WebRTC 基于 GCC 的拥塞控制(上) - 算法分析

实时流媒体应用的最大特点是实时性，而延迟是实时性的大敌。从媒体收发端来讲，媒体数据的处理速度是造成延迟的重要原因；而从传输角度来讲，网络拥塞则是造成延迟的最主要原因。网络拥塞可能造成数据包丢失，也可能造成数据传输时间变长，延迟增大。

拥塞控制是实时流媒体应用质量保证(QoS)的重要手段之一，它在缓解网络拥堵、减小网络延迟、平滑数据传输等质量保证方面发挥重要作用。WebRTC 通控制发送端数据发送码率来达到控制网络拥塞的目的，其采用谷歌提出的拥塞控制算法(Google Congestion Control，简称 GCC[1])来控制发送端码率。

本文在研究 GCC RFC 文档和论文的基础上，详细分析 WebRTC 源代码中 GCC 算法的实现，力图对 WebRTC 的 QoS 有一个完整深入的认识。

1 GCC 算法综述

Google 关于 GCC 的 RFC 文档在文献[1]，该 RFC 目前处于草案状态，还没有成为 IETF 的正式 RFC。此外，Google 陆续发布了一系列论文[2][3][4]来论述该算法的实现细节，以及其在 Google Hangouts、WebRTC 等产品中的应用。本节根据这些文档资料，从理论上学习 GCC 算法。

GCC 算法分两部分：发送端基于丢包率的码率控制和接收端基于延迟的码率控制。如图 1 所示。

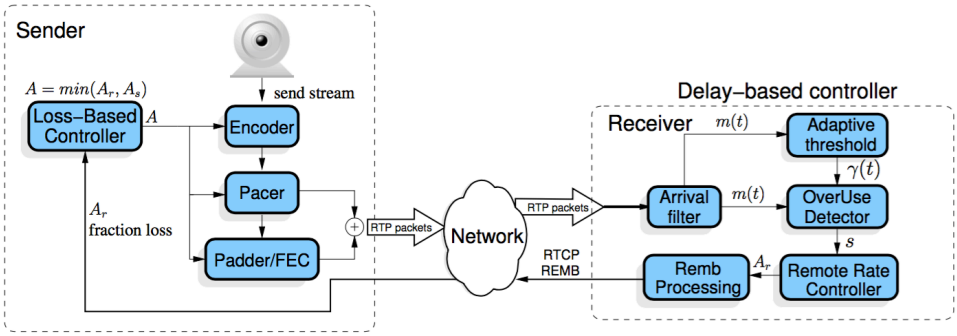

图 1 GCC 算法结构

WebRTC 在发送端收到来自接收端的 RTCP RR 报文，根据其 Report Block 中携带的丢包率信息，调整发送端码率 As。在接收端，WebRTC 根据数据包到达的时间延迟，通过到达时间滤波器，估算出网络延迟 m(t)，然后判断网络的拥塞状况；最后根据算法规则计算出瞬时接收码率 Ar，通过 RTCP REMB 报文返回发送端，发送端以 Ar 作为最终码率的上限。

2 发送端基于丢包率的码率控制

GCC 算法在发送端基于丢包率控制发送码率，其基本思想是：丢包率反映网络拥塞状况。如果丢包率很小或者为 0，说明网络状况良好，在不超过预设最大码率的情况下，可以增大发送端码率；反之如果丢包率变大，说明网络状况变差，此时应减少发送端码率。在其它情况下，可维持发送端码率保持不变。

GCC 使用的丢包率根据接收端 RTP 接收统计信息计算得到，通过 RTCP RR 报文中返回给发送端。RTCP RR 报文统计接收端 RTP 接收信息，如 Packet Loss，Jitter，DLSR 等等，如图 2 所示：

