GCC的必要性

在实时视频会议、语音通话等应用场景中，“低延迟”和“足够的带宽”是非常关键的需求。传统的 TCP 协议并不适合直接用于这类实时通信流量，主要原因可归纳如下：

1. **TCP 的可靠性与按序交付机制导致延迟累积**
   TCP 在设计上强调可靠性和顺序性，出现丢包时会执行重传和滑动窗口阻塞，保证“零丢包、按序交付”。但对实时场景而言，视频或语音数据即使出现某些丢包，也往往不值得花过多时间等待重传，因为“实时到达”比“完全正确”更重要。TCP 过度追求可靠性与顺序性的特征，使得在一旦出现丢包的链路上，延迟会显著上升，对音视频通话的用户体验影响很大。
2. **TCP 探测带宽方式导致排队时延波动**
   TCP 通过丢包信号来探测可用带宽，会周期性地让队列填满（造成时延上升），然后通过丢包触发拥塞回退，使得队列逐渐被清空（时延下降）。这会产生明显的时延振荡。对视频会议等延迟敏感的业务而言，大幅度的时延波动会引起画面卡顿、音频延迟，造成用户体验不佳。
3. **实时通信要“轻量且快速适应网络”**
   视频会议的码率需要灵活跟随网络状况波动，并尽量避免把网络拥堵缓冲塞满（否则延迟急剧增大）。传统 TCP 依赖丢包和 RTT 来调整发送窗口，适配速度和机制并不理想；对于视频通话，更希望基于端到端延迟（尤其是“排队延迟”）的细微变化来及时进行发送码率调整，从而始终保持低排队、低延迟，同时又能占到合理的带宽。

基于以上原因，Google 针对在 UDP 上传输的 RTP 媒体流，研发了 GCC（Google Congestion Control）算法。它的“必要性”主要体现在以下几方面：

1. **专门适配“实时、低延迟”场景**
   相比传统 TCP 的拥塞窗口调度，GCC 更关注端到端排队时延的微小变化，并使用自适应阈值来探测和判断拥塞。通过快速探测网络状态并避免始终灌满队列，可以将时延控制在较低水平，减少在视频会议中常见的卡顿感。
2. **兼顾对网络带宽的高效利用**
   GCC 一方面针对延迟敏感场景做出了优化，另一方面也需要探测到合理的可用带宽，并尝试尽量接近它，以提供尽量清晰和流畅的视频画面。它在“低延迟”与“高带宽利用率”之间做了综合平衡。
3. **在复杂网络环境下保持适应性**
   GCC 并不仅局限于同类流量（如同是 WebRTC 的多路流），还需要与 TCP、短连接、以及其他基于丢包的流量公平竞争。它的设计里，既包含基于延迟梯度的算法，也融入了基于丢包的降速策略，从而在网络环境出现并发的异构流量时，不至于被 TCP 或其他流完全挤占，也不会过度抢占带宽。

综上所述，之所以不能直接使用 TCP 并需要研发 GCC，根本原因在于：**视频会议类的实时交互流量需要快速适应网络变化并维持低排队延迟，而 TCP 的可靠性机制和基于丢包的带宽探测在这些场景下会造成严重延迟、卡顿与体验下降。** 因此，Google 的 GCC 算法顺应了实时媒体传输的特殊需求，通过在端到端测量排队时延梯度来检测拥塞，并结合自适应阈值与丢包辅助控制，实现了对视频流码率的敏捷控制和对网络带宽的充分利用。

实时通信场景下的相关工作

在论文的第 2 节（相关工作）中，作者回顾了在广域网环境下设计针对实时通信（尤其是视频会议）流量的拥塞控制方案时所面临的挑战，并简要介绍了已有的典型研究工作。可将其总结为以下几个方面：

1. **基于 RTT 的拥塞检测**
   - 早期很多拥塞控制算法直接基于端到端往返时延（RTT）来判断网络拥塞情况，如 TCP Vegas [3]、TCP FAST [30] 等。
   - 这类方法通常需要预估或测量 RTT 的统计信息，并设置一个延迟“目标值”或阈值，用以判断是否进入拥塞状态。
   - 但在存在反向流或与传统“基于丢包”的 TCP 流竞争时，单纯使用 RTT 往往会面临测量误差大、与对端流公平性不足等问题。
2. **基于单向延迟（One-Way Delay）的拥塞检测**
   - 一些研究转而使用“单向延迟”来降低对反向路径流量的敏感度，例如 LEDBAT [26]、TCP-LP [18]。
   - 以 LEDBAT 为例，它通过维持一个固定的单向延迟目标值来“让路”于传统的 TCP 流，达到“低优先级”数据传输的目的。
   - 不过，这类算法有时会遭遇所谓“迟到者效应”问题：多个流共享同一瓶颈时，后加入的流可能“饿死”先加入的流（或反之），导致带宽分配不平衡。
3. **基于延迟梯度（Delay Gradient）的拥塞检测**
   - 为克服如 LEDBAT 等算法中出现的“迟到者效应”，一些工作开始采用“延迟梯度”来进行拥塞判断，例如 CDG [14]、Verus [34]。
   - 这些算法关注的是延迟在一段时间内的增减趋势（即梯度），从而可以更及时地探测到拥塞即将发生或缓解的动态过程。
   - 也有研究在数据中心场景中利用网卡硬件时间戳做更精准的延迟梯度检测 [19]，对实时调度具有积极意义。
4. **其他思路（如机器学习、随机方法、FEC 等）**
   - Sprout [32] 提出了基于随机建模的拥塞控制方案，能在单流场景中实现低延迟和较高吞吐量，但在多流共享瓶颈时的公平性尚未深入评估。
   - Remy [31] 使用先验网络知识与机器学习训练来自动生成拥塞控制算法，通过定义利用率函数使其自适应特定环境需求。
   - FBRA [20] 则把“发送 FEC 探测带宽”与“丢包时冗余恢复”结合，尝试保证视频传输的连续性。
5. **RTP/RTCP 下的拥塞控制**
   - 随着 WebRTC 在浏览器端实时通信的普及，在 RTP/RTCP（基于 UDP）的传输层做端到端拥塞控制日益受到关注。
   - IETF RMCAT 工作组中已有多项端到端算法提案：

1. NADA [35] 结合了丢包率、单向时延等多重指标；
2. SCREAM [17] 部分借鉴了 LEDBAT 的延迟目标思路；
3. Google Congestion Control（GCC）[15] 则基于延迟梯度，利用卡尔曼滤波获取单向延迟变化并做自适应阈值控制。

总的来说，“相关工作”部分主要突出以下要点：

- 实时通信需要关注低延迟、低抖动，以及带宽利用率和与其他流公平性间的综合平衡；
- 不同算法在测量指标上各有侧重：有些基于 RTT，有些基于单向延迟，有些基于延迟梯度；
- 在普适的互联网环境下，与基于丢包的流量（如传统 TCP）同时存在是常态，因而任何基于延迟的拥塞控制都要考虑与传统流量竞争时的公平性与鲁棒性；
- WebRTC 及 IETF RMCAT 工作组正积极推动在 RTP/RTCP 层的端到端拥塞控制研究，GCC 即为其中的典型代表。

GCC框架

GCC整体框架概览

在这篇文章中，GCC（Google Congestion Control）可以被视作一个“端到端”的拥塞控制方案，整体由以下两大核心控制环节和若干辅助模块构成：

1. **基于延迟的控制器（接收端）**
   - 该部分在接收端运行，用来估计是否出现瓶颈队列积压（即延迟增加）。
   - 它通过“到达时间滤波器”对接收到的视频包做时间戳比较，得到“单向延迟梯度”信息，并结合一个“自适应阈值”机制来判断当前网络是否处于“过度使用”（overuse）状态。
   - 一旦检测到过度使用，远程端会通过 RTCP 报文（REMB 消息等）把“推荐发送速率”反馈给发送端。
2. **基于丢包的控制器（发送端）**
   - 该部分在发送端侧运行，主要根据 RTCP 中的丢包率（或分数）信息来补充性地调整发送速率。
   - 当丢包率较高时，发送端降低码率；丢包很少时，发送端适度地将速率向上探测。

此外还有若干关键组件或机制：

- **远程速率控制（Remote Rate Control）状态机**：根据延迟检测器输出的状态信号（正常 / 过度使用 / 不足使用），决定反馈给发送端的目标码率应“加大”“减小”还是保持。
- **REMB 反馈处理**：当接收端给出新的建议速率时，会立即或周期性地用 RTCP 扩展报文发送给发送端，以便发送端得知最新的带宽估计。
- **Pacer 与编码器调度**：发送端在得到新的目标发送码率后，通过 Pacing 机制把包以更平滑、更接近目标速率的方式发出去；编码器也会依据目标码率做相应编码速率调整。

总体而言，GCC 的框架就是：**接收端基于延迟梯度来估计并反馈网络拥塞状况 → 发送端结合丢包信息和接收端反馈综合调整实际发送速率 → 通过 Pacer 等手段将实际数据流严格控制在目标码率附近，从而在低延迟与合理带宽使用间取得平衡。**

延迟梯度概念理解

- 在论文的第 3.1 节“排队延迟梯度”中，作者从理论上阐述了**为什么要通过“延迟梯度”而非“绝对延迟”来判断网络中是否正在发生拥塞。**以下是该部分的详细思路和推导过程。

---

1. 定义：排队延迟 $T_q(t)$

- **基本概念**
  假设瓶颈链路的发送容量（带宽）为 (C)（单位：比特/秒），某个时刻该瓶颈处的队列长度为 (q(t))（单位：比特），那么**排队延迟**可视作队列中所有比特被按带宽 (C) 发送完所需的时间：

  $$T_q(t) = \frac{q(t)}{C}.$$

  这里的 $T_q(t)$ 表示了**单向**的排队时延贡献（不含传播时延等其他部分），可以简单理解为“当前队列还有多少秒可发送完”。
- **含义**
  $T_q(t)$ 一旦变大，说明瓶颈处缓存中排队的比特变多；如果 $T_q(t)$ 接近 0，表示网络基本没有排队。对于实时业务（如视频会议），理想情况下希望排队时延很小，从而保证端到端延迟低。

---

2. 排队延迟梯度：$\dot{T}_q(t)$

- **队列变化率与延迟梯度**
  如果把 $T_q(t)$ 当作一个时间函数，那么它的导数（时间上的变化率）可写为：

$$\dot{T}_q(t) \;=\; \frac{d\,T_q(t)}{dt} \;=\; \frac{d}{dt}\Big(\frac{q(t)}{C}\Big) \;=\; \frac{\dot{q}(t)}{C}.$$

  其中 $\dot{q}(t) = \frac{dq(t)}{dt}$ 表示队列长度随时间的变化率（比特/秒）。

- **如何表示 ($\dot{q}(t)$)?**
  论文将瓶颈队列的变化速率建模为：

$$\dot{q}(t) = \begin{cases} r(t) \;-\; C, & 0 \;\le\; q(t) \;\le\; q_M, \\ 0, & \text{否则}, \end{cases}$$

  这里：

  - r(t) 是**进入队列的有效速率**（单位：比特/秒），即发送方对该瓶颈的实际输入速率。
  - C 是链路容量。
  - q_M 表示队列的最大长度上限（即队列满时不再继续增长，而会丢包）。
  - 若队列已经为空（ $q(t) = 0$ ）或满了（ $q(t) = q_M$ ），则进一步的出队或入队都不再让 q(t) 有效变化，所以 $\dot{q}(t) = 0$。

---

3. 解释：$\dot{T}_q(t) = 0$ 的三种情形

根据上式可知，如果 $\dot{T}_q(t) = 0$，说明 $\dot{q}(t) = 0$。这在理论上可能出现在以下三种情况下：

1. **队列为空时（利用率不足）**
   当发送速率 r(t) 低于容量 C 一段时间后，队列会被完全清空，此时 q(t) = 0，继续发小于 C 的速率也不会再积累队列。
   - 这意味着网络带宽没有被充分利用。
2. **队列已满时（严重拥塞）**
   当 $r(t) \gg C$ 并持续增长，则队列会达到上限 q_M。到达满队列后，再多余的数据包会被丢弃（假设漏尾队列等），导致 q(t) 维持在满状态且不再增长。
   - 这意味着网络严重过载，排队延迟将非常大，甚至丢包。
3. **输入速率与链路容量"刚好匹配"**
   当 r(t) = C （且 0 < q(t) < q_M ）时，队列长度保持某个常数而不再变化，($\dot{q}(t) = 0$)。
   - 这被称为"驻留队列 (standing queue)"，会持续引入固定的排队时延，不是实时应用希望看到的情况。因为即便它不再增加，依旧是个"稳定的、较大的排队延迟"。

