长值，如果这个值越大说明网络延迟越来越严重，如果这个值越小，说明延迟逐步下降。以下是计算这个值的过程。

先计算单个包组传输增长的延迟，可以记作：

$$\text{delay}_i = delta_{arrival} - delta_{timestamp}$$

然后做每个包组的叠加延迟,可以记作：

$$\text{acc}_{\text{delay\_i}} = \sum delay_0 + delay_1 + delay_2 + \cdots + delay_i$$

在通过累积延迟计算一个**均衡平滑延迟值**，alpha=0.9可以记作：

$$\text{smo}_{\text{delay\_i}} = alpha \times smo_{delay\_i-1} + (1 - alpha) \times acc_{delay\_i}$$

然后统一对累计延迟和均衡平滑延迟再求平均，分别记作：

$$x_i = \frac{\sum acc_{delay\_0} + acc_{delay\_1} + \cdots + acc_{delay\_i}}{i}$$

$$y_i = \frac{\sum smo_{delay\_0} + smo_{delay\_1} + \cdots + smo_{delay\_i}}{i}$$

我们将第i个包组的传输持续时间记作：

$$trans_i = t_i - first\_arrival_i$$

趋势斜率分子值为：

$$numerator_i = \sum_{k=0}^{i} (trans_k - x_k) \times (smo_{delay\ k} - y_k)$$

趋势斜率分母值为：

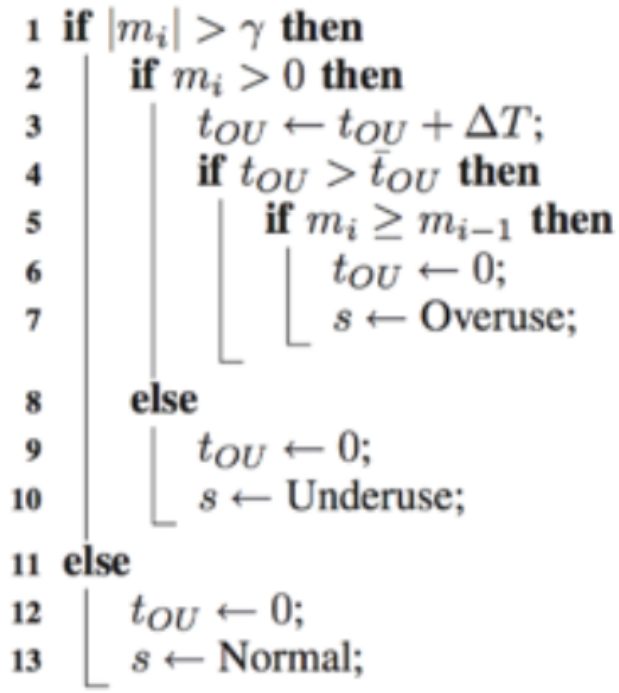
$$denoinator_i = \sum_{k=0}^{i} (trans_k - x_k) \times (trans_k - x_k)$$

最终的趋势值为：

$$trendline_i = \frac{numerator_i}{denoinator_i}$$

1.1.3过载检测

在计算得到trendline值后WebRTC通过**动态阈值gamma_1**进行判断拥塞程度，trendline乘以周期包组个数就是**m_i**,以下是判断拥塞程度的伪代码：