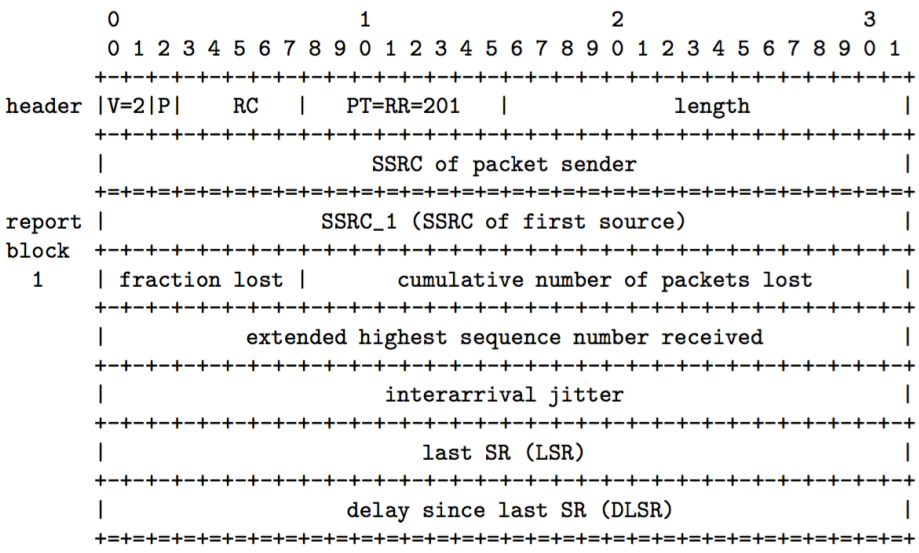

图2 RTCP RR报文结构[5]

发送端收到RTCP RR报文并解析得到丢包率后，根据如下公式[4]计算发送端码率：当丢包率大于0.1时，说明网络发生拥塞，此时降低发送端码率；当丢包率小于0.02时，说明网络状况良好，此时增大发送端码率；其他情况下，发送端码率保持不变。

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

图3 GCC基于丢包率的码率计算公式

最终码率会作用于 Encoder 模块和 PacedSender 模块，用以在编码器内部调整码率和平滑发送端发送速率。

3 接收端基于延迟的码率控制

GCC 算法在接收端基于数据包到达延迟估计发送码率 Ar，然后通过 RTCP REMB 报文反馈到发送端，发送端把 Ar 作为最终目标码率的上限值。其基本思想是：RTP 数据包的到达时间延迟 m(i)反映网络拥塞状况。当延迟很小时，说明网络拥塞不严重，可以适当增大目标码率；当延迟变大时，说明网络拥塞变严重，需要减小目标码率；当延迟维持在一个低水平时，目标码率维持不变。

基于延时的拥塞控制由三个主要模块组成：到达时间滤波器，过载检查器和速率控制器；除此之外还有过载阈值自适应模块和 REMB 报文生成模块，如图 1 所示。下面分别论述其工作过程。

3.1 到达时间滤波器(Arrival-time Filter)

该模块用以计算相邻相邻两个数据包组的网络排队延迟 m(i)。数据包组定义为一段时间内连续发送的数据包的集合。一列数据包短时间里连续发送，这段时间称为突发时间，建议突发时间为 5ms。不建议在突发时间内的包间隔时间做度量，而是把它们做为一组来测量。通过相邻两个数据包组的采集时间和到达时间，计算得到组间延迟 d (i)。组间延迟示意图及计算公式如图 4 所示：

