# WebRTC 源代码分析

**wzw**

# 目录

第一章 WebRTC 概述

1.1 WebRTC 概述

1.2 WebRTC 源码下载

1.3 GYP 构建系统

1.4 WebRTC 编译

1.5 WebRTC 开发流程

1.6 总结

1.7 参考文献

第二章 WebRTC 总体架构及源码目录

第三章 PeerConnection Demo 示例程序

第四章 PeerConnection 及 API

第五章 信道建立过程及 SDP

第六章 WebRTC MediaEngine

第七章 Video Data Pipeline

第八章 Audio Data Pipeline

第九章 RTP、RTCP 和 QoS

第十章 网络传输

第五章 Javascript API

总结

# 第一章 WebRTC 概述

## 1.1 WebRTC 概述

WebRTC(Web Real Time Communication)是一项免费开源工程，旨在通过简单的 API 接口向浏览器和移动应用提供实时通信的能力。WebRTC 的组件经过充分优化以达成这个目标。WebRTC 的使命：为浏览器、移动平台和物联网设备开发功能丰富、高质量的实时通信应用，并通过一系列通用协议使这些应用互联通信。

第七章 Video Data Pipeline

本章按照 Video 数据从采集到渲染显示的过程，分别介绍各个模块。这包括视频采集模块、视频预处理模块、视频编码、视频发送、视频接收、视频解码、视频渲染显示模块。

## 7.1WebRTC 的模块机制

WebRTC 具有相对独立性的功能组织称模块，比如音频采集模块，RTP 传输模块，视频编解码模块等。这提高了模块的高内聚性，并且模块间的通信和合作接口很清晰。目前 WebRTC 的主要模块包括：

audio_coding：音频编码模块

audio_conference_mixer：音频混音模块

audio_device：音频设备及采集模块

audio_processing：音频处理模块

bitrate_controller：发送端码率控制模块

congestion_controller：拥塞控制模块

desktop_capture：桌面抓取模块

media_file：媒体文件模块

pacing：发送端平滑发送模块

remote_bitrate_estimator：接收端码率估计模块

rtp_rtcp：RTP／RTCP 模块

video_capture：视频采集模块

video_coding：视频编解码模块

video_processing：视频预处理模块

7.2 视频采集模块

视频采集

7.3 视频编码模块

视频编码

7.4 视频引擎

视频视频引擎

7.5 视频发送

视频发送

7.6 视频接收

视频接收

7.5 视频解码

视频解码

7.5 视频渲染

视频渲染显示

# 第九章 RTP／RTCP 和 QoS

## 9.1 RTP／RTCP

### 9.1.1 理解 RTP 报头中的时间戳

Timestamp in rtp header – RFC3550.Section 5.1

时间戳反映了RTP数据包中第一个字节的采样时间。采样时间必须从一个单调线性递增的时钟中获取，以便允许RTP同步和抖动计算。时钟必须足够精确以支持预期同步精度和测量数据包到达抖动，因此每个视频帧一个滴答显然不够。时钟频率依赖于负载数据的格式，数据格式可由档次或者负载格式规格静态确定，或者通过非RTP方法动态确定。如果RTP数据包周期性产生，那么采样时间由采样时钟确定，而不是由系统时钟确定。例如，对于固定速率的音频来说，时间戳时钟会在每个采样周期内增加一。如果一个音频应用每160个周期从输入设备中读取数据块，那么时间戳会每次增加160，而不管该数据块是通过数据包传输还是丢弃)。

时间戳初始值应该像序列号那样是一个随机值。一些连续的RTP数据包如果逻辑上在同一时刻产生，比如属于同一视频帧，那么它们应该具有相同的时间戳。如果数据包没有按照它们的采样时间先后顺序进行传输，那么连续的RTP数据包可能包含非单调递增的时间戳，比如在MPEG中差值产生的视频帧；然而数据包的序列号依然是线性递增的。

来自不同媒体流的RTP时间戳可能以不同的速率增长，因此它们有各自独立随机的偏移量。因此，尽管对于单个媒体流来说比较时间戳足以重建时间序列，但是对于不同媒体流来说通过比较RTP时间戳来进行同步不是有效的方法。相反，对于每一个媒体流来说，RTP时间戳和参考时钟时间戳是配对的：参考时钟时间戳用来表示和RTP时间戳相关的RTP数据的采样时刻，参考时钟由所有同步源共