通过以上伪代码就可以判断出当前网络负载状态是否发生了过载，如果发生过载，WebRTC是通过一个有限状态机来进行网络状态迁徙，关于状态机细节可以参看下图：

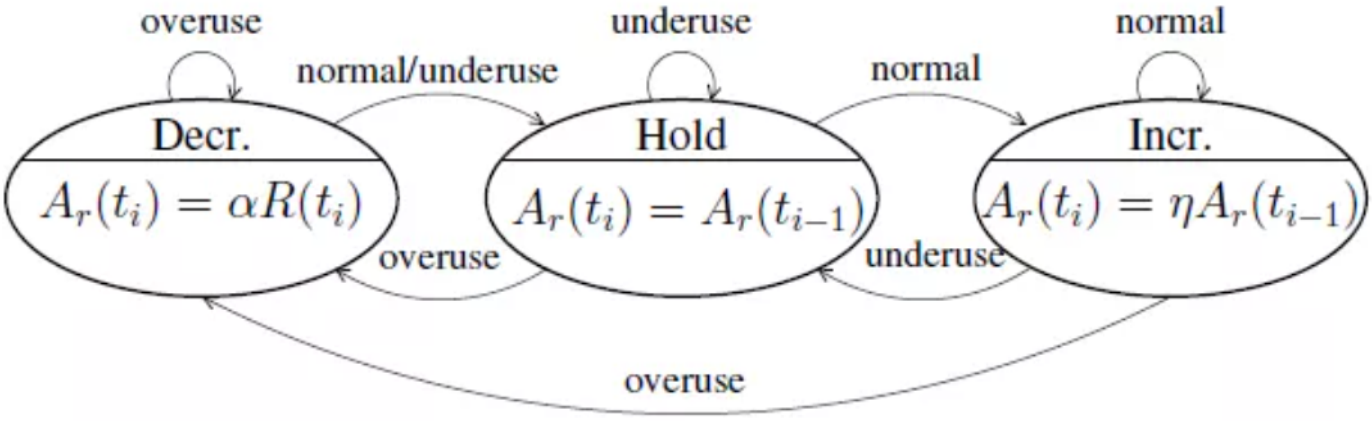


图5:过载检测状态机

从上图可以看出，网络状态机的状态迁徙是由于网络过载状态发生了变化，所以状态迁徙作为了aimd带宽调节的触发事件，aimd根据当前所处的网络状态进行带宽调节，其过程是处于Hold状态表示维持当前码率，处于Decr状态表示需要进行码率递减，处于Incr状态需要进行码率递增。那他们是怎么递增和递减的呢？WebRTC引入了aimd算法解决这个问题。

1.1.4 AIMD码率调节

aimd的全称是Additive Increase Multiplicative Decrease，意思是：和式增加，积式减少。aimd controller是TCP底层的码率调节概念，但是WebRTC并没有完全照搬TCP的机制，而是设计了套自己的算法，用公式表示为：

$$A_r(t_i) = \begin{cases} \eta A_r(t_{i-1}) & \sigma\text{=Increase} \\ \alpha R_r(t_i) & \sigma\text{=Decrease} \\ A_r(t_{i-1}) & \sigma\text{=Hold} \end{cases}$$

- **如果处于Incr状态**，增加码率的方式分为两种：一种是通信会话刚刚开始，相当于TCP慢启动，它会进行一个倍数增加，当前使用的码率乘以系数，**系数是1.08**；如果是持续在通信状态，其增加的码率值是当前码率在一个RTT时间周期所能传输的数据速率。

- **如果处于Decrease状态**，递减原则是:过去500ms时间窗内的最大acked bitrate乘上系数0.85,acked bitrate通过feedback反馈过来的报文序号查找本地发送列表就可以得到。

aimd根据上面的规则最终计算到的码率就是基于延迟拥塞评估到的bwe bitrate码率。

1.2 基于丢包的拥塞控制

除了延迟因素外，WebRTC还会根据网络的丢包率进行拥塞控制码率调节，描述如下：

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

解释下上面的公式：

- **当丢包率<2%时**，这个时候会将码率（base bitrate）增长5%,这个码率(base bitrate)并不是当前及时码率，而是单位时间窗周期内出现的最小码率，WebRTC将这个**时间窗周期设置在1000毫秒内**。因为loss fraction是从接收端反馈过来的，中间会有时间差，这样做的目的是防止网络间歇性统计造成的网络码率增长过快而网络反复波动。
- **当2% < 丢包率 < 10%**,维持当前的码率值
- **当丢包率 >= 10%**, 按丢包率进行当前码率递减，等到新的码率值

丢包率决策出来的码率（base bitrate）只是一个参考值，WebRTC实际采用的带宽是base bitrate、remb bitrate和bwe bitrate中的最小值，这个最小值作为estimator最终评估出来的码率