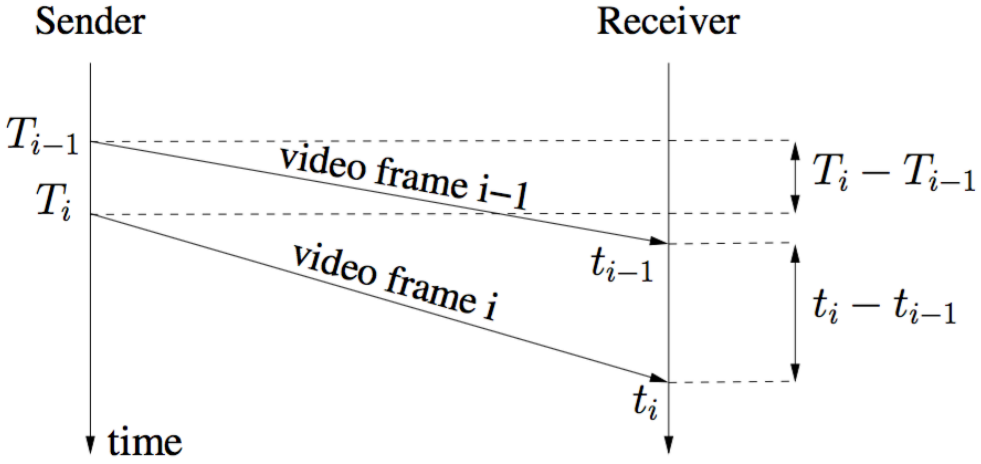

图 4 组间延迟示意图

T(i)是第 i 个数据包组中第一个数据包的发送时间，t(i)是第 i 个数据包组中最后

一个数据包的到达时间。帧间延迟通过如下公式计算得到：

$$d(i) = t(i) - t(i-1) - (T(i) - T(i-1)) \qquad (1.3.1)$$

公式 1.3.1 是 d(i)的观测方程。另一方面，d(i)也可由如下状态方程得到：

$$d(i) = dL(i)/C(i) + w(i) \qquad (1.3.2)$$

$$d(i) = dL(i)/C(i) + m(i) + v(i) \qquad (1.3.3)$$

其中 dL(i)表示相邻两帧的长度差，C(i)表示网络信道容量，m(i)表示网络排队延迟，v(i)表示零均值噪声。m(i)即是我们要求得的网络排队延迟。通过 Kalman Filter 可以求得该值。具体计算过程请参考文献[1][4][6]。

3.2 过载检测器(Over-use Detector)

该模块以到达时间滤波器计算得到的网络排队延迟 m(i)为输入，结合当前阈值 gamma_1，判断当前网络是否过载。判断算法如图 5 所示[2]。

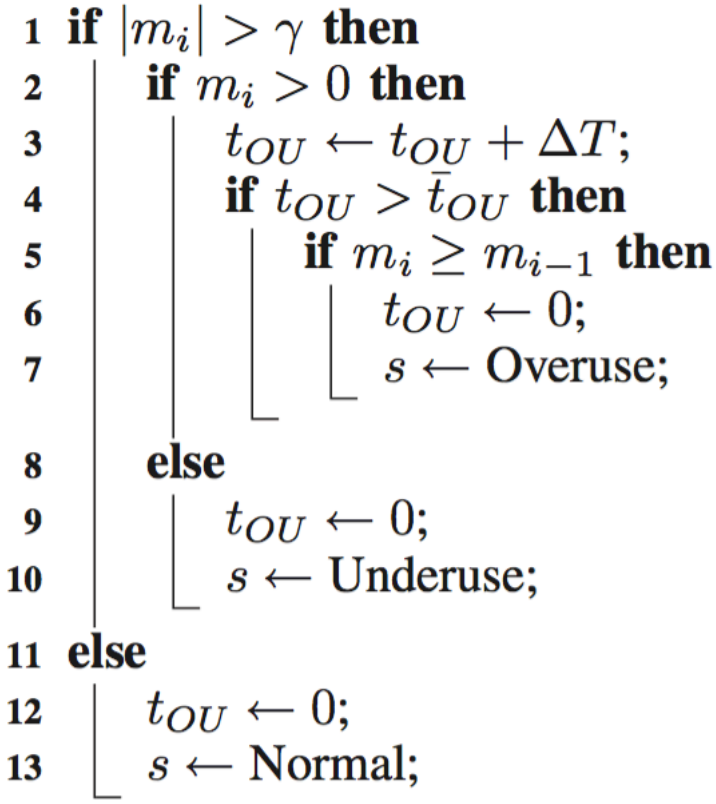

图 5 过载检测伪代码

算法基于当前网络排队延迟 m(i)和当前阈值 gamma_1 判断当前网络拥塞状况[2]：当 m(i) > gamma_1 时，算法计算处于当前状态的持续时间 t(ou) = t(ou) + delta(t)，如果 t(ou)大于设定阈值 gamma_2(实际计算中设置为 100ms)，并且 m(i) > m(i-1)，则发出网络过载信号 Overuse，同时重置 t(ou)。如果 m(i)小于 m(i-1)，即使高于阀值也不需要发出过载信号。当 m(i) < -gamma_1 时，算法认为当前网络处于空闲状态，发出网络低载信号 Underuse。当 – gamma_1 <= m(i) <= gamma_1 是，算法认为当前网络使用率始终，发出保持信号 Hold。算法随着时间轴的计算过程可从图 6 中看到。