享。时间戳对 并不会在每个数据包中都传输，但是以一个较低频率在RTCP SR数据包中进行传输。

采样时刻被用来选作RTP时间戳的参考点，因为它对另一个传输端是已知的，并且对所有媒体有一个公共定义，独立于编码延迟和其它处理。这样做的目的是允许对同一时刻采样的所有媒体进行同步表示。

传输非实时数据的应用通常使用源于参考时钟的时间作为虚拟显示时间线，以决定何时显示下一帧数据或者其它数据单元。在这种情况下，RTP时间戳反映数据单元的显示时间，也就是说，每个数据单元的RTP时间戳和参考时钟时间相关联，在这个时间点上数据单元在虚拟显示时间线上变为当前显示单元。事实上显示时间相比于接收端确定的时间稍微延迟。

一个描述 预先录制视频的直播音频解说的例子说明了选择采样时间作为RTP时间戳参考点的意义。在这种场景下，视频将在本地呈现给解说者观看，并同时使用RTP进行发送传输。在RTP中传输的视频帧的采样时间参考视频帧呈现给解说者的参考时钟时间进行建立。包含解说者声音的RTP音频包的采样时刻同样参考相同的参考时钟的时间进行建立。音频和视频甚至有可能在不同主机上传输，如果这两台主机的参考时钟采用像NTP这样的方式同步。接收者可以使用RTCP SR数据包中的时间戳对来同步音视频的显示时间。

## 9.2 WebRTC PacedSending 工作原理

PacedSender线程工作在视频数据编码之后、发送到网络之前，用以平滑发送速率，减小数据burst造成的抖动。PacedSender线程由上层配置一个发送码率N bps，以30ms为时间间隔发送数据，单位间隔内发送的设定数据量由 1000 * N * 30得到。如果本次间隔发送的数据量超过设定数据量，那么在下次间隔不发送或者少发送，以达到平滑发送速率之目的。

以30ms为操作间隔IntervalBudget，每个间隔有固定的media budget，该budget根据设定的target_bps计算的到。一个interval内buget如果没用完，那么下个interval

就不能用；一个interval内的发送量如果超过budget，那么下个interval就不发或者少发，直到这个坑填平再开始发送。在某些情况下，还通过需要发送padding使数据量维持在min_bps以上，而实际上min_bps设定为0，也就是说不发送padding数据。

pacedsending的process()的调用间隔为每6~16ms调用一次。

## 9.3 QoS初探

### 9.3.1 理论基础

算法：Google Congestion Control

RFC draft: A Google Congestion Control Algorithm for Real-Time Communication

Paper: Experimental Investigation of the Google Congestion Control for Real-Time Flows

Paper: Understanding the Dynamic Behavior of Google Congestion Control for WebRTC

RFCs：

rfc3550: RTP - A Transport Protocol for Real-Time Applications

rtc3611: RTP Control Protocol Extended Reports (RTCP XR)

rfc4585: Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF)

Rfc5104: Codec Control Messages in the RTP Audio-Visual Profile with Feedback (AVPF)