2 pacer

在estimator根据网络状态决策出新的通信码率（target bitrate），它会将这个码率设置到pacer当中，要求pacer按照新的码率来计算发包频率。因为在视频通信中，单帧视频可能有上百KB，如果

是当视频帧被编码器编码出来后，就立即进行RTP打包发送，瞬时会发送大量的数据到网络上，可能会引起网络衰减和通信恶化。WebRTC引入pacer，pacer会根据estimator评估出来的码率，按照最小单位时间（5ms）做时间分片进行递进发送数据，避免瞬时对网络的冲击。**pacer的目的就是让视频数据按照评估码率均匀的分布在各个时间片里发送，** 所以在弱网的WiFi环境，pacer是个非常重要的关键步骤。以下WebRTC中pacer的模型关系：

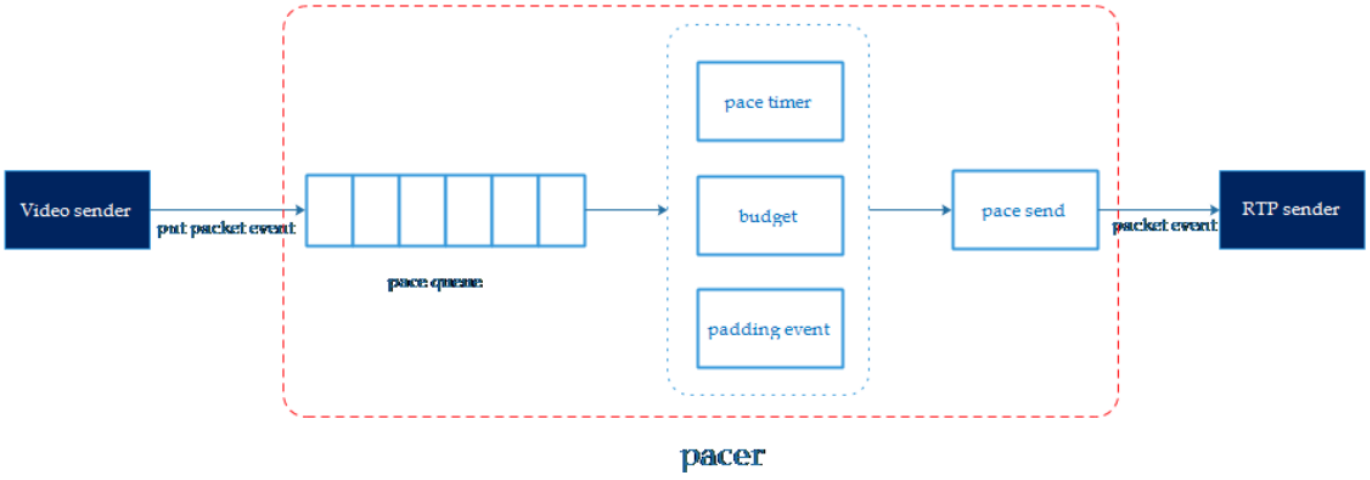


图6：pacer模型图

WebRTC中pacer的流程比较清晰，分为三步：

1. 如果一帧图像被编码和RTP切分打包后，先会将RTP报文存在待发送的队列中，并将报文元数据(packet id, size, timestamp, 重传标示)送到pacer queue进行排队等待发送，插入队列的元数据会进行优先级排序。
2. pacer timer会触发一个定时任务事件来计算budget，budget会算出当前时间片网络可以发送多少数据，然后从pacer queue当中取出报文元数据进行网络发送。
3. 如果pacer queue没有更多待发送的报文，但budget却还可以发送更多的数据，这个时候pacer会进行padding报文补充。