---

4. 为什么要基于"梯度"而非单纯看"延迟大小"

论文指出，如果仅看当前延迟大小 $T_q(t)$，当队列持续但稳定地保持在中高水位时（"驻留队列"），**延迟不会继续变大**，从单纯的绝对延迟角度可能误判"没有拥塞动态变化"。但实际上，这说明**链路已经产生了显著排队**，对实时业务极为不利。

- **梯度 $\dot{T}_q(t)$ 的价值**
  如果我们检测到 $\dot{T}_q(t)$ 连续为**正**，说明队列在积累，需尽快触发拥塞反应（降速）；若 $\dot{T}_q(t)$ 连续**为负**，说明队列在被清空，网络可能有余量，可以适当提高发送速率。
- **搭配一定的"探测"排队**
  基于延迟梯度的控制思路往往要允许一点儿小排队出现，这样才能通过"延迟的变化"来感知带宽是否充分被利用。若始终不产生任何排队，延迟梯度也难以测出带宽上限。

因此，"延迟梯度"的思想允许**在低时延和探测带宽**之间取得平衡：只要监测到正的延迟梯度，就能迅速降低速率，防止队列膨胀；一旦观察到负的延迟梯度或没有梯度，也就意味着可以继续增速或保持速率。

GPT举例解释说明：

> 我们可以通过一个**直观小例子**来理解为什么说"延迟保持不变"并不一定代表"网络没问题"，反而可能说明网络中已经存在"驻留队列"并对实时流造成了较大延迟。
>
> **1. 举个例子**
>
> - **假设：**
>   - 瓶颈链路容量 (C = 1) Mbps；
>   - 队列的最大可容纳长度可对应约 200ms 的排队时延（也就是说，若队列满载相当于 200ms 的数据量）；
>   - 发送方正好以 (1) Mbps 的速率源源不断发送数据。
> - **现象：**
>   如果发送速率 (r(t)) 与链路容量 (C)"刚好持平"，队列里就会维持一个**不满不空**的常量 backlog。举例说，队列中一直存着约 100kbits 的数据，对应的排队时延就是
>
> $$T_q(t) \;=\; \frac{100\text{kbits}}{1\text{Mbps}} \;=\; 0.1\text{s} = 100\text{ms}.$$

这时，随着时间推移，**队列中存量既不增加也不减少**，因此

$$\dot{T}_q(t) = 0.$$

- **为什么这可能是个“大问题”?**
  - 从“绝对值”角度，实际上用户的每个包都要多等 100ms 以上才能走出瓶颈，这对视频会议、语音通话等实时应用来说已经是“相当高”的附加延迟。
  - 可是从“延迟梯度”($\dot{T}_q(t)$) 或者“延迟变化”角度看，这个队列是“稳定”的——它没有继续升高，也没有下降，($\dot{T}_q(t) = 0$)。
  - 如果某个算法**只盯着“是否还在继续变大”**（比如只要梯度为正才判断要降速），那就会“误以为”网络没再恶化，于是维持当前速率不变。可事实上，系统已经处在一个“排队延迟始终为 100ms”且“没有额外变化”的状态，对实时流非常不利，称之为“驻留队列”现象。

---

2. 为什么对实时应用不利

1. **实时要求“低延迟”**
   比如互动视频、语音聊天中，往返时延最好在 150ms 甚至更低，这里光瓶颈排队就损失 100ms，若再加上传播延迟、编解码处理延迟等，其总时延会大幅上升，造成用户感受到“对方说话或视频画面明显延时”的困扰。
2. **单纯看“延迟变化”不足以判断“队列是否过大”**
   如果我们的控制逻辑仅在“延迟继续涨”时才做出反应，那么当延迟已经涨到一个“对实时而言过大但稳定”的水平时，就会被误判为“没有新拥塞”，不做进一步降速，最终“保持大排队”状态。

---

3. 文章为什么还要依赖梯度?

- **需要一点小队列来探测带宽**
  如果发送端一点排队也不产生，通常也难以判断还能否再增加发送速率。所以在**探测到某些正的延迟变化**（梯度 > 0）时，就能尽早知道队列正在变多，可以及时降速；当延迟梯度 < 0（队列在清空）时，则说明可以加大发送速率。
- **如何避免“驻留队列”过大**
  论文后续章节提到，“自适应阈值” 机制等，会对“梯度”或“测量到的排队延迟”设置一个动态门限，防止在竞争激烈或队列较大的场景下陷入“拥塞却不降低速率”。
  - 也就是除了观察“是否还在涨”，GCC 还会通过调节阈值、结合丢包反馈等策略，力求让队列和延迟保持在一个更低水位，而不是停留在较大的“驻留队列”。

---

4. 小结

“延迟不再变大”并不代表“已经很小”；它可能只是维持在一个对实时业务仍过高的水平。这便是“驻留队列”问题：从斜率（梯度）看是零，但绝对排队时延却依然偏高。正因如此，**在实时通信里，不能只依赖‘变化率为零就不调速’的逻辑；还需要结合对排队延迟绝对值的某些控制或自适应判断机制，避免长期处于大队列。**这是文章里提出基于延迟梯度 + 自适应阈值 + （丢包反馈）的综合方案的原因之一。

---

5. 在 GCC 设计中的作用

论文第 3.1 节的描述主要为后续章节做铺垫：它解释了在实时通信的场景下，**如何用“延迟梯度”来辅助判断网络正处于拥塞积累、拥塞消退或闲置状态。**基于这一点，后续就能设计**延迟梯度估计模块**（卡尔曼滤波）、**过度使用检测**以及**自适应阈值**来更好地捕捉网络负载的细微变化，并在发送方做出相应的“码率调节”。

小结

“排队延迟梯度”是 GCC 乃至其他基于延迟的拥塞控制算法的核心指标之一。它比单纯的 RTT 或绝对排队延迟更能**及时**反映队列是“在加速膨胀”还是“在快速消退”，从而帮助系统做出更灵敏、更准确的拥塞判断。对于实时音视频通信，这种灵敏度对于兼顾低延迟和高吞吐率至关重要。

基于延迟的控制器（接收端）

到达时间滤波器（含卡尔曼滤波）

- 在 GCC 中，“到达时间滤波器”是**接收端**进行“单向延迟梯度”估计的关键模块。它本质上借助卡尔曼滤波器，将每次测量到的“到达时间变化”与过去的状态估计相结合，得到一个平滑且更可信的“延迟梯度”值。下面分步骤说明其原理与实现过程。

---

1. 目的：为何要用到达时间滤波器？

1. **延迟测量的不确定性**
   在实际网络中，单向时延会受到各种抖动因素影响（中间路由排队、操作系统调度延迟、时间戳精度等），直接采用“相邻帧的到达时间差 - 发送时间差”可能带来较大噪声，若直接把“噪声颇高的测量值”当做判断依据，很容易导致错误地进行拥塞或非拥塞判断。
2. **需要克服瞬时抖动，捕捉真实趋势**
   如果能够对噪声做滤波/估计，就能更准确地检测到队列是否“处于上升阶段（正梯度）”或“处于下降阶段（负梯度）”。同时也可减少算法因抖动而不停切换“增/减速率”的情况。

基于这些考虑，GCC 引入了卡尔曼滤波器进行到达时间滤波。

---

2. 测量量：单向延迟变化 $d_m(t_i)$

论文中先定义了一个**测量值**：

$$d_m(t_i) \;=\; \big(t_i \;-\; t_{i-1}\big) \;-\; \big(T_i \;-\; T_{i-1}\big),$$

其中：

- t_i 表示第 i 个视频帧（或批数据包）到达接收端的“最后一个包”时刻；
- T_i 表示对应的第 i 个视频帧“开始发送”时刻（通常在发送端打上时间戳）；
- 所以 $(t_i - t_{i-1})$ 是接收端看到的帧间隔，$(T_i - T_{i-1})$ 是发送端的帧间隔；
- **两者之差**就能近似衡量“在这两帧之间，单向排队延迟大致增加或减少了多少”。

无噪声理想情况下，如果网络排队没变动，二者之差为 0；若排队在变多，($t_i - t_{i-1} > T_i - T_{i-1}$)，差值正；反之为负。

---

3. 状态量：延迟梯度 $\overline{m}(t)$

3.1 为什么只保留“单向延迟梯度”这一状态？

- 作者最初也考虑过“容量 C 与排队延迟梯度 $\mu(t)$”两维状态，但在实际应用中，这往往因采样数据不足、帧大小变化不显著而不可观测，导致收敛、稳定性变差。
- 最终简化为**标量状态** $\overline{m}(t)$，表示当前单向延迟梯度本身，去掉对“容量”显式估计，既简化了实现，也够用来判断网络排队走势。

3.2 系统模型

令 $\overline{m}(t_i)$ 是“单向延迟梯度”的真实状态，论文用一个**线性状态空间模型**来描述：

1. **状态方程**

$$\overline{m}(t_{i+1}) \;=\; \overline{m}(t_i) + w(t_i),$$

   - 意思是：假设状态随时间平稳演化，每次只在原基础上增加一个小随机扰动 $w(t_i)$ 。
   - $w(t_i)$ 代表对排队梯度的过程噪声，其均值为 0，方差记为 (Q)。

2. **输出方程**

$$d_m(t_i) \;=\; \overline{m}(t_i) \;+\; n(t_i),$$

   - 测量值 $d_m(t_i)$ 等于“真实梯度 $\overline{m}(t_i)$”加上测量噪声 $n(t_i)$。
   - $n(t_i)$ 代表各种抖动、时间戳误差等，方差记为 σ_n^2。

由于这是**标量**状态空间模型，卡尔曼滤波方程会非常简单直观。

---

4. 卡尔曼滤波器的执行流程

在每个帧到达时刻 t_i ，滤波器会进行一次“状态更新”。下面是卡尔曼滤波的核心表达式（在标量场景更容易理解）：

1. **预测 & 先验协方差**

$$\overline{m}(t_i)^{(\mathrm{pred})} \;=\; \overline{m}(t_{i-1}) + 0, \quad P(t_i)^{(\mathrm{pred})} \;=\; P(t_{i-1}) + Q.$$

   - 因为状态方程是 $\overline{m}(t_i) = \overline{m}(t_{i-1}) + w(t_{i-1})$，均值增加 0，协方差增加 Q 。

2. **计算卡尔曼增益**

$$K(t_i) \;=\; \frac{P(t_i)^{(\mathrm{pred})}}{P(t_i)^{(\mathrm{pred})} + \sigma_n^2}.$$

   - 当测量噪声 σ_n^2 越大时，$K(t_i)$ 越小，更信任“过去的状态预测”；

- 反之当 σ_n^2 小（测量很准），$K(t_i)$ 较大，更信任“最新测量”。

3. **更新状态**

$$\overline{m}(t_i) \;=\; \overline{m}(t_i)^{(\mathrm{pred})} + K(t_i)\big(d_m(t_i) - \overline{m}(t_i)^{(\mathrm{pred})}\big)$$

- 这可以被看作是一个“加权平均”：

$$\overline{m}(t_i) \;=\; (1 - K(t_i))\,\overline{m}(t_{i-1}) \;+\; K(t_i)\,d_m(t_i).$$