#### 9.3.2 GCC 算法流程

##### 9.3.2.1 GCC 总体流程

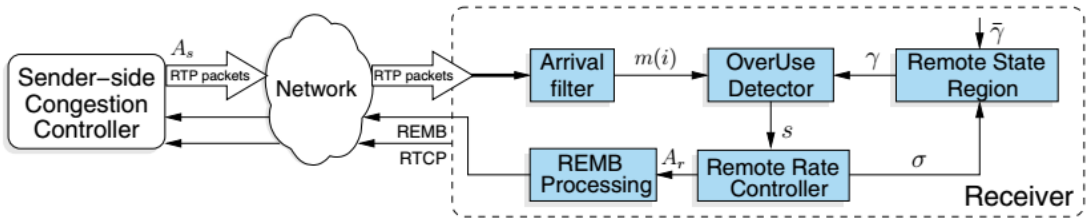



图 1 GCC 算法总体流程

#### 9.3.2.2 GCC 算法细节

文档：矛与盾-互联网音视频通信的抗丢包与带宽自适应

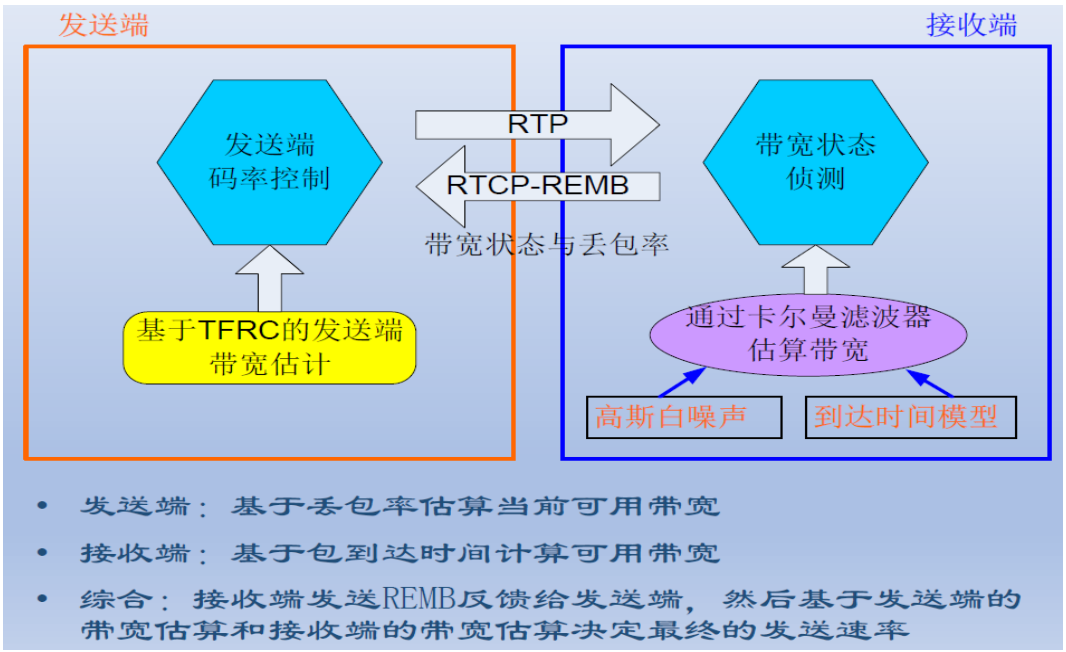



图 2 GCC 的带宽自适应模型

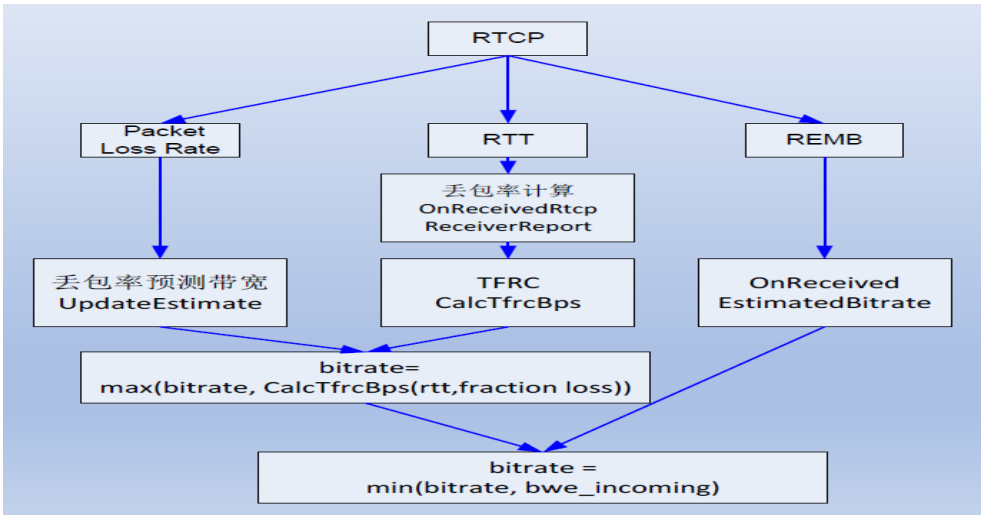



图 3 发送端流程

发送端：

1. 基于丢包带宽估算，丢包信息来自RTCP RR(Receiver Report )

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

2. 计算TCP-Friendly Send Rate

$$A_{TF}=\frac{8\times s}{R\times\sqrt{2bp/3}+RTO\times 3\times\sqrt{3bp/8}\times p\times(1+32p^2)}$$