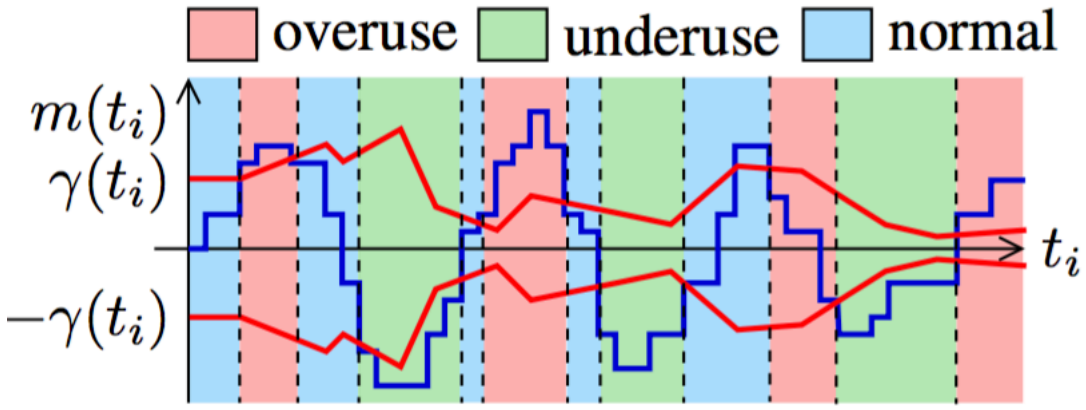

图 6 时间轴上的过载检测过程

需要注意的是，阀值 gamma_1 对算法的影响很大，并且阈值 gamma_1 是自适应性的。如果其是静态值，会带来一系列问题，详见文献[4]。所以 gamma_1 需要动态调整来达到良好的表现。这就是图 1 中的 Adaptive threshould 模块。

阈值 gamma_1 动态更新的公式如下：

gamma_1(i) = gamma_1(i-1) + (t(i)-t(i-1)) * K(i) * (|m(i)|-gamma_1(i-1)) (1.3.4)

当|m(i)|>gamma_1(i-1)时增加 gamma_1(i)，反之减小 gamma_1(i)，而当|m(i)| - gamma_1(i) >15 ， 建 议 gamma_1(i) 不 更 新 。 K(i) 为 更 新 系 数 ， 当 |m(i)|<gamma_1(i-1)时 K(i) = K_d，否则 K(i) = K_u。

同时建议 gamma_1(i)控制在[6,600]区间。太小的值会导致探测器过于敏感。建议增加系数要大于减少系数 K_u > K_d。文献[1]给出的建议值如下：

gamma_1(0) = 12.5 ms

gamma_2 = 10 ms

K_u = 0.01

K_d = 0.00018

3.3 速率控制器(Remote Rate Controller)

该模块以过载检测器给出的当前网络状态为输入，首先根据图 7 所示的有限状态机判断当前码率的变化趋势，然后根据图 8 所示的公式计算目标码率 Ar。

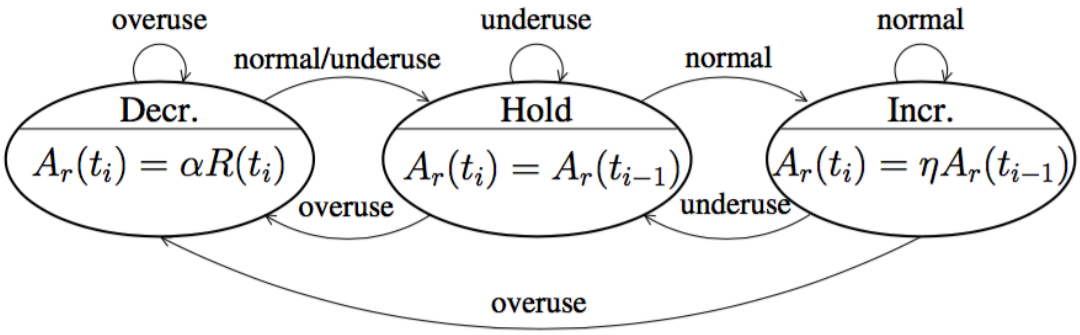

图 7 目标码率 Ar 变化趋势有限状态机

当前网络过载时，目标码率处于 Decrease 状态；当前网络低载时，目标码率处于 Hold 状态；当网络正常时，处于 Decrease 状态时迁移到 Hold 状态，处于 Hold/Increase 状态时都迁移到 Increase 状态。当判断出码率变化趋势后，根据图 8 所示公式进行计算目标码率。