从上面的步骤描述中可以看出pacer有几个关键技术：**pace queue、padding、budget。**

2.1 pace queue与优先级

pace queue是一个基于优先级排序的多维链表，它并不是一个先进先出的fifo，而是一个按优先级排序的list。

报文优先级规则

1. 优先级高的报文排在fifo的前面，低的排在后面。
2. 优先级是最先判断报文的QoS等级，等级越小的优先级越高
3. 其次是判断重发标示，重发的报文比普通报文的优先级更高
4. 再次是判断视频帧timestamp，越早的视频帧优先级更高。

pacer每次触发发送事件时是先从queue的最前面取出优先级最高的报文进行发送，这样做的目的是让视频在传输的过程中延迟尽量小，重传的报文尽快能到达防止等待卡顿。pace queue还可以设置最大延迟，如果超过最大延迟，会计算queue中数据发送所需要的码率，并且会把这个码率替代target bitrate作为budget参考码率来加速发送。

2.2 budget

budget是个评估单位时间内可以发送多少数据量的一个机制，因为pacer是会根据pace timer定时来触发发送检查。Budget会根据评估出来的参考码率计算这次定时事件能发送多少字节，可以表示为：

0.005*3Mbps/8 = 1,966

$$\text{remain bytes} = \text{delta}_{\text{time}} \times target_{bitrate} \div 8$$

- **delta time**是上次检查时间点和这次检查时间点的时间差。
- **target bitrate**是pacer的参考码率，是由estimator根据网络状态评估出来的。
- **remain_bytes**每次触发发包时会减去发送报文的长度size,如果remain_bytes > 0，继续从pace queue中取下一个报文进行发送，直到remain_bytes <=0 或者 pace queue没有更多的报文。

2.3 pacer延迟

那么肯定有人会有疑问pacer queue和budget进定量计算来发送网络报文，相当于cache等待发送，难道不会引起延迟吗？可以肯定的说会引起延迟，但延迟不严重。pacer产生的延迟可以表示为：

$$\text{delay} = \frac{frame\_size \times 8}{bitrate}$$

计算处理一帧数据发送所需要的时间 = 帧数据大小*8/bitrate

假如评估出来的码率是10mbps，一个视频关键帧的大小是300KB，那么这个关键帧造成的**pacer delay**是240毫秒。从实际应用观察到的关键帧引起的pace delay在200 ~ 400毫秒之间，这个值相对于视频传输来说是比较大的，但是不严重。WebRTC为了减少这个延迟，**会评估出尽量大的bitrate**。那么怎么评估出尽量大的码率呢？从前面的estimator描述中我们知道要发送出尽量多的数据才能评估尽量大的码率，但是视频编码器不会发送多余的数据，所以**WebRTC引入了padding机制**来保障发送尽量大的数据来探测网络带宽上限。

限制每路视频的最大比特率，比如3M。对于I帧适当调高这个bitrate

2.4 padding

pace padding除了保障能pace delay尽量小外，它可以让有限的带宽获得尽可能好的视频质量。padding的工作原理很简单，就是在单位时间片内把budget还剩余的空闲用padding数据填满。我个人认为**padding只是适合点对点通信，一旦涉及到多点分发，会因为padding占用很多服务转发带宽，这并不是一件好事情。**

3 sender

WebRTC的发送模块和拥塞控制控制相关的主要是增加了附加的RTP扩展来携带便宜 接收端统计丢包率和延迟间隔的信息、配合pacer的发包策略、带宽分配和FEC策略的信息。