3. 根据接收端发送的REMB包确定接收端的接收速率

$$BW_{in}=BitrateFromREMB$$

4. 确定发送速率As

$$A_s\le\min(BW_{in},Max\_Bitrate\_Configured)$$

$$A_s\ge\max(Min\_Bitrate\_Configured,A_{TF})$$

图 4 发送端带宽计算

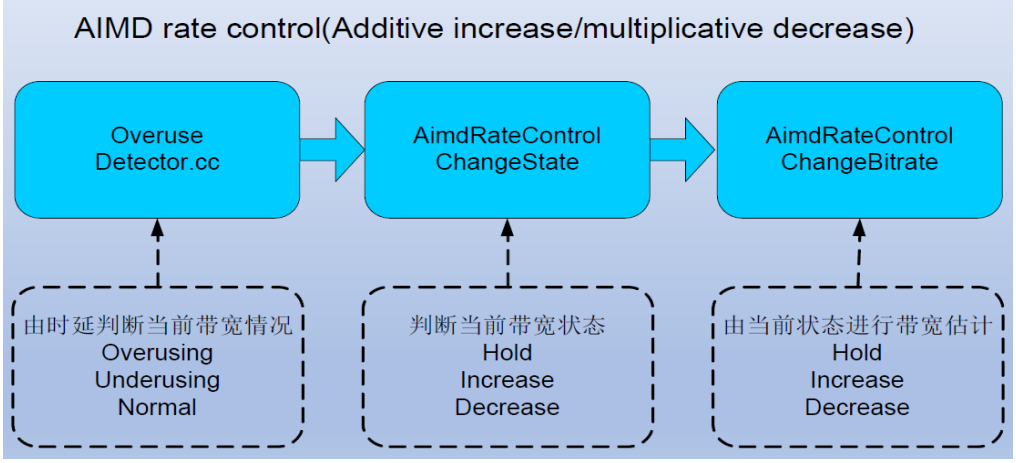
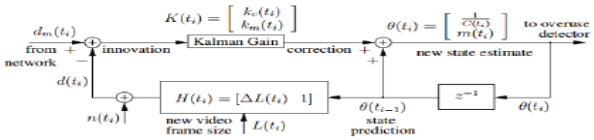



图 5 接收端带宽估计

接收端：

1. Arrival Filter

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

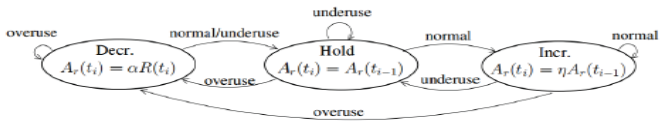



2. Overuse Detector

产生驱动信号控制Rate Controller模块的状态机转移

$$s=\begin{cases}overuse, m(t_i)>\gamma\\ underuse, m(t_i)<\gamma\end{cases}$$

3. Rate Controller




4. REMB Processing

- 将估算的Ar通过REMB发送至发送端，间隔为1s
- 当Ar下降超过0.03，立刻发送REMB

图 6 接收端带宽计算

### 9.3.3 WebRTC 中 GCC 算法的实现

#### 9.3.3.1 功能模块

WebRTC 中关于 QoS 实现的代码分布在若干模块中，各模块功能及相互关系记录如下：

**call**：表示一个会话实例，包含多个 A/V stream，开启两个线程 module_process、pacer_thread，集成 BitrateAllocator、CongestionController 对象，继承实现 PackerReceiver 和 BitrateObserver 接口。功能上：1）接收 RTP 数据包，交给底层 recv stream 处理；2）接受来 RTCP 数据包，交给底层 recvstream 处理;

3）处理和 QoS 相关的 REMB 报文和 ReceiverReport 报文，则在底层解析之后通过 observer 回调到 RtcpBandwidthObserver 对象，该对象通过发送端带宽控制算法计算出最新带宽，把结果反馈到 VideoCodecModule 模块和 PacerSend 模块，进而控制 video codec 码率和 smooth send 发送速率。