4. **更新协方差**

$$P(t_i) \;=\; \big(1 - K(t_i)\big)\,P(t_i)^{(\mathrm{pred})}.$$

执行完这几步，就得到在时刻 t_i 上对单向延迟梯度的最好估计 $\overline{m}(t_i)$。然后下一帧到来时重复以上过程。

---

5. 参数调整：Q、σ_n^2及初始条件

- **过程噪声方差 Q**
  - 论文中指出，若 Q 过小，滤波器对输入的突变不够敏感，容易导致反应过慢、排队累计过多；
  - 若 Q 过大，则对抖动过于敏感，导致“过度”发出“拥塞”信号，容易错过带宽机会。
  - 实验表明，在他们测试平台上，$Q = 10^{-3}$ 效果较佳。
- **测量噪声方差 σ_n^2**
  - 实际不可事先精确已知，因此文章提到用“残差”做自适应估计：

$$\hat{\sigma}_n^2(t_i) = \beta\,\hat{\sigma}_n^2(t_{i-1}) + (1-\beta)\,\big(d_m(t_i) - \overline{m}(t_{i-1})\big)^2,$$

    其中 $\beta = 0.95$。
  - 这样可以随网络状况及时间戳精度波动进行在线调整。
- **初始状态**
  - 在第一次滤波时，可令 $\overline{m}(t_0) = 0$，并给 $P(0)$ 一个相对大的值（如 0.1），这样滤波器会尽快收敛到一个合理水平。

---

6. 滤波器输出：$m(t_i)$

经过卡尔曼滤波之后的“估计延迟梯度” $m(t_i)$ 就成为后续“过度使用检测器”比较阈值的核心依据——如果 $m(t_i)$ 呈现明显正值并超过阈值 $\gamma(t_i)$ 一定时间，则认为网络进入“过度使用”；若 $m(t_i)$ 呈现明显负值，也会触发“不足使用”的信号。

---

7. 小结

在 GCC 中：

1. **测量 ⇒ 滤波 ⇒ 输出梯度**
   - 先用到达间隔差减发送间隔差获得原始测量 $d_m(t_i)$；
   - 然后通过卡尔曼滤波，去除随机抖动噪声，得到**平滑**的延迟梯度估计 $m(t_i)$。
2. **在“基于延迟”控制器的作用**
   - 该平滑后的延迟梯度是“过度使用检测器”判断网络状态（正常/过载/闲置）的基础；
   - 结合“自适应阈值”，可做出更鲁棒的带宽调节信号，最终以 REMB 形式反馈给发送端。

通过这些步骤，“到达时间滤波器”能让 GCC **敏锐地捕捉队列快速累积或清空的动态**，又不至于被瞬时测量噪声所干扰，算是 GCC 在接收端测量拥塞信息时的“神经中枢”。

- **GPT举例解释：**

> 下面给出一个**简化、通俗的数值示例**，帮助你从“具体数字”层面理解到达时间滤波器（尤其是卡尔曼滤波）是如何工作的，以及它为什么可以平滑噪声、得出更可信的单向延迟梯度估计值。
>
> ---
>
> ### 示例场景设定
>
> - 我们正在接收一段视频流，每一帧（frame）到达后都会记录“接收时间差”和“发送时间差”的差值 $d_m(t_i)$。
> - 设定：初始时刻（帧 0）我们把延迟梯度估计 $m(t_0)$ 设为 0，并给它一个较大的不确定度（方差）(P(0) = 0.1)。
> - 假定过程噪声方差 $Q = 10^{-3}$，测量噪声方差假设暂时为常数 $\sigma_n^2 = 10^{-4}$（仅为例子，实际上可以自适应估计）。
> - 接下来我们每来一帧，就会测一个 $d_m(t_i)$，然后根据卡尔曼滤波更新当前的延迟梯度估计 $m(t_i)$。
>
> ---

第一步：帧 1 到达时

1. **实际测量：** 假设我们观察到

$$d_m(t_1) \;=\; +0.002 \text{ (单位：秒)}$$

表示此帧与上一帧相比，到达时间间隔多了 2ms（去除发送间隔影响后）。我们猜测这是队列稍有增长的迹象。

2. **预测：**
   - 先前状态 $m(t_0) = 0$ 。
   - 卡尔曼滤波预测：

$$m(t_1)^{(\text{pred})} = m(t_0) = 0, \quad P(t_1)^{(\text{pred})} = P(t_0) + Q = 0.1 + 0.001 = 0.101.$$

3. **计算卡尔曼增益 (K(t_1))**

$$K(t_1) = \frac{P(t_1)^{(\text{pred})}}{P(t_1)^{(\text{pred})} \;+\; \sigma_n^2} = \frac{0.101}{0.101 + 0.0001} \approx 0.9990.$$

   - 因为此时我们对“测量噪声”假设很小(仅 10^{-4})，而先前协方差很大(0.101)，滤波器倾向“非常信任”新的测量结果。
   - 获得的增益接近 1。

4. **更新状态：**

$$m(t_1) \;=\; m(t_1)^{(\text{pred})} \;+\; K(t_1)\,\big(d_m(t_1) - m(t_1)^{(\text{pred})}\big).$$

带入数值：

$$m(t_1) = 0 + 0.9990 \times (0.002 - 0) \approx 0.001998.$$

（也就是约 2ms）

5. **更新协方差：**

$$P(t_1) = (1 - K(t_1)) \times P(t_1)^{(\text{pred})} = (1 - 0.9990) \times 0.101 \approx 1.0 \times 10^{-4}.$$

解释：

- 由于初始对状态的“不确定度”比较大，而测量值看起来很“可信”，滤波器就几乎完全采纳了本次测量值——因此 $m(t_1) \approx 0.002$。
- 现在我们的估计告诉我们：“当前延迟梯度是 +2ms/帧间隔”——意味着队列有些增长趋势。

第二步：帧 2 到达时

1. **新的测量：** 假设本帧我们得到

$$d_m(t_2) = -0.0005 \text{ (秒)}$$

说明这次观测到，帧与帧之间接收时间差竟然比发送差略小了 0.5ms，暗示可能队列在减少。（也可能是某种抖动）

2. **预测：**

$$m(t_2)^{(\text{pred})} = m(t_1) = 0.002, \quad P(t_2)^{(\text{pred})} = P(t_1) + Q = 1.0 \times 10^{-4} + 10^{-3} = 1.1 \times 10^{-3}.$$

3. **卡尔曼增益：**

$$K(t_2) = \frac{1.1 \times 10^{-3}}{1.1 \times 10^{-3} + 1.0 \times 10^{-4}} \approx 0.9167.$$

   - 这时增益还是比较高，但没那么接近 1 了，因为不确定度已明显下降，相对测量噪声不那么极端大。

4. **更新状态：**

$$m(t_2) = 0.002 + 0.9167 \times (-0.0005 - 0.002) = 0.002 + 0.9167 \times (-0.0025) = 0.002 - 0.00229 \approx -0.00029.$$

结果是一个略微负的小值。

5. **更新协方差：**

$$P(t_2) = (1 - 0.9167) \;\times\; 1.1 \times 10^{-3} \approx 9.17 \times 10^{-5}.$$

解释：

- 当前测量与上一个状态差距较大（前一次说“增加 2ms”，这次说“减少 0.5ms”），滤波器的结果“折中”了二者：先前预测 0.002，而测量比它低得多，于是新的估计转而变为 -0.00029。
- 这说明滤波器认为：“也许刚才的 +2ms 是真实的，但现在这个 -0.5ms 也不完全是假象，队列可能确实在变小，但可能没测量显示的那么剧烈。”
- 最终结果是一个小负值，代表“延迟梯度稍微偏向下降”。

第三步：帧 3 到达时（示例结束）

1. **新的测量：** 假设这次

$$d_m(t_3) = +0.001.$$

2. **预测：**

$$m(t_3)^{(\text{pred})} = -0.00029, \quad P(t_3)^{(\text{pred})} = 9.17 \times 10^{-5} + 10^{-3} = 0.001092.$$

3. **卡尔曼增益：**

$$K(t_3) = \frac{0.001092}{0.001092 + 0.0001} \approx 0.916.$$

4. **更新状态：**

$$m(t_3) = -0.00029 + 0.916 \times \big(0.001 - (-0.00029)\big) = -0.00029 + 0.916 \times (0.00129) \approx -0.00029 + 0.00118 = +0.0008$$

5. **更新协方差：**

$$P(t_3) \approx (1 - 0.916) \times 0.001092 = 0.0000917.$$

最终，滤波器给出的状态是 +0.00089 秒，这表明这一次我们又回到一个小的正延迟梯度，推测队列又在轻微增长中。

结论：通俗理解

- **多次测量，数据互相“平衡”**
  我们通过卡尔曼滤波，将每一个新的“到达时间差”测量与过去的“状态预测”相结合，得到一个**折衷**的值。
- **滤波器会削弱剧烈的测量波动**
  例如在第 2 步时，测量突然从 +0.002 跳到 -0.0005，但滤波结果只到 -0.00029，没有完全跟测量走，这就是“滤波”去噪的功效。
- **不确定度逐步收敛**
  初始我们对状态“不确定”很大（$P = 0.1$），但每次结合测量后，P 变得更小，对下一次预测更自信，也就更平稳。
- **服务于后续拥塞检测**
  这个平滑的 $m(t_i)$ 输出再和自适应阈值比较，如果多次出现较大的正值，就能触发“过度使用”警告；若值持续为负，表示队列在清空，则可以提示“可以增加速率”。

小结

这个数值例子说明了“到达时间滤波器”如何把每一帧的测量(容易受抖动干扰)与之前的延迟梯度估计做平滑融合，从而既不过分依赖最新的噪声测量，也不会一味固执于旧估计。最终给出一个相对平稳、可信的“单向延迟梯度”序列，帮助 GCC 做出更准确的拥塞判断和速率调整。

- **GPT解释单向延迟梯度与排队延迟梯度的概念异同：**

在这篇论文里，“延迟梯度”其实可以从**理论定义**与**实际测量实现**两个角度去理解，二者在本质上指向同一个物理含义——即队列排队延迟随时间的变化趋势，但它们在表述方式和实现形式上有所不同。下面分别阐述两者的关系与差异：

---

1. 理论层面的“排队延迟梯度” (3.1 节)

- **定义**
  论文 3.1 节中给出的排队延迟梯度 $\dot{T}_q(t)$ 来自对连续模型的分析：设瓶颈带宽为 C，队列长度为 $q(t)$，排队延迟为

$$T_q(t) = \frac{q(t)}{C}.$$

  那么

$$\dot{T}_q(t) = \frac{d\,T_q(t)}{dt} = \frac{d\,q(t)}{dt} \times \frac{1}{C} = \frac{\dot{q}(t)}{C}.$$

  这描述了**网络排队时延随时间的瞬时“变化率”**。若 $\dot{T}_q(t) > 0$，说明队列在积累；若 < 0，则队列在消退；=0 则队列不再变化（可能是空、满或驻留队列）。

- **侧重点**
  - 这是一个**连续时间的理论概念**，在模型中用导数 $\dot{q}(t)$ 表达队列增长或减少的速率。
  - 强调“若队列一直在变多或变少”，即 $\dot{T}_q(t) \neq 0$，则需要触发相应的拥塞反应；而如果排队延迟保持固定值（$\dot{T}_q(t) = 0$），则说明要么网络利用率不足，要么已经出现稳定的大排队。

---

2. 实际测量层面的“单向延迟梯度”（到达时间滤波器）

- **测量做法**
  在“到达时间滤波器”中，论文采用

$$d_m(t_i) = \big(t_i - t_{i-1}\big) \;-\; \big(T_i - T_{i-1}\big),$$

  来获取**相邻帧之间的到达时间差**与**发送时间差**之差值，这就是所谓的“单向延迟变化 (delay difference)”。当把这一离散测量值再经过卡尔曼滤波，即得到平滑的“单向延迟梯度”估计 $m(t_i)$。
- **侧重点**
  - 这是**离散时间、以帧（或批数据包）为间隔**的测量方法，用于在接收端实际获取“排队延迟的增减”迹象。
  - 将 $(t_i - t_{i-1}) - (T_i - T_{i-1})$ 视作“上一次到这一次之间，网络单向时延是否增加、增加了多少毫秒”，相当于对连续 $\dot{T}_q(t)$ 进行了一种**离散近似**。
  - 通过卡尔曼滤波修正噪声后，就能得到一个相对平滑的“单向延迟梯度”序列，用于后续过度使用检测。