3.1 RTP扩展

首先我们先来复习下RTP固定报头结构：

```
C++ 复制代码
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                               ....                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

可以看到有一个sequence number字段，用于记录RTP包的序列号。一般情况下我们一个传输通道（PeerConnection）只包含一路视频流，这个sequence number能满足大多数需求。但是在一些情况下，我们一个连接可能传输多个视频流，这些视频流复用一个传输通道，例如simulcast或者single PC场景，一个PeerConnection可能包含多个不同的视频流。在这些视频流中，RTP报头的sequence number是单独计数的。

这里举个例子，假设同一个PeerConnection下，我们传输两个视频流A与B，它们的RTP包记为Ra(n)，Rb(n)，n表示sequence number，这样我们观察同一个PeerConnection下，视频流按如下形式传输：

- Ra(1), Ra(2),Rb(1),Rb(2),Ra(3),Ra(4),Rb(3),Rb(4)

在对某条PeerConnection进行带宽估计时，我们需要估计整条PeerConnection下所有视频流，而不是单独某个流。这样为了做一个RTP session（传输层）级别的带宽估计，原有各个流的sequence number就不能满足我们需要了。

为此Transport-cc中，使用了RTP报头扩展，用于记录**transport sequence number**，同一个PeerConnection连接下的所有流的transport sequence number，使用统一的计数器进行计数，方便进行同一个PeerConnection下的带宽估计。

这里我们使用前面的例子，视频流A与B，它们的RTP包记为Ra(n，m)，Rb(n,m)，n表示sequence number，m表示transport sequence number，这样同一个PeerConnection下，视频流按如下形式传输：

- Ra(1,**1**), Ra(2,**2**),Rb(1,**3**),Rb(2,**4**),Ra(3,**5**),Ra(4,6),Rb(3,7),Rb(4,8)

这样进行带宽估计时，通过transport sequence number我们就能关心到这条传输通道下所有数据包的情况了。

WebRTC为了配合接收端进行延迟包序列和丢包统计做了下列扩展：

- **transport sequence**number传输通道的只增sequence，每次发送报文时自增长，配合接收端统计丢包、通过反馈这个sequence可以计算得到发包的时刻。