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

图 7 目标码率 Ar 计算公式

当码率变化趋势为 Increase 时，当前码率为上次码率乘上系数 1.05；当码率变化趋势为 Decrease，当前码率为过去 500ms 内的最大接收码率乘上系数 0.85。当码率变化趋势为 Hold 时，当前码率保持不变。

目标码率 Ar 计算得到之后，下一步把 Ar 封装到 REMB 报文中发送回发送端。REMB 报文是 Payload 为 206 的 RTCP 报文[7]，格式如图 8 所示。

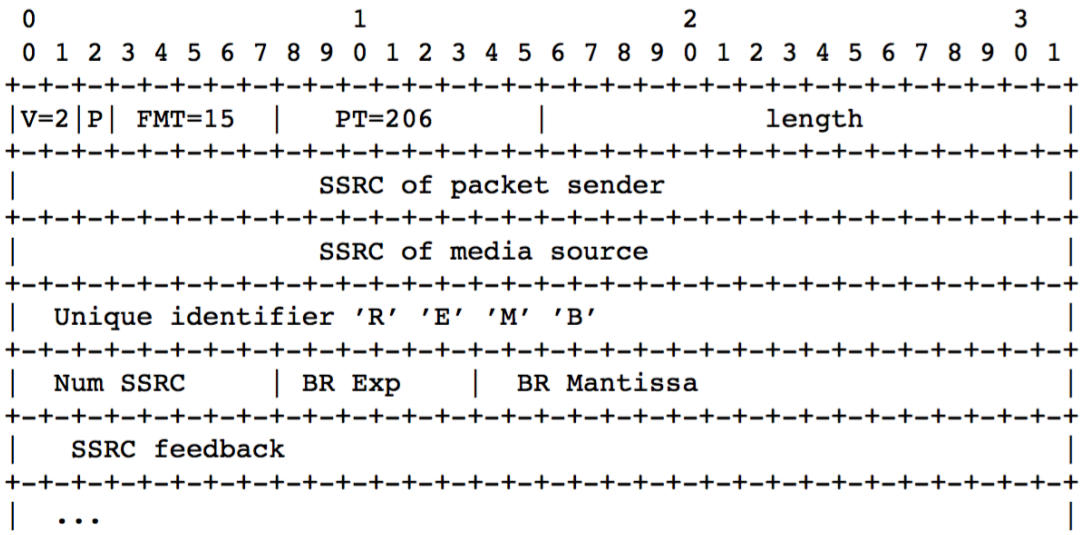

图 8 REMB 报文格式

REMB 报文每秒发送一次，当 Ar(i) < 0.97 * Ar(i-1)时则立即发送。

3.4 发送端目标码率的确定

发送端最终目标码率的确定结合了基于丢包率计算得到的码率 As 和基于延迟计算得到的码率 Ar。此外，在实际实现中会配置目标码率的上限值和下限值。综合以上因素，最终目标码率确定如下：

target_bitrate = max(min(min(As, Ar), Amax), Amin)

目标码率确定之后，分别设置到 Encoder 模块和 PacedSender 模块。

4 总结

本文在广泛调研 WebRTC GCC 算法的相关 RFC 和论文的基础上，全面深入学习 GCC 算法的理论分析，以此为契机力图对 WebRTC 的 QoS 有一个全面直观的认识。为将来深入 WebRTC 源代码内部分析 GCC 的实现细节奠定基础。

参考文献

[1] A Google Congestion Control Algorithm for Real-Time Communication. draft-alvestrand-rmcat-congestion-03

[2] Understanding the Dynamic Behaviour of the Google Congestion Control for RTCWeb.

[3] Experimental Investigation of the Google Congestion Control for Real-Time Flows.

[4] Analysis and Design of the Google Congestion Control for Web Real-time Communication (WebRTC). MMSys'16, May 10-13, 2016, Klagenfurt, Austria

[5] RFC3550: RTP - A Transport Protocol for Real-Time Applications

[6] WebRTC 视频接收缓冲区基于 KalmanFilter 的延迟模型. http://www.jianshu.com/p/bb34995c549a

[7] RTCP message for Receiver Estimated Maximum Bitrate. draft-alvestrand-rmcat-remb-03