---

3. 二者之间的关系与差异

1. **同一个物理本质**
   - 不论“排队延迟梯度”($\dot{T}_q(t)$) 还是“单向延迟梯度测量”($(t_i - t_{i-1}) - (T_i - T_{i-1})$)，核心都在反映**队列延迟是否在上升或下降**，也就是“延迟的变化率”。
   - 从概念上说，单向延迟变化 $d_m(t_i)$ 就是 $\dot{T}_q(t)$ 在离散帧级的一种度量。
2. **理论模型 vs. 现实测量**
   - 论文 3.1 节的 $\dot{T}_q(t)$ 是一个**连续时间**且基于队列长度 $\dot{q}(t)$ 的抽象模型，用以阐述为什么要基于“延迟的变化率”而非绝对值去判断拥塞。
   - 到达时间滤波器里的 $d_m(t_i)$ 则是**实际实现**时，每个视频帧或数据包到达后的“离散测量”，通过发送端与接收端时间戳差值计算得来。
3. **滤波/估计 vs. 理想导数**
   - 理论上，$\dot{T}_q(t)$ 直接用“队列长度变化率/带宽”即可得到；可实际中我们**无法**直接得知链路带宽 C 或队列长度 $q(t)$，并且还要面对噪声、抖动、非理想时间戳等问题。
   - 因此论文在“到达时间滤波器”部分使用**卡尔曼滤波**等方法，将离散测量 $d_m(t_i)$ 变为平滑可靠的估计 $m(t_i)$，再去做过度使用检测。

---

4. 小结

- **在原理上**：二者都表达了**网络单向延迟随时间的变化（排队上涨或下降）的速率**。可以把**“单向延迟梯度”**视作论文 3.1 节“排队延迟梯度”在实际实现中的离散化版本与测量办法，二者是**同一个物理概念**的不同表述。
- **在实践中**：前者（3.1 节的 $\dot{T}_q(t)$) 是对队列模型的连续导数分析，帮助作者解释为什么要看“延迟变化率”而非“绝对延迟”；后者（到达时间滤波器中的 $d_m(t_i)$) 则是把这种概念**落地到可测量的帧间到达/发送时间差**，并用滤波器来处理噪声，以做实时的拥塞检测。

自适应阈值块

- 在 GCC 的“基于延迟的控制器”里，除了要准确测量和估计单向延迟梯度 $m(t)$ 以外，还需要决定“多大程度的延迟变化算拥塞”。**自适应阈值块（Adaptive Threshold Block）** 正是解决这个问题的关键：它生成一个随时间动态变化的阈值 $\gamma(t)$，用于在“过度使用检测器（Overuse Detector）”里与估计的延迟梯度作比较，从而判断当前网络状态。下面我们分步骤讲解其原理与实现思路。

---

1. 为什么需要“自适应”而不是固定阈值?

1) 小瓶颈队列场景

- 有些网络链路的队列规模（缓冲区大小）相对很小，或者开启了小 Buffer、AQM 等。此时排队延迟绝对值可能无法变得很大。
- 如果我们事先设了一个比较大的静态阈值 $\bar{\gamma}$，那么延迟梯度 $|m(t)|$ 即便“真在上升”，数值也达不到 $\bar{\gamma}$，就会**一直无法判定为过度使用**；结果导致队列依然可能累积不小的延迟，却因“达不到阈值”而被忽视。

2) 与并发 TCP（或其他大流）竞争

- 若网络中另外存在大量 TCP 流，这些流会不时地突发性使得排队时延出现较大波动（尤其是 TCP 拥塞窗口加速膨胀阶段）。
- 如果阈值设得过小，那么每一次这种并发流导致的排队波动都会使 GCC 误判"过度使用"，频繁降速，结果就是**GCC 流被饿死**，丢失带宽份额。
- 在这种情况下，我们希望阈值能够**适度提高**，容忍一定幅度的延迟梯度波动，避免过于保守地判定拥塞。

由此可见，**固定的阈值无法兼顾各种网络缓冲规模、并发流竞争模式**等多变场景，需要一个能"上下浮动"的动态阈值 $\gamma(t)$。

---

2. 公式：自适应阈值的更新

文中给出的自适应阈值设计如下（见第 4.2 节）：

$$\gamma(t_i) = \gamma(t_{i-1}) \; + \; \Delta T \cdot k_\gamma(t_i) \left(|m(t_i)| \; - \; \gamma(t_{i-1}) \right),$$

其中：

- t_i 表示第 i 个视频帧（或测量样本）到达接收端的时刻；
- $\Delta T = t_i - t_{i-1}$ 为相邻帧的时间间隔；
- $m(t_i)$ 是"到达时间滤波器"输出的单向延迟梯度估计；
- $|m(t_i)|$ 表示它的绝对值（关心"幅度"而不是正负方向）；
- $\gamma(t_{i-1})$ 为上一次的阈值；
- $k_\gamma(t_i)$ 是一个增益系数，决定了阈值调节速度。

2.1 增益系数 $k_\gamma(t_i)$

作者给出了：

$$k_\gamma(t_i) = \begin{cases} k_d, & \text{if } |m(t_i)| < \gamma(t_{i-1}), \\ k_u, & \text{otherwise.} \end{cases}$$

也就是说：

1. 当测量到的梯度 $|m(t_i)|$ 还**没到**当前阈值 $\gamma(t_{i-1})$ 时，**减少**阈值的速度用 k_d，让它更快往下靠近 $|m(t_i)|$。这样能使阈值**不要一直过高。**
2. 当 $|m(t_i)|$ **超过** $\gamma(t_{i-1})$ 时，阈值用 k_u（通常比 k_d 大）**相对更慢**地增加。这样能让阈值上升到一个稍高水平，但不会一下子涨很多，从而不会马上抑制过度使用检测。

> - **结论**：$\gamma(t)$ 相当于对 $|m(t)|$ 的一个"低通或平滑"跟踪。只是在 $|m(t)|> \gamma$ 时涨得比较慢，$|m(t)| < \gamma$ 时降得比较快。

---

3. 直观理解：避免两个极端问题

3.1 当瓶颈队列小，$|m(t)|$不大时

- 如果阈值是固定且大，$|m(t)|$ 永远无法超越阈值 → 检测不到过度使用 → 队列无故被允许继续膨胀一段时间，导致延迟可能偏大。
- 自适应阈值 γ 在看到 $|m(t)|$ 一直小于自己后，就会**以 k_d 速度下调**，逐步趋近当前的 $|m(t)|$，从而在后续稍有进一步增长时就能触发"过度使用"信号并降速。

3.2 当和并发 TCP 竞争，$|m(t)|$频繁大起大落时

- 如果阈值设得很小，则每次并发流带来的大幅度延迟波动都会导致 GCC 误判"自己"过度使用，不断降速，最终**被饿死。**
- 有了自适应阈值，初期当 $|m(t)|$ 多次超越当前阈值，γ 会慢速地上调。虽然短时间会触发几次"过度使用"，但随着阈值逐渐变大，后续就不再为普通幅度的波动而过于敏感，从而保留了带宽分享能力（不会一味地大幅降速）。

---

4. 参数 k_u, k_d 的选择

论文第 4.2 节还提到作者通过一系列实验（含可变链路容量、多路 GCC 流、GCC 与 TCP 流竞争等）进行调参，用一个目标函数（包含吞吐率和时延的综合衡量）选出比较优的组合。

- 其中他们的结论是：

$$(k_u, k_d) = (0.01,\ 0.00018),$$

大体保证了阈值在 $|m(t)|$之上时"上升缓慢"、在 $|m(t)|$之下时"下降相对快"，既不至于过于迟钝，也不会频繁触发过度使用。

---

5. 与"过度使用检测器"如何交互?

1. **过度使用检测器**
   - 在每帧到达时，比较 $m(t_i)$ 与 $\gamma(t_i)$。如果 $m(t_i)$ 持续大幅超过 $\gamma(t_i)$ 并维持一定时间（论文中是约 100ms），则判定为 "Overuse"（过度使用）。
   - 如果 $m(t_i)$ 落在区间 $-\gamma(t_i)$ 到 $+\gamma(t_i)$之间，则判定为"Normal"。
   - 过低（小于 $-\gamma(t_i)$）则表示 "Underuse"（不足使用）。

2. **自适应阈值的好处**
    - 每次做完比较后，阈值会在下一帧根据公式动态更新，使检测器能在不同网络条件下都有合适的“触发灵敏度”。
    - 避免了固定阈值造成的“要么过敏，要么迟钝”两种极端现象。

---

6. 直观举例

- **情形 A**：排队增长缓慢，且 $|m(t)|$ 远小于当前 γ。
    - 因为 $|m(t)| < \gamma$，更新时会使用 k_d 做“快速下降”，γ 会向 $|m(t)|$ 靠拢。几次迭代后，γ 变小；当下次排队有轻微抖动，就会超越 γ 并触发 Overuse。
- **情形 B**：跟一个 TCP 流共享，队列起伏很大，初期 $|m(t)|$ 常常远大于 γ。
    - 此时阈值用 k_u 较小的增量慢慢往上调，虽然初期或许会多次判定 Overuse，但当阈值上去后，就不会再一惊一乍地判断拥塞，GCC 也因此不会被频繁降得过低，从而保持一定的带宽竞争力。

---

7. 小结

自适应阈值块是“基于延迟”拥塞检测的关键补充——它让系统能够**动态调节对延迟梯度的敏感度**。配合“到达时间滤波器”输出的 $m(t)$，以及“过度使用检测器”的判断逻辑，共同保证：

1. 在小队列、较干净链路场景下，也能较早地发现排队增长；
2. 在有大流并发或大抖动的场景下，不会过度降速而“饿死”；
3. 兼顾维持较低排队时延、相对高带宽利用率、并发公平性等目标。

这正是 GCC 能在不同网络条件下都取得较好平衡的核心设计之一。

过度使用检测器

- 在 GCC 的接收端“基于延迟控制器”中，**过度使用检测器（Overuse Detector）**负责**将“延迟梯度估计”与“自适应阈值”相比较**，做出“网络是否过度使用”的最终判断。下面详细介绍它的原理与实现。

---

1. 过度使用检测器的整体定位

- 在“到达时间滤波器”中，我们已经用卡尔曼滤波等方法得到一个相对平滑、可信的单向延迟梯度估计 $m(t_i)$。
- 在“自适应阈值块”中，我们根据网络状况为每个时刻生成一个阈值 $\gamma(t_i)$。
- **过度使用检测器**就是用来比较这两个值，并根据比较结果输出一个“信号”给“远程速率控制器（Remote Rate Controller）”的有限状态机，决定是增加、保持还是减少发送端码率。

---

2. 过度使用检测的触发条件

根据论文图 3、图 4（以及相关文字）描述，过度使用检测器在每帧到达时执行以下逻辑：

1. **比对延迟梯度与阈值**
    - 若 $m(t_i) > \gamma(t_i)$，则我们可能处于“过度使用”状态；
    - 若 $m(t_i) < -\gamma(t_i)$，则我们可能处于“不足使用”状态；
    - 若 $-\gamma(t_i) \le m(t_i) \le \gamma(t_i)$，则为“正常”状态。
2. **连续判定时间**
    - 如果检测到 $m(t_i) > \gamma(t_i)$ 并保持一段时间（例如论文提到约 100ms）都保持在阈值之上，则最终判定**过度使用**；
    - 若只是瞬时地冲破阈值，但很快又掉下去，可能是噪声或短暂抖动，不必马上宣布过度使用；
    - 同理，对于**不足使用**也要看是否持续满足 $m(t_i) < -\gamma(t_i)$ 。
3. **输出信号 (s)**
    - **Overuse** 信号：表示队列正在持续膨胀，需要让发送端降速。
    - **Normal** 信号：表示队列变化不大，可以保持当前速率或继续观察。
    - **Underuse** 信号：表示队列在减少，可提示发送端增大速率。

---

3. 为什么需要一个“持续时间判定”？

- **减少错误触发**
  网络抖动和测量噪声都可能使得 $m(t)$ 一瞬间超过 γ，若立刻宣布拥塞，系统就会频繁减速，又很快检测到不足使用再加速，造成“速率振荡”。