- **TransmissionOffset**发送报文的相对时刻，这个相对时刻值t是发送报文的绝对时刻T1和视频帧时间戳T0差值。早期的WebRTC是在接收端进行estimate bitrate，所以过载判断是在接收端完成的，这个值就是为了kalman filter计算发包造成的延迟用的，新版本还携带这个值以便低版本的WebRTC能兼容。

3.2 packet cache

packet cache是一个key/value结构的包缓冲池，视频帧在进行RTP分片打包后不会立即发送出去，而是要等待pacer的发送信号进行发送。所以打包后会按[id,packet]键值对插入到packet cache中。一般packet cache会保存600个分片报文，最大9600个，插入新的会将最旧的报文删除（超过了覆盖），packet cache这样做的目的除了配合pacer发送外，也为了后面响应nack的丢包重传。

3.3 NACK与丢包重传

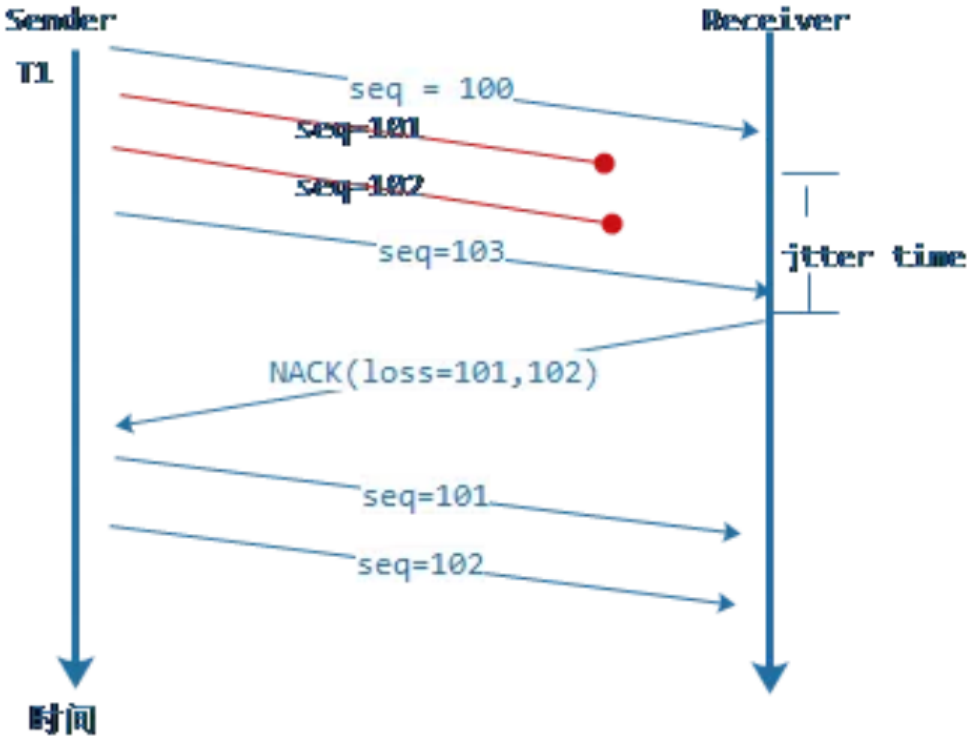


图7:RTP NACK过程的示意图

WebRTC在评估到收发端之间RTT延迟比较小的时候会采用NACK来进行丢包补偿，NACK是一个请求重发过程，其流程如上图所示。这个过程有一个问题是在网络抖动和丢包很厉害的情况下有可能造成同一时刻收到很多NACK的重传请求，发送端瞬间把这些重传请求放入pacer中进行重发，这样pacer的延迟会增大，而且pace的参考码率会随着pace queue的延迟控制变的很大而出现间歇性网络风暴。WebRTC在处理NACK重传时设计了一个**重传码率控制器**，其设计原理是通过统计单位时间窗口周期中发送的字节数据来限流，如果这个时间窗内发送的数据的码率大于estimator评估的码率，不进行当前NACK请求的重传，等待下一个NACK。

3.4 FEC与码率分配

WebRTC应对丢包时除了NACK方式，在收发端之间RTT很大时候会开启FEC来进行丢包补偿，我们在这里不介绍FEC具体算法，只介绍FEC的码率分配策略。从整个通信机制我们很容易得出这样一个共识：

$$\text{target bitrate} = \text{video bitrate} + \text{feedback bitrate} + \text{FEC bitrate} + \text{padding bitrate}$$

FEC bitrate到底应该设置多大呢？它先根据feedback中反馈过来的丢包率(loss fraction)来确定使用哪一种FEC，在根据每种FEC和丢包率来确定FEC使用的码率，**但需要满足以下条件**:

$$\text{FEC bitrate} < \text{target bitrate} / 2$$

feedback的码率被设定为target bitrate的5%，WebRTC是通过控制feekback的频率来进行调控分配的。padding bitrate是通过pacer queue和budget来控制的。**Target bitrate减去这些码率之和就是给视频编码器的码率。每次estimator评估出来码率后，会先进行这些计算得到最后的video bitrate，**并将这个值作为编码器的编码码率，以此来达到防止拥塞的目的。