**BitrateController/BitrateAllocator**：实现 GCC 基于丢包率的发送端带宽控制算法。call 对象把接收到的 Rtcp 报文向下交给 RtcpReceiver，RtcpReceiver 首先对报文内容进行解析，然后触发关心 rtcp 报文内容的回调函数，对于 remb/tmmbr/rr 报文，其回调对象是 RtcpBandwidthObserver，该对象的 OnReceivedEstimatedBitrate()接口用于处理 remb/tmmbr 报文内容，OnReceivedRtcpReceiverReport()接口用于处理 rr 报文内容。BitrateController 模块根据丢包率重新计算出当前带宽，并通过调用 MaybeTriggerOnNetworkChanged()接口向其观察者(也就是 call 对象)发送 OnNetworkChanged()消息。Call 对象在收到 OnNetworkChanged()消息之后，分别通知 VCM 和 PacedSender 执行码率控制策略。

**Pacing**：pacedsender 线程实现，起到平滑网络发送速率的作用。Media data 在经过 codec 编码之后，并在发送到 network 之前，经过该 pacedsender 线程控制速率。Pacedsender 线程接收来自 Call 对象的最新估计带宽控制，然后据此控制 RTP 数据包发送到网络的速率。

**RemoteBitrateEstimator**：接收端带宽控制算法实现。基于 AIMD 算法控制带宽。Call 接收到 RTP 数据包后，在 ViEReceiver 中首先会将 RTP 数据包信息交给 RemoteBitrateEstimator 进行带宽估计，然后把 RTP 数据包转到 VCM 进行解码操作，最后更新接收端统计信息，为发送 ReceiverReport 报文更新统计数据（如 fractionLoss 等）。RemoteBitrateEstimator 基于数据包的延时进行码率估计，判断当前带宽的使用状况。整个工作过程在一个有限状态机中进行，如图 7 所示。经过 RemoteBitrateEstimator 估计后会发送 REMB 报文（如果达到发送条件），而经过接收端数据统计之后，则由 RTPRTCP 模块定期发送 ReceiverReport 报文。