- **滤除短暂尖刺**
  要让 $m(t)$ 在阈值之上维持一个最小时间窗，如 100ms，才能判定过度使用，确保队列确实处于真实的持续膨胀而非瞬时波动。

这在论文图 3 中有示意：$m(t)$ 超过阈值后，需要看其超过阈值持续的时长，如果达到了指定阈限（如 100ms），才触发过度使用事件。

---

4. 与远程速率控制器的交互

4.1 有限状态机 (FSM)

- 论文图 4 给出了一个名为“远程速率控制器 (Remote Rate Controller)”的有限状态机，它的状态包括**增加、减少、保持**三种。
- **过度使用检测器**每次生成的信号 (s)（Overuse / Normal / Underuse）会驱动该 FSM 切换状态，从而更新“接收端计算的推荐码率” $A_r(t_i)$。
  - Overuse → 状态机进入或保持“减少”状态；
  - Underuse → 状态机进入或保持“保持”状态（也可能转移到下一步的“增加”，视实现细节）；
  - Normal → 状态机进入或保持“增加”状态。

4.2 调用示例

当过度使用检测器输出 Overuse，远程速率控制器就会把 A_r 乘以一个因子（如 0.85）或直接设置成 $\alpha \cdot R_r(t)$ 来减小码率；Underuse 时则保持、Normal 时采用略大于 1 的因子（如 1.05）增大码率。最后把这个新的速率通过 REMB 等 RTCP 扩展字段反馈给发送端。

---

5. 小例子说明

1. **初始情况**：假设网络暂时空闲，$m(t)$ 在小范围内抖动，$|m(t)| < \gamma(t)$，过度使用检测器输出 Normal → 远程速率控制器状态为“增加”，A_r 逐步增大。
2. **队列堆积**：当发送端码率逼近瓶颈容量，队列开始积累，$m(t)$ 连续地 > $\gamma(t)$ 一定时间（>100ms），触发 Overuse → 远程速率控制器切到“减少”状态 → A_r 降低 → 发送端码率相应收缩。
3. **排队恢复**：随着发送端码率收缩，队列排空，$m(t)$ 将变为负值或非常小，一旦检测到 $m(t) < -\gamma(t)$，触发 Underuse → 状态机暂时保持或很快转到“增加”→ 再次提高码率。

---

6. 与自适应阈值块配合的重要性

- 如果没有“自适应阈值”，当并发流或小队列场景发生时，过度使用检测器可能过度频繁或过度迟钝地触发 Overuse。
- 自适应阈值块让 $\gamma(t)$“跟随” $|m(t)|$ 的变动并在不同场景中调整灵敏度，从而让过度使用检测器在各种网络状况下保持合适的敏感度（既不遗漏真正拥塞，也不过度降速）。

---

7. 小结

1. **过度使用检测器的核心使命**：对滤波后的延迟梯度 $m(t)$ 与动态阈值 $\gamma(t)$ 做比较，输出 Overuse / Underuse / Normal 三种信号。
2. **持续时间判定**：通过设定诸如“须连续超阈值 100ms”之类的机制，过滤掉瞬时误报。
3. **与远程速率控制器互动**：最终通过状态机来调节接收端给发送端的“带宽建议”，形成闭环拥塞控制。
4. **搭配自适应阈值**：确保在各种队列规模、并发流冲击下都能适度做出“过度使用”或“不足使用”的准确判定。

这使得 GCC 的“基于延迟”检测方案既能快速响应真正的队列膨胀，也不会被瞬时噪声或外部流量波动过度触发，从而在实践中取得较稳定、低时延的效果。

远程速率控制器 FSM

- 在 GCC 的接收端，“远程速率控制器（Remote Rate Controller）”用一个有限状态机 (FSM) 来根据“过度使用检测器”的输出信号，决定给发送端反馈的目标速率 A_r。其核心思路是：**一旦检测到网络排队上升，就降低速率；当检测到网络闲置或正常时，则逐步增大速率。**下面分步骤介绍。

---

1. FSM 的三个主要状态

论文图 4 显示了远程速率控制器的有限状态机 (Finite State Machine, FSM)，它包含以下三种状态：

1. **Increase（增加状态）**
   - 处于此状态时，控制器会让推荐速率 A_r 以适度的"指数"方式增长（例如乘以 1.05），用于持续探测更高的可用带宽。
   - 一般在网络"没有显著排队膨胀"且也不空闲时，会进入或保持此状态。
2. **Decrease（减少状态）**
   - 当检测到明显的排队上升（过度使用）后，系统会将推荐速率 A_r 乘以一个缩减因子（如 0.85），或设为 $\alpha \cdot R_r(t)$ 来快速降低码率。
   - 目的是缓解瓶颈队列继续膨胀，减少高延迟和丢包风险。
3. **Hold（保持状态）**
   - 如果检测到网络处于"不足使用（Underuse）"或队列正在变空，就会进入或保持"Hold"，不再增加速率，先观察是否真正处于空闲或还有余量。
   - 该状态可避免频繁地在"加速/减速"之间来回振荡。

---

2. 状态切换逻辑：与"过度使用检测器"信号的对应

过度使用检测器会输出以下三种信号 s：

- **Overuse**（过度使用）
- **Underuse**（不足使用）
- **Normal**（正常）

FSM 根据检测器输出的信号，做对应的转移：

1. **Overuse** → 切换到 **Decrease**
   - 表示队列在积累，延迟梯度超过阈值并持续一段时间，故要赶紧降速防止大量排队。
2. **Underuse** → 切换到 **Hold**
   - 表示队列在下降或网络容量尚有余量，但不会马上"飙升"到 Increase；先保持观察，避免抖动。
3. **Normal** → 切换到 **Increase**
   - 表示队列变化不显著，即网络未拥塞也未空闲过度，此时可尝试小幅度或指数方式增加发送速率。

在这三种主状态之上，也有可能存在更细的过渡机制，比如从 **Hold** 状态再转到 **Increase** 等，具体取决于实现细节。不过论文的主要结构即这三大状态 + 三种信号的交互。

---

3. 更新速率 $A_r(t_i)$ 的公式

当 FSM 进入不同状态时，控制器会用不同的规则计算新的推荐码率 $A_r(t_i)$。论文中给出的核心逻辑（式 (3)）可以概括为：

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

- 其中 $\eta \approx 1.05$ 表示"每次 Increase 都比上一时刻增加约 5%"；
- $\alpha \approx 0.85$ 表示"每次 Decrease 会将速率缩减为 85% × 接收速率"；
- $R_r(t_i)$ 是过去一小段时间（如 500ms）测得的接收带宽；
- 如果状态是 Hold，则保持原速率。

> 注：文中有时用"$\alpha R_r(t_i)$"替代"$0.85 \times R_r(t_i)$"一类，这种写法是等价的。

---

4. 上限限制：1.5 × (R_r(t_i))

论文中还提到，$A_r(t_i)$ 会被限制为不超过当前接收带宽的 1.5 倍，这样能防止**在没有可靠依据时，速率瞬间增大过多**，导致突然挤爆网络。

- 即

$$A_r(t_i) \;\le\; 1.5\, R_r(t_i).$$

---

5. 与 REMB 报文交互

- 当远程速率控制器计算出新的 $A_r(t_i)$ 后，就会决定是否立即发送一个 RTCP-REMB（Receiver Estimated Maximum Bitrate）消息给发送端，或定期每秒发一次。
- 发送端在收到更新后，就会相应调整视频编码器码率或 Pacer 的出包速率；形成一个端到端的闭环控制。

---

6. 小例子场景

1. **初始阶段**
   - 网络比较空闲，检测器输出 Normal → FSM 处于 Increase；远程速率控制器把 A_r 从较低值慢慢乘以 1.05 提升，探测更高带宽。
2. **逼近瓶颈**
   - 当速率接近或超过可用带宽，排队膨胀 → 过度使用检测器输出 Overuse → FSM 切到 Decrease；控制器把 A_r 设为 0.85 × 当前测量带宽，显著减速。
3. **排队清空**
   - 随着减速后，队列得以排空 → 过度使用检测器发现变为负的延迟梯度 → Underuse → FSM 切到 Hold（或先 Hold 再切回 Increase），保持速率等待下一步观察。
4. **再次增加**
   - 若后续检测器输出 Normal 又回来了 → FSM 又进 Increase，轻度加速。

这样周而复始地在 Increase / Decrease / Hold 三态间切换，并在延迟和可用带宽间找到平衡点。

---

7. 总结

- **远程速率控制器 FSM** 是 GCC 接收端最终的"码率决策引擎"：
  1. 接收来自"过度使用检测器"的 Overuse/Normal/Underuse 信号；
  2. 决定进入 "Increase/Decrease/Hold" 三种状态之一；
  3. 相应地计算并更新推荐速率 A_r；
  4. 通过 RTCP （REMB 消息）告知发送端。
- 该设计使 GCC 能够**在发现拥塞时迅速减速**、在网络有余量时"适度地、渐进地"增速，并通过 Hold 状态避免频繁震荡。

这就是"远程速率控制器 FSM"在整个"基于延迟控制器"中所扮演的角色：**实现具体的码率调控动作。**它与到达时间滤波器、自适应阈值、过度使用检测器一起，构成了 GCC 在接收端判断和控制拥塞的完整工作流程。

REMB 处理模块

- 在 GCC 的接收端，"REMB 处理"模块（Receiver Estimated Maximum Bitrate Processing）承接"远程速率控制器 (Remote Rate Controller)"产生的最新带宽估计 A_r，并决定**何时、如何**通过 RTCP 扩展报文发送这一估计值给发送端，从而形成端到端的闭环。下面分步骤介绍其工作原理与作用。

---

1. 什么是 REMB?

- **概念**
  REMB（Receiver Estimated Maximum Bitrate）是一个 RTCP 扩展报文格式，全称为 *"Receiver Estimated Maximum Bitrate"*。它并非在原始 RFC 3550（RTP/RTCP 标准）中定义，而是作为后续的补充或扩展提案。
- **用途**
  REMB 报文用来让接收端在 UDP/RTP 场景下，告诉发送端"基于我这边观测到的情况，你最好把视频的发送码率限制在某个数值以下"。

在 GCC 中，接收端实际上就是持续更新这个"最大码率"估计 $A_r(t)$，并通过 REMB 告诉发送端。

---

2. REMB 处理模块的职责

1. **判断何时发送 REMB**
   - 论文中提到，默认情况下，REMB 报文每隔 1 秒发送一次；
   - 但如果 $A_r(t)$ 降得很快（比如比上一时刻下降超过 3%），则会**立刻**发送一次 REMB，确保发送端能够尽快降码率，避免更多排队或丢包。
2. **生成 REMB 包的格式**
   - 在 RTCP 协议头里，需要写入接收端估算的最大码率值（即 $A_r(t)$）。
   - 也可包含其他信息，如"所针对的 SSRC 列表"等。
3. **与其他 RTCP 信息配合**

- 同时可以和丢包统计（packet loss fraction）、接收速率等信息放在同一个 RTCP Receiver Report (RR) 中发送，或者单独发扩展报文。
- 最终都是为了让发送端能够在“基于丢包的控制器” + “基于延迟的控制器反馈”这两个维度上进行综合调速。

---

3. 为什么要在“码率快速下降”时立即发送?

- **避免排队延迟或丢包进一步恶化**
  如果检测到网络已经出现“过度使用”并触发 Decrease，通常会把 $A_r(t)$ 降到原先的 85% 或其他更低水平。若要等到默认的下一个“整 1 秒”周期，发送端才能得到新信息，这中间可能还会造成更多的排队甚至丢包。
- **实时性考虑**
  视频会议场景对延迟非常敏感，既然我们已经知道“网络拥塞了”，就应该**立刻**通知发送端尽快减速。

因此，论文设定：当 $A_r(t_i)$ 与 $A_r(t_{i-1})$ 相比下降幅度超过 3%（这个阈值也可在实现中调整）时，立即发送 REMB 更新通知。

---

4. 发送端如何处理收到的 REMB?