4 receiver

receiver模块的工作相对来说比较简单，它就做三件事情：记录

1. **每个报文的到达时刻(arrival timestamp)**、
2. **丢包率(lost fraction)**
3. **receiver bitrate**。

4.1 报文到达时间

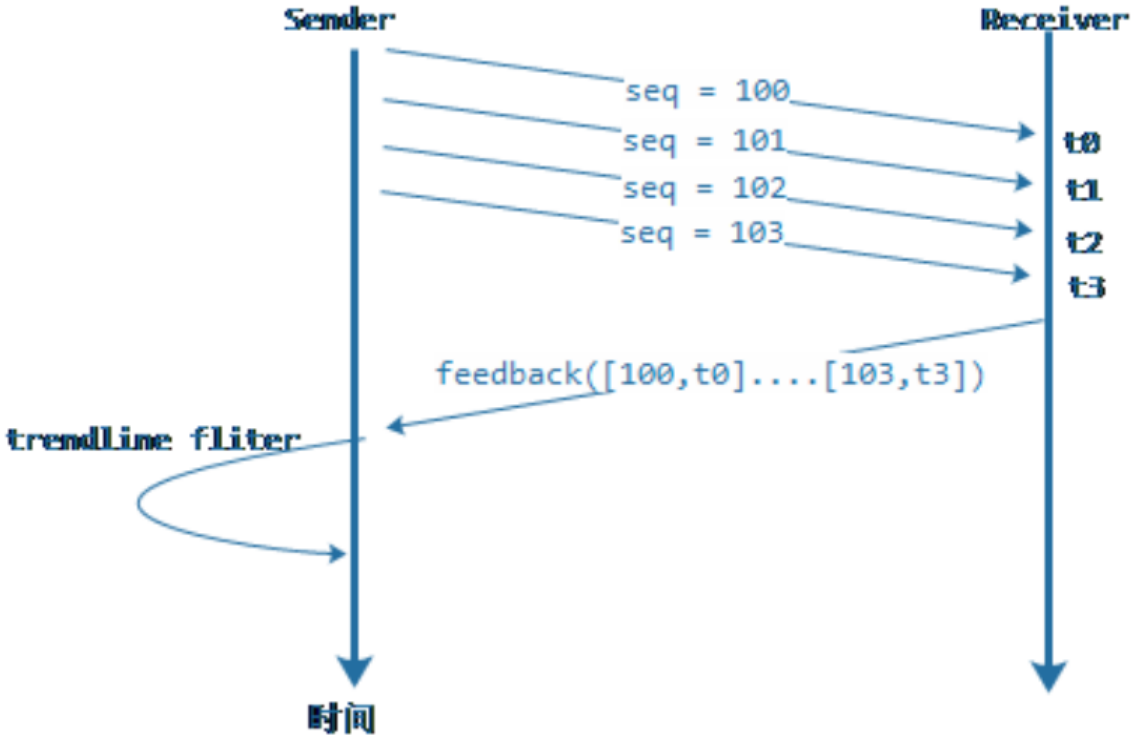


图8：到达报文统计图

上图是一个统计RTP报文到达时刻的序列图，图中的seq是**RTP扩展中的transport sequence**,接收端用一个k/v（[seq,arrival timestamp]）键值对数据结构来保存最近**500毫秒**未反馈的到达时刻信息，通过**时间窗口周期**来进行淘汰老的到达时刻记录。

4.2 丢包率计算

丢包率计算过程是这样的，我们把上次统计丢包率时刻的最大sequence记着**prev_seq**，把当前收到的最大sequence记着**cur_seq**,当前统计丢失的报文记着count，WebRTC在RTCP中描述丢包率采用的是uint8，为了保证精确度将256记着100%的丢包率，那么很容易得：

$$\text{loss fraction} = \frac{count \times 256}{cur_{seq} - prev_{seq}}$$

这里需要提的是WebRTC在统计报文是否丢失是通过sequence的连续性和**网络的jitter时间**来确定的，只有落在jitter抖动范围之外的丢包才是算是作丢包。

4.3 接收码率统计

接收端码率统计采用的是最近单位时间窗口（1000毫秒）周期内收到的的字节数来计算，WebRTC设计了一个1毫秒为最小单位的窗口数组来进行统计，每个最小单位是数字，这个数字是在这个时刻收到的网络数据大小，大致的示意图如下：

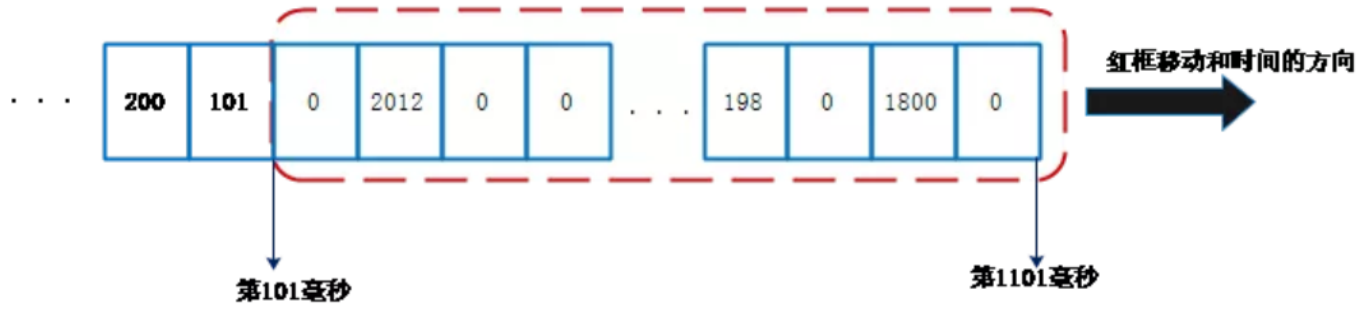


图9：接收码率统计示意图

计算码率只需要将红框中所有的数字加起来，当时间发生改变后，就红框就向右移动并且填写新时刻接收到的数据大小，等下一个统计时刻既可。

5 feedback

前面介绍的estimator依赖于feedback反馈的报文到达时刻和丢包率来进评估码率的，也就是说feedback需要将这些信息及时反馈给接收端，主要是记录的**报文到达时刻、通道丢包率和remb带宽**。因为报文到达时刻和丢包率统计都是多个数据项，WebRTC利用了report block来进行编码存放。为了有效的利用RTCP的report block空间，WebRTC采用了相对时间转换和位压缩算法来对到达时间序列做编码压缩。

除了report编码，**feedback的周期也很重要**，如果是单纯的remb反馈，一般是1秒一次反馈。但如果是需要反馈报文的到达时间，它会根据占用5%的target bitrate来计算发送feedback的时间间隔，计算流程如下：

$$\text{feedback}_{\text{bitrate}} = \text{target}_{\text{bitrate}} \times 5\%$$