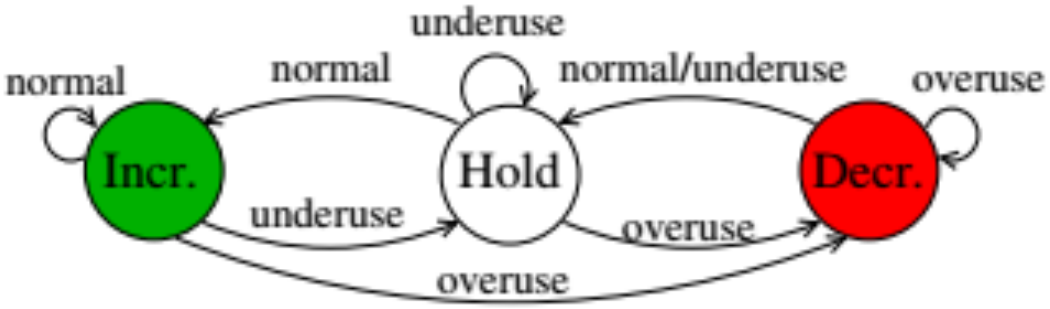


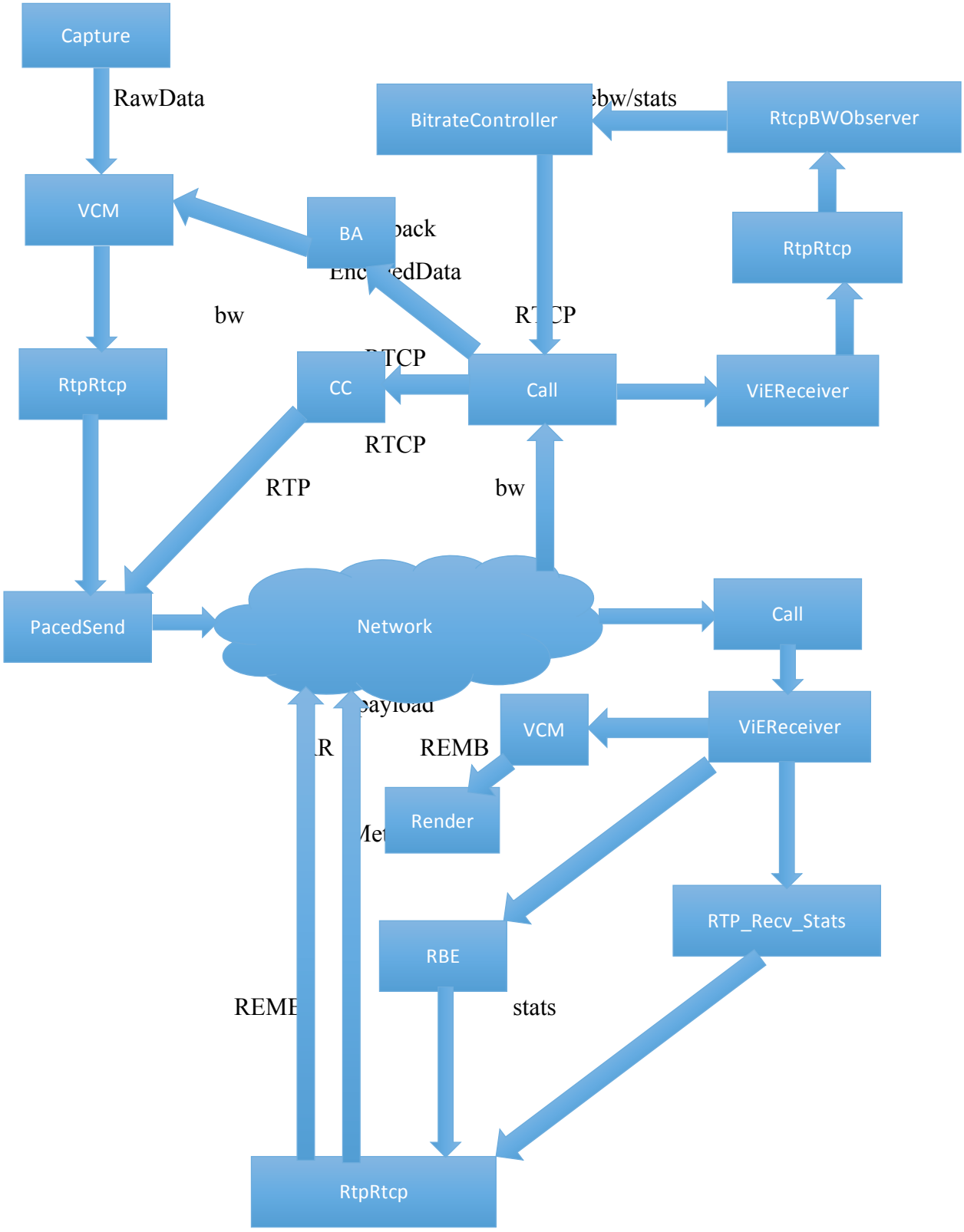

图 7 接收端带宽估计有限状态机

**CongestionController：** 拥塞控制算法的总控模块，同时兼顾发送端的码率控制 BitrateController 和接收端的码率估计 RemoteBitrateEstimator。在发送端，更新后的码率会经过 CongestionController 反馈到 PacedSender 线程。作为模块，CongestionController 在函数 Process()中驱动 BitrateController 和 RemoteBitrateEsitimator 模块的 Process()函数的执行。

**RTP_RTCP：**该模块 RTP/RTCP 报文的构建、发送、接收、解析以及上层回调等功能。Media data 在 codec 编码之后，会经过该模块打包成 RTP 模块发送到网络；接收端在收到 RTP 数据包之后，会经过该模块解包后发送到 codec 进行解码工作；并且同事进行 QoS 的带宽估计和数据统计等工作。RTRTCP 模块的 Process()函数会定期（每秒）发送 RTCP 报文到接收端。接收端在收到 RTCP 报文后会由 RTPRTCP 模块解包获得 rtcp 信息，并由回调函数交由上层处理。

#### 9.3.3.2 总体流程图

如下页流程图所示：


Capture
RawData
VCM
BA
BitrateController
bw/stats
RtcpBWObserver
RtpRtcp
bw
RTCP
RtpRtcp
CC
Call
ViEReceiver
RTCP
RTP
bw
PacedSend
Network
Call
VCM
ViEReceiver
REMB
Render
RBE
RTP_Recv_Stats
stats
RtpRtcp

### 9.3.4 总结

本文结合 Google Congestion Control 算法和相关 RFC 文档，对 WebRTC 中 QoS 的实现代码进行了分析：初步理清了发送端基于丢包率和对端估计码率的带宽控制策略和接收端基于数据包到达时间和丢包率的带宽估计策略；对整个基于 RTCP 报文的控制回路有一个较为清晰的认识。但是目前对于 GCC 算法细节的代码实现还需要进一步深入研究。