- **发送端接到新的 A_r**
  它会在“基于丢包的控制器”计算出的发送速率 $A_s(t)$ 与 $A_r(t)$ 之间取一个最小值——也就是

$$A(t) = \min\big(A_r(t),\, A_s(t)\big).$$

- **更新编码或 Pacer**
  发送端的码率控制引擎（如视频编码器、或发送 Pacer）随后会基于这个新的目标速率调节实际发送比特率，避免占用超过所提示的网络能力。

---

5. 总结：REMB 处理模块在整个 GCC 的位置

1. **接收端层面**
   - 先经过到达时间滤波器、自适应阈值和过度使用检测器判断出网络状态；
   - 远程速率控制器 FSM 计算出新的推荐发送速率 $A_r(t)$；
   - “REMB 处理模块”基于该速率决定何时发 RTCP-REMB 包。
2. **发送端层面**
   - 收到 REMB，即可快速了解接收端实际能承受的最大码率，并据此调整编码、发送节奏。
   - 同时，发送端还会结合丢包率来做一些独立的增减判定，但最终取 A_r 与丢包控制后速率的最小值。

正因为有了 REMB 这条快速反馈通道，GCC 在“检测到拥塞→通知发送端减速”之间的时延可以相对缩短，**提高了对拥塞的反应灵敏度**，从而将排队延迟控制在更低水平。

基于丢包的控制器（发送端）

概览

- 在 GCC 的整体设计中，“基于丢包的控制器”位于发送端，用以辅助或补充接收端基于延迟的控制器所反馈的建议码率。它主要通过**丢包率**这一信号来判断是否出现了网络过载，从而在必要时及时降低发送速率。下面我们分解说明其主要组成部分及工作流程。

---

1. 整体框架：如何接收丢包信息

1. **RTCP 反馈**
   - 发送端会周期性地从接收端获得 RTCP 报文，内含“丢包分数” $f_l(t_k)$。这通常是基于标准 RTP/RTCP 机制中“Fraction Lost”字段计算得到，反映在某一段时间内丢失的包数占比。
   - 此外，接收端还可能通过 “REMB” 等 RTCP 扩展字段告知其估计的最大可用码率 $A_r(t)$（基于延迟控制器的结果）。
2. **发送端的丢包控制**
   - 发送端会读取 RTCP 中的“丢包分数” $f_l(t_k)$，并基于一个简单的阈值策略来决定当前发送速率是否需要大幅降低、轻度探测增加或保持不变。
3. **发送端最终目标速率**
   - 发送端最终采用

$$A(t) \;=\; \min\big(A_r(t),\; A_s(t)\big),$$

其中 $A_r(t)$ 来自接收端的延迟反馈（REM 反馈），$A_s(t)$ 则是基于丢包控制器自身逻辑算出的目标速率。

---

2. 主要模块及逻辑

在论文中（见第 3.4 节、公式 (5)），基于丢包的控制器主要包括以下几个组成或逻辑步骤：

1. **丢包分数测量**
   - 接收端按照 RTP/RTCP 标准在一定统计窗口内计算丢失包数占总发送包数的比例，记为 $f_l(t_k)$。这在 RTCP Sender/Receiver Reports 中常见。
   - 发送端收到 RTCP 时，可以得知过去一段时间丢包状况。
2. **阈值判断 (三个区间)**
   根据丢包分数 f_l 与 0.02、0.1 这两个阈值的比较，发送端采取不同动作：

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

   - 若丢包率 > *10%*，表示网络拥塞严重，需要**成倍地明显减速**，具体是乘以 $(1 - 0.5\, f_l(t_k))$，近似会把带宽压低到当前值的一半或更低；
   - 若丢包率 < *2%*，表示丢包很少，可以尝试**乘以 1.05**（约 5% 的增量）来进一步探测是否能获得更高带宽；
   - 若丢包率介于 2% 与 10% 之间，则保持当前发送速率不变。
3. **发送频度**
   - 论文中提到，每当收到携带丢包分数的 RTCP 报文或 REMB 消息时，就会执行上述计算来更新 $A_s(t_k)$。
   - RTCP 通常以约数百毫秒到 1 秒的频度往返一次，因此丢包控制逻辑也是一个“分段周期性”执行的流程。
4. **与接收端反馈 (A_r) 的联合决策**
   - 一旦算出新的 $A_s(t_k)$，发送端还要将其与“延迟控制器”在远端给出的码率建议 $A_r(t_k)$ 相比较：

$$A(t) \;=\; \min\bigl(A_s(t_k),\, A_r(t_k)\bigr).$$

   - 这样可保证**同时**兼顾“丢包率”信号（避免在丢包很高时依旧暴力发送）和“延迟梯度”信号（防止长队列、保证低延迟）。

---

3. 整体工作流程举例

1. **初始阶段**
   - 丢包分数可能比较低 (<2%)，发送端基于上述公式选择“增速”，不断把 A_s 向上探测。
   - 若同时延迟控制器反馈也比较宽松，则 A_r 没有限制，两者一起使发送速率持续上升。
2. **出现较高丢包**
   - 当丢包率观测值连续几次 >10%，发送端会立刻大幅度地将 A_s 减到一半或更小（因公式中的 0.5×f_l）。
   - 这从丢包角度表明网络已明显拥塞，需要迅速降码率，防止更多丢包。
3. **丢包回落**
   - 随着发送端降速，丢包率回到 <2%，发送端又会每次增速约 5%，探索更高带宽是否可行。
4. **防抖设计**
   - 当丢包率在 2%~10% 之间，发送端保持 A_s 不变，避免频繁在“增 / 减”之间快速切换，导致输出码率震荡。

---

4. 与 Pacer / 编码器的配合

- **目标码率 A(t) 的应用**
  - 一旦计算出新的目标码率后，发送端会将其传给内部的 Pacer（发包速率控制器）和视频编解码器。
  - 编码器尽量生成接近该码率的比特流，以保持视频质量与网络状态平衡。
  - 若编码器一时输出过多数据，Pacer 也会对发包进行排队并尽量按限速发送，以避免发送侧自身形成拥堵。

---

5. 小结

1. **基于丢包的控制器**是对接收端“基于延迟控制器”的补充：
   - 当延迟控制器过于保守或失效（比如队列已满时会产生丢包），丢包信号能**及时**告诉发送端“该降速了”；
   - 同时在丢包率很低时，可适度探测带宽上限，提升视频质量。
2. **三个区间的分段逻辑**（>10% 大幅减速、<2% 小幅增速、否则保持）相当简单但行之有效，与传统 TCP 的“丢包 = 拥塞”思路类似。
3. **最终发送速率**取决于“丢包控制”与“接收端延迟反馈”两条路径中的最小值，从而综合实现：
   - **低延迟**（受延迟梯度驱动）
   - **不大规模丢包**（受丢包率约束）
   - **带宽探测**（增/减策略）
   - **较好兼容其他流**（与 TCP 等流量共存）。

通过这些机制，发送端能够在“实时满足接收端延迟感知”与“丢包优先级”之间取得平衡，进一步提高 WebRTC 视频通话在实际网络中的稳定性与性能。

Pacer（发送端）

- 在 GCC 的整体传输流程中，“Pacer”是发送端的一部分，用来**将实际发送的数据流做时间上的平滑和切分**，从而避免一次性把大量视频包集中发送出去，引起突发队列和延迟。虽然这篇论文（*Analysis and Design of the Google Congestion Control*）并没有专门用大量篇幅介绍 Pacer，但它在第 3.5 节（Sending Rate Drive）和框架图中简要提到 Pacer 的角色。下面我结合该论文中简要的描述与 WebRTC 的常见实现来进行更详细的说明。

---

1. 为什么需要 Pacer?

1. **避免突发发送（burst）**
   - 视频编码器在编码一个帧后，可能会产生成百上千字节甚至几兆字节的数据，并且常常是**帧级别**地一次性喂给发送端。
   - 如果没有 Pacer，就可能在极短时间内把这一整帧的数据包全部送入网络，易导致发送端或中间路由器产生短时队列高峰。
2. **控制发送的“瞬时速率”**
   - 即便算法层面设置了目标码率（比如 1 Mbps），但在毫秒级尺度上，如果一次性发送多个 RTP 包，瞬时速率可能远高于平均目标值，依旧会造成队列抖动和延迟飙升。
3. **平滑输出**
   - Pacer 以更均匀的节奏将数据包排出，使网络“看到”的发送流量更稳定。
   - 这在中小带宽、或移动网络情境下尤其显著地降低了网络抖动和拥塞波动。

---

2. Pacer 的主要功能

1. **按短时间片调度发送**
   - 通常 Pacer 会在一个固定的小间隔（比如 5ms 或 10ms）里，只允许发送一定数量的字节或数据包，以符合设置的目标码率。
   - 若编码器瞬时输出比目标码率能支持的更多包，Pacer 会将它们缓存并分散在后续的几个时间片里发送。
2. **服从上层的目标发送速率**
   - 在 GCC 框架里，最终的目标速率 $A(t) = \min\big(A_r(t),\, A_s(t)\big)$（由基于延迟和基于丢包的控制器共同决定）。Pacer 会根据这个目标来计算“每 5ms、10ms 等时间片能发送多少字节”。
3. **处理多路流量（多 SSRC）**
   - 在实际的 WebRTC 场景中，Pacer 有时需要同时处理音频流、视频流、FEC/RTX 流等多个 SSRC 的数据优先级。它会确保高优先级（如音频）数据先出队，再轮到视频等更大流量数据。
   - 不过这是在更复杂的多流场景下才更明显。

---

3. 论文中提到的“Pacing Factor”

在论文第 3.5 节（Sending Rate Drive）和图 5 中，作者提到：

> 如果编码器产出的比特率暂时高于目标 (A)，Pacer 可以“加快速度”（例如乘以 1.5 的 pacing factor），以便在短期内迅速清空发送端队列，避免发送侧也产生排队延迟。

具体原理是：

1. **短时超速**
   - 当编码器码率偶尔超过了拥塞控制设定的目标值 A 时，Pacer 先把这批多余数据排进队列，但会以“fA”（其中 $f = 1.5$ 等）略高于 A 的速率发送出去。
   - 这样能在相对较短时间内清空发送端队列，保证发送端自己不会有太大延迟。
2. **长时平均仍不超过 (A)**
   - 如果编码器长期超额产生数据，那么 Pacer 队列会持续积压，最终会让平滑后的实际平均发送速率接近 A（因为 Pacer 不会无限加速）。
   - 仅在峰值时刻给出一点弹性，但整体上仍保持在目标之内。

4. 具体的运行方式（示例）

设定目标码率 (A = 1) Mbps，时间片为 5ms。

- **计算每个时间片能发送的字节**：

$$\frac{1\,\text{Mbps} \times 5\,\text{ms}}{8} = \frac{1\,000\,000\,\text{bit/s} \times 0.005\,\text{s}}{8} = 625\,\text{bytes}.$$

- **若编码器在某 5ms 内产出 800 bytes**，则 Pacer 队列里会多出 175 bytes 需要等到下一个或更多个时间片再发出。
- **Pacing factor** 若是 1.5，则在短期内 Pacer 可能以 1.5Mbps 速率发送，实际上每 5ms 可以发 625×1.5≈938 bytes。但它不会一直这样加速，如果下一次编码器再产生更大爆发，Pacer 队列继续累积，总体平均值还是绕不过 1Mbps 这个上限。

---

5. Pacer 对拥塞控制的贡献

1. **减少发送侧突发**
   - 避免了"编完帧就一股脑发送"的模式，降低了送到瓶颈路由的突发性。
   - 这让接收端的延迟梯度检测也更准确，因为大部分延迟波动来自真正的网络瓶颈，而不是发送端突发。
2. **与基于延迟 / 丢包控制相配合**
   - 无论是基于延迟的反馈 (A_r)，还是基于丢包的控制 (A_s)，最后都变成一个目标码率 A。Pacer 以更均匀的方式把该码率落实到包级调度上。
   - 一旦接收端或发送端的控制器认定需要降速，Pacer 得到的新、更低的 A，就会相应地减少"每 5ms 能发送的字节数"，从而实际降低发往网络的数据。

---

6. 小结

- **Pacer 的定位**：它是发送端把"抽象目标码率"转化成"具体、持续的限速发送调度"的模块。
- **论文中提及但未深入**：在 *Analysis and Design of the Google Congestion Control* 里，这一部分只在 3.5 节简单描述了"Pacer and Encoder"如何配合远端速率控制 (A_r) 和丢包控制 (A_s) 的最小值来实际驱动发送速率，并提到"pacing factor = 1.5"的做法。
- **通俗理解**：
  1. Pacer 相当于"带速率限制的水龙头"，以小固定时间片方式均匀放水，保证不一次性把水桶里过多的水瞬间倒出；
  2. 避免短时突发导致网络队列积压过多；
  3. 保持整体发送平均速率不超出上层设定的目标限制。

这就是 WebRTC（以及 GCC 发送端）中 Pacer 模块的核心原理与作用：**把目标码率平滑地映射到时间尺度上，减少突发，进一步降低端到端延迟与抖动。**

Padder/FEC（发送端）

- 在 GCC 的发送端框架图（论文图 1）里，除了 Pacer 和编码器外，还可以看到一个 "**Padder/FEC**" 模块（有时称 "Padding/FEC"）。该论文并未对其进行详细介绍，不过我们结合 WebRTC 的常见实现原理，可以得知它的主要作用是**在编码器无法或不需要使用全部目标带宽时，通过发送冗余包或填充包来维持一定的发送速率**，并在需要时提供前向纠错 (FEC) 以对抗丢包。下面分几个方面加以说明。

---

1. 为什么需要 "Padder"?

1) 维持探测或稳定的发送速率

- **编码器产出低于目标码率**
  在实际视频场景中，如果画面内容比较简单（比如背景静止、参与者不多），视频编码器可能难以产生足够多的数据来"吃满"拥塞控制算法指定的目标速率 (A)。举例说，目标码率是 2 Mbps，但低复杂度画面编码只用到 1 Mbps。
- **"空洞"会降低带宽探测的准确性**
  GCC 等拥塞控制算法需要一条"尽量饱和但又不过量"的发送流来测量延迟、丢包等。如果实际发送速率大大低于目标值，那么就无法有效探测是否仍有网络带宽富余。
- **Padder 在此时发挥作用**
  当编码器的实际输出低于拥塞控制设定的速率，Padder 会发送一定数量的"填充包 (padding packets)"。这些包不包含真实视频数据，可被标记为冗余、低优先级的数据，用来填补速率差，使整体发送速率更接近目标值，以保持带宽探测和码率调控的连贯性。

2) 降低突发和延迟波动

- **Pacer + Padder 配合**
  Pacer 在调度每个时间片发送量时，如果从视频编码器队列里拿不到足够数据，就会调用 Padder 提供"虚拟数据包"，从而保持平滑且接近恒定的发包节奏，不会出现空闲的发送时间片。
- **网络端保持相对稳定**
  这样做也使得网络流量图更平滑，避免实际速率忽高忽低，让拥塞控制测量（如延迟梯度检测）更准确。

---

2. 为什么需要 "FEC"?

1) 抗网络丢包，提高质量

- **FEC（Forward Error Correction）**
  在多媒体实时传输中，可以通过附加冗余信息（FEC 包）来在接收端重建丢失的原始媒体包，而无需等待重传（如同 TCP 那样）。
- **减少可感知的视频卡顿**
  对于实时视频/音频流，若丢包无法立即重传，画面或声音就会出现缺失或明显损伤。FEC 帮助在丢包时快速恢复一部分数据，减轻卡顿感。

2) 与丢失率/延迟的平衡

- **FEC 并不免费**
  增加 FEC 包就意味着占用额外带宽，如果网络带宽充足或者目标码率尚有富余，才能选择做 FEC 而不至于进一步提高拥塞或牺牲视频主码流质量。
- **动态自适应**
  WebRTC 通常允许发送端（或中间层的冗余策略）根据丢包率、延迟、可用带宽等情况，决定是否启用 FEC、启用多少冗余度。若网络稳定且丢包率低，可减少 FEC 包；反之则增大 FEC 冗余保护。

---

3. Padder 与 FEC 的结合

在实际实现中，"Padder/FEC" 往往是**同一逻辑框架**：

- 当编码器未产生足够数据时，可以选择：
  - **发送零负载填充包**（Padding），纯粹是维持发送速率；
  - 或者利用这部分"空闲带宽"发送一部分 "FEC 冗余" 数据包，这些包在丢包时能起到纠错作用。
- **自适应策略**
  一些实现会先尝试把空余带宽给 FEC，以提升抗丢包能力；若仍有余量，或者 FEC 配额也到顶了，再放 Padding。也可以只做 Padder，而不启用 FEC。

---

4. 与 GCC 的交互

1. **当目标码率 (A) 升高**
   如果拥塞控制器判断可用带宽升高，编码器可能来不及瞬间提升视频质量（或因为场景简单本身码率低），这时"Padder/FEC" 会填充空余，保持实际发送速率与 (A) 大致相符。
2. **当网络出现高丢包**
   - 基于丢包控制器可能触发大幅降速，但同时**若我们选择启用 FEC**，也可能适度发送部分 FEC 包来对抗丢失，对最终画面可用性有好处。
   - 如果丢包特别严重，实际上也需要注意别因为 FEC 的额外负荷进一步拥堵。GCC + FEC 会根据网络情况自适应调节。

---

5. 典型数据流动示例

1. **编码器输出** 产生实际视频包 X kbps；
2. **如果 X < A**，则 "Padder/FEC" 模块计算还能发送多少 kpbs 而不超过 (A)。
   - 可能一部分带宽分给 FEC (比如 N kbps) 用于冗余纠错，
   - 若还有剩余则填入纯"Padding"包 M kbps（不含实质有效载荷），
   - 使最终送入 Pacer 的总量约为 X + N + M ≈ (A)。
3. **Pacer** 以固定节奏分时发包，达成平滑发送。
4. **网络** 接收端测量到一个相对平稳且近似 A 的数据流量，从而能准确判断延迟梯度等拥塞信号。

---

6. 小结

- **Padder**
  - 解决"视频编码器产出 < 目标速率"时，如何保持带宽探测与流量平滑的问题；
  - 避免发送端速率大起大落、影响拥塞控制精度。
- **FEC**

- 在可用带宽和时延允许的范围内，额外发送纠错冗余包，减少画面/音频因丢包产生的可感知质量劣化；
- 一种在实时媒体传输中常见的对抗丢包策略，可与 ARQ（请求重传）互补。

- **在 GCC 中的位置**
  - 论文图 1 里该模块与 Pacer、编码器一起构成“发送速率生成/驱动”部分；
  - 当延迟控制器和丢包控制器给出目标速率后，该模块负责填充或 FEC 使实际发送流量与目标保持一致，从而让拥塞控制能够有效探测并控制队列。

虽然论文本身并未深入介绍 Padder/FEC 机制，但从 WebRTC 的整体设计可知，这部分对**维持平滑速率**和**增强数据可靠性**都至关重要，是 GCC 发送端端到端拥塞控制闭环中的重要补充。

宏观总结

- 以下是一份较为系统、宏观的 GCC（Google Congestion Control）整体工作流程概述。它涵盖了**接收端的各模块**（基于延迟的控制器）是如何内部配合，以及**发送端的各模块**（基于丢包的控制 + 发送速率驱动）是如何互动，最终形成端到端低延迟、适应性良好的拥塞控制闭环。

一、接收端（基于延迟的控制器）

在 GCC 中，接收端主要通过**测量单向延迟的变化**（延迟梯度）来判断网络拥塞是否正在发生，并将得到的码率估计值 A_r 反馈给发送端。论文中把这部分称为 **Remote Rate Control**（远端速率控制），其核心模块包括：

1. **到达时间滤波器（Arrival-time Filter / 卡尔曼滤波）**
   - **功能**：在每帧（或每批数据包）到达时，测量

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

     得到原始的“单向延迟变化”（延迟梯度测量）。然后使用**卡尔曼滤波**等方法对该测量值去噪，得到平滑的梯度估计 $m(t_i)$。
   - **意义**：避免瞬时网络抖动造成的测量噪声，使拥塞检测更稳定、更准确。
2. **自适应阈值块（Adaptive Threshold）**
   - **功能**：动态地调节阈值 $\gamma(t_i)$，用于判断“过度使用 / 不足使用 / 正常”。
   - **关键思想**：阈值 $\gamma(t)$ 根据当前测量到的 $|m(t)|$ 作自适应上、下调整：
     - 当 $|m(t)|$ 经常高于$\gamma(t)$，则 γ 以较小速率上升（k_u）；
     - 当 $|m(t)|$ 小于$\gamma(t)$，则以较快速率下降（k_d）。
   - **目的**：1）当瓶颈队列小，$|m(t)|$不大时能够及时捕捉到延迟膨胀；2）在并发大流（如 TCP）时不要太敏感而“饿死”自己的流量。
3. **过度使用检测器（Overuse Detector）**
   - **功能**：比较 “$m(t_i)$” 与 “$\gamma(t_i)$” 来判定网络是否“过度使用”（Overuse）、“不足使用”（Underuse）或“正常”（Normal）。
   - **判定条件**：若 $m(t_i) > \gamma(t_i)$ 并持续一段时间（约 100ms），则判定 Overuse；若 $m(t_i) < -\gamma(t_i)$ 则为 Underuse；否则为 Normal。
   - **输出**：每次生成一个信号 *s*（Overuse/Underuse/Normal）发送给远程速率控制器的有限状态机。
4. **远程速率控制器 FSM**
   - **功能**：根据过度使用检测器的信号 *s*，在 “增加 (Increase) / 减少 (Decrease) / 保持 (Hold)” 三种状态间切换，并相应计算下一步的推荐发送码率 $A_r(t_i)$。
   - **主要规则**：

     $$A_r(t_i) = \begin{cases} 1.05 \times A_r(t_{i-1}), & \text{if OveruseDetector} = \text{Normal (Increase)} \\ 0.85 \times R_r(t_i), & \text{if OveruseDetector} = \text{Overuse (Decrease)} \\ A_r(t_{i-1}), & \text{if OveruseDetector} = \text{Underuse (Hold)} \end{cases}$$

     同时限制 $A_r(t_i) \le 1.5\, R_r(t_i)$，避免过度猛增。
   - **意义**：维持“低排队延迟”同时又进行有效的带宽探测——遇到过度使用就降速，排队减少后就正常或适度增速。
5. **REMB 处理模块**
   - **功能**：把 FSM 输出的码率 *Ar(t)* 以 RTCP 扩展报文“Receiver Estimated Maximum Bitrate (REMB)”的方式反馈给发送端。
   - **发送时机**：

- 默认每隔 1 秒发送一次；
- 若 $A_r(t)$ 相比上次下降超过 3%，则**立即**发送，以加速通知发送端"赶紧降码率"。

（小结：接收端内部协作）

- **到达时间滤波器**得到平滑的单向延迟梯度 → **自适应阈值块**持续更新阈值 (\gamma(t)) → **过度使用检测器**判断当前是 Overuse / Normal / Underuse → **远程速率控制器 FSM** 调整推荐速率 $A_r(t)$ → **REMB 模块**定期/即时向发送端通告该速率。
- 这一整套逻辑形成了**基于延迟梯度**的拥塞侦测和速率反馈。

二、发送端

发送端在得到**接收端反馈 (A_r)** 的同时，也会基于 RTCP 中的"丢包率"做补充性的拥塞判断，最终综合两者生成实际的发送速率并落地到发送调度过程（Pacer、编码器、Padder/FEC）。

1. **基于丢包的控制器**
   - **来源**：发送端通过 RTCP Receiver Report 获取丢包统计 $f_l(t_k)$（某一时间窗口内丢失的包数/总包数），进而判断是否需要大幅降速或缓慢增速。
   - **核心规则**（论文公式 (5)）：

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

     - 丢包率 > 10% ⇒ 强力减速（乘以 (1 - 0.5×丢包率)）；
     - 丢包率 < 2% ⇒ 增速 5%；
     - 否则保持。
   - **意义**：丢包信号可在延迟控制还没来得及反应时，立即告知发送端存在严重拥塞；或者在丢包率很低时，做适度带宽探测。
2. **综合速率决定**
   - 最终目标发送速率 $A(t) = \min\{A_s(t),\, A_r(t)\}$。
   - 即同时受"丢包控制器" + "接收端延迟反馈"约束，取二者的最小值，确保**既不让丢包率过高**，也**不使排队时延过大。**
3. **Pacer（发送端调度器）**
   - **功能**：将上一步得到的目标速率 $A(t)$ 以更均匀、更平滑的方式在毫秒级时间片里发包，避免"帧级别突发"造成瞬时队列飙升。
   - **Pacing Factor**：论文中提到当编码器不时输出过量数据包时，Pacer 允许短时以 $1.5 \times A$ 的出包速率排空队列，但长时平均依然受限于 A。
   - **效果**：显著减少发送侧突发、平滑网络流量曲线，有助于稳定延迟。
4. **编码器（Video Encoder）**
   - 根据 $A(t)$（或稍迟一点更新后的码率）进行**自适应编码**，动态调整视频图像量化参数、帧大小等，使输出码率尽可能逼近拥塞控制给定的目标值。但通常不能像 Pacer 那样频繁变动，以避免画面质量频繁闪变。
   - 若编码器在某一时段产生的码率**低于** $A(t)$，则发送端依靠后述 Padder/FEC 填补剩余带宽（如果希望继续探测或添加 FEC）。
5. **Padder/FEC（Padding & Forward Error Correction）**
   - 当编码器产出低于 A 时，为了保持带宽探测或稳定发送，发送端会发送**纯填充包** (Padding) 或用于纠错的 **FEC 包。**
   - FEC 包含纠错冗余信息，可在丢包时帮助接收端恢复部分缺失数据；Padding 则是"空负载"包，用来维持网络上传输速率，便于延迟检测器更好地量测带宽余量。
   - 与 Pacer 配合，保证发送端输出流**平稳且接近**目标速率。

（小结：发送端内部协作）

- **基于丢包的控制器** 定期更新 A_s；同时，接收端给出的延迟反馈 A_r 被接收 → 取 $\min(A_s, A_r)$ 作为最终发送目标速率 **A。**
- **Pacer** 依据 **A** 将实际包流拆分到微小的时间片去发送；
- **编码器** 尽可能产生符合 **A** 的视频流；若编码器输出不足，则 **Padder/FEC** 会补冗余以维持速率并增强抗丢包能力。

三、接收端与发送端之间的端到端交互

1. **接收端 → 发送端**
   - 通过 RTCP（含丢包分数/REMB/延迟反馈等），周期性或事件驱动地把**丢包率**、**推荐发送码率 A_r** 等信息传回发送端；
   - 其中 REMB 特别用于告知"基于延迟梯度"得出的最大安全码率，**可每秒**或**一旦大幅降速时**立刻发送。
2. **发送端 → 接收端**
   - 发送端据此更新发送速率、在下一时段发出的视频/音频 RTP 流在新的码率水平上进入网络；
   - 接收端继续观测到达间隔、进行延迟梯度更新，周而复始。
3. **闭环形成**
   - 接收端：若检测到排队变多(过度使用) ⇒ 降低 A_r ⇒ RTCP 发送；
   - 发送端：立刻/定期收到后，减少发包，导致排队回落；接收端再次感知延迟渐趋正常 ⇒ 允许增速；
   - 同时丢包信息也会让发送端做大幅或小幅调整，弥补仅仅依赖延迟梯度可能存在的盲区。
   - 这样就形成端到端**低时延、动态适配带宽**的自调节循环。

四、整体小结

- **接收端“基于延迟的控制”**：
  1. 通过 **到达时间滤波器** 获取平滑的延迟梯度；
  2. **自适应阈值** 确定是否对该梯度敏感；
  3. **过度使用检测器** 综合判断 Overuse / Normal / Underuse；
  4. **远程速率控制器 FSM** 作“加 / 减 / 保持”码率的具体决策并输出 A_r；
  5. **REMB** 决定何时告诉发送端这一码率；
     ——形成实时检测排队膨胀的机制，保证延迟不被拉高。
- **发送端“基于丢包的控制” + “发送速率驱动”**：
  1. 根据 RTCP 丢包率推断拥塞（<2% 小增速、>10% 大减速，2%~10% 保持）；
  2. 将丢包控制得到的 A_s 与接收端反馈的 A_r 取最小值；
  3. **Pacer** 以平滑调度的方式落实到实际 RTP 包的发送；
  4. **编码器** 按照新的目标速率编码视频；
  5. 若编码器产出不足，**Padder/FEC** 用填充包 / 冗余包维持或探测剩余带宽；
     ——在避免大丢包的同时，也保证对可用带宽的充分利用。
- **端到端协同**
  - 接收端和发送端通过定期 RTCP 报文互通延迟、丢包、推荐码率等关键信息，使得系统保持**低延迟**（不出现大的排队或过量丢包），同时在网络条件改变时也能**及时自适应**，追求尽可能高的视频传输质量。

通过这样的双轨（延迟 / 丢包）互补，以及一系列子模块（卡尔曼滤波、阈值自适应、有限状态机、Pacer、Padding/FEC 等）的精细配合，GCC 实现了对带宽动态变化的快速响应，并将端到端排队延迟控制在较低水平，从而满足视频会议等实时业务对于**低延迟、高吞吐、稳定质量**的综合需求。

相关论文实验结论

以下结论来自于《Analysis and Design of the Google Congestion Control》文章。在作者的实验评估（第 6 节）和结论（第 7 节）部分，论文针对 WebRTC 环境下的 GCC 做了多场景的实验测试，并总结出若干结论性看法。以下归纳主要的实验结果与作者的最终结论：

---

1. 实验设计与测量指标回顾

- **测试平台**：作者在可控的 Linux 测试环境中，通过令牌桶过滤器 (TBF) 设定瓶颈链路的容量与排队大小，还使用 NetEm 调整传播延迟等；
- **度量指标**：
  1. 信道利用率 ($U = R_r / C$)，即平均接收速率除以链路容量；
  2. 丢包率 (l)，在 RTCP 反馈中测量的丢失字节数占总发送字节数的比例；
  3. 排队延迟的中位数与 95% 分位数（用 $RTT(t) - RTT_{min}$ 估计）；
  4. Jain 公平指数（JFI），衡量多路流之间的带宽分配公平性；
  5. 其他用于说明算法如何跟踪容量突变和控制时延的时间序列图。

---

2. 主要实验场景及结论

论文第 6 节罗列了几个典型场景，结果如下：

2.1 单个 GCC 流，链路容量可变

- **场景**：将瓶颈容量在若干时间区间内从 1Mbps → 2.5Mbps → 0.5Mbps → 1Mbps，共持续 200 秒。
- **发现**：
  1. GCC 能够**跟踪时变链路容量**，在容量提升时逐步拉高码率，在容量下降时能够迅速收敛以避免大范围丢包；
  2. **排队延迟控制较好**：RTT 在大部分时间接近传播延迟 (50ms)，极端情况下的 95% 分位数也相对可接受；
  3. 在容量从 0.5Mbps 再恢复到 1Mbps 时，GCC 需要约 25 秒才回到新容量值，信道利用率约在 84% 左右，说明在增大带宽的探测上略显保守，但能保持低排队。

2.2 多个并发 GCC 流（协议内公平性）

- **场景 A**：在一条 3Mbps 链路上同时跑 3 个 GCC 流，每个流相隔 30 秒启动。
- **结论**：
  1. **没有"迟到者效应"**：即后启动的流不会被前面的流"饿死"，所有流都能渐渐收敛到接近平均的带宽份额；
  2. **Jain 公平指数**约为 0.93，表明流间的速率分配比较公平；
  3. 几乎没有丢包，排队延迟的 50% 分位数接近传播延迟（10ms 额外排队时间），95% 分位数约 61ms，体现了"低排队时延"特性。
- **场景 B**：在可变链路容量条件下放 2 条 GCC 流，也展现了相似的收敛与公平特性，Jain 公平指数约 0.87。

2.3 一个 GCC 流与并发长寿命 TCP 流（协议间公平性）

- **场景**：1 个 GCC 流与 99 个长寿命 TCP 流共享一个 100Mbps 的链路；TCP 流在 100s ~ 300s 的时间段内处于活动状态。
- **结果**：
  1. GCC 流在 TCP 流到来后，从初始较高比特率降到约 1Mbps 附近，与整体 100Mbps / 100 流 大致对应的"公平份额"；
  2. 当 TCP 流结束后，GCC 又能重新升到更高速率；
  3. 证明 GCC 不会被大规模 TCP 流"饿死"，也不会过度侵占带宽，而能适度共享资源。

2.4 两个 GCC 流在存在短寿命 TCP 流时（网页突发等场景）

- **场景**：2 个 GCC 流在 2Mbps 链路中持续发送，期间每隔 12s 注入一个仅持续 3s 的 TCP 短流(模拟网页等小文件下载)。
- **结论**：
  1. 每当短流加入会使 RTT、丢包等出现瞬时波动，但 GCC 流码率相应收缩；
  2. 短流退出后，GCC 又能回到其"平分"1Mbps 左右；
  3. 总体信道利用率约 72%，表明在频繁突发的场景下，GCC 维持了较低排队和适度的资源使用。

2.5 反向流量的影响

- **场景**：1Mbps 下行方向跑 GCC 视频流，而在反向上行路径另有一个 TCP 流。
- **结论**：
  1. 反向流的出现并不会让 GCC 流显著丢包或降速，这说明**单向延迟测量**对反向拥塞不敏感；
  2. GCC 流维持 90% 左右的信道利用率，且丢包几乎为 0，队列时延也依旧比较低（10ms 中位、159ms 95% 分位）。
  3. 这印证了"单向延迟"方法能有效避免对反向拥塞的干扰。

---

3. 作者的总体结论（第 7 节）

在第 7 节"结论与未来工作"中，作者做了以下归纳：

1. **GCC 能在没有并发竞争的环境下尽量匹配链路容量**
   - 在容量突变时，有一定的上升/下降周期，维持较低排队延迟和丢包率；
2. **在存在异构或并发流竞争时，仍能保证合理的带宽份额和延迟**
   - 对内（多个 GCC 流）有良好的"协议内公平性"，对外（与 TCP 并发）有基本的"协议间公平性"；
3. **对反向流量不敏感**
   - 单向延迟测量能避免反向流导致的 RTT 干扰。
4. **整体延迟保持低水位**
   - GCC 通过延迟梯度检测，避免大排队积累，RTT 多数情况下接近传播延迟，95% 分位数也不算过高。

作者也表示：在大规模真实用户场景下还需要更多测量与收集数据进行验证，但从实验室评估来看，GCC 具备在实时通信（视频会议）环境中保持**低延迟、高带宽利用、并对并发流量友好**的特性。

小结

综上，在作者的实验与结论部分，针对WebRTC（或 Chrome WebRTC 堆栈中所使用的）GCC 所做的测试表明：

- **速率对网络环境变化有较好适应性**，能相对平稳地提升或收敛；
- **排队延迟始终维持在较低水平**（低队列积累、低丢包）；
- **在多路同类（GCC）或异类（TCP）流并发时**, 实现了比较均衡的带宽分配与可用性。

这是作者对 GCC 在 WebRTC 场景下的实验性结论，也正是论文所要突出的主要价值所在。

其他问题

- 如果处于网络十分拥塞的弱网环境下，或者发送和接收方的中一方或者双方上传或者下载带宽严重受限的环境下。Padder/FEC模块的冗余纠错机制是否会进一步加剧整体通信环境的恶化？在实际应用中又是如何应对这种场景的

> 当网络带宽本身十分有限、或者已经处于严重拥塞时，引入额外的 FEC 冗余数据确实可能让通道负担更重，从而“火上浇油”。在实际系统（包括 WebRTC）里通常会有相应的自适应策略来应对这种问题，保证在“弱网”或“极度拥塞”环境下不会盲目塞入大量 FEC 包，反而进一步阻塞。以下从几个角度说明实际做法：
>
> ---
>