$$\text{feedback}_{\text{size}} = \text{IPv4}_{\text{header}} + \text{UDP}_{\text{header}} + \text{SRTP}_{\text{header}} + \text{report\_block}_{\text{size}}$$
$$\text{feedback}_{\text{interval}} = \frac{feedback_{size}}{feedback_{bitrate}}$$

feedback interval需要满足一个条件：50ms < interval < 250ms,这个条件中的 50ms< internal 是为了防止interval太小造成发送feedback太过频繁而消耗网络性能，而**interval < 250ms是为了防止feedback频次太低造成estimator反应迟钝（反馈的信息已经迟滞）**。

6 总结

以上就是WebRTC拥塞控制和码率调节策略的5个过程，里面涉及到很多传输相关的技术，我在这里也是简单介绍了下其工作原理，很多细节的并没有描述出来，也很难描述出来，有兴趣的同学可以翻看WebRTC的源代码。如果觉得webRTC代码费劲，我照虎画猫将WebRTC的拥塞控制用C重新实现了个简易版本,但是去掉了padding，可到https://github.com/yuanrongxi/razor下载。

6.1 效果

WebRTC的GCC在网络适应上表现还是比较良好的，既然兼顾延迟，也能兼顾丢包，网络发生拥塞时在2 ~ 3秒内能评估出相对的码率来适应当前的网络状态，但是会造成短时间的卡顿。对于网络发生间歇性丢包，在2秒左右能将传输码率适配到当前网络状态。它在网络相对稳定且延迟较大的网络进行高分辨率传输时，视频很稳定，适合长距离延迟稳定的网络环境。在弱网环境下，WebRTC容易将码率降到很低而造成图像失真。

6.2 网络大抖动

对于乱序和抖动WebRTC的拥塞控制显得有点无力，如果抖动超过rtt*2/3时，基于kalman filter的带宽评估机制不起作用（不知道是不是我用错了）；基于trendline滤波的评估机制波动很大，敏感度不够，不能完全反应当前的网络过载状态，尤其是在终端Wi-Fi拥挤的情况下，比较容易造成间歇性风暴。

6.3 延迟问题

WebRTC的pacer在传输大分辨率视频时，关键帧会引起大约200毫秒的延时，尤其是在移动4G网络下这个问题更加明显，海康威视工程师郑鹏提出了用H.264的intre_refresh模式来应对，在测试过程中确实比较适合WebRTC用来减少关键帧造成的延迟，但是intre_refresh是普通模式编码CPU的3倍左右，而且很多移动设备的编码器不一定支持。

总之，WebRTC的拥塞控制存在反应慢、怕抖动的特性，但是这块也是WebRTC改进最为频繁的模块，几乎每个版本都有新的改进。要彻底解决这样的问题，需要从视频编码器和网络传输进行融合来解决，以后我用单独的篇幅来介绍下这样的解决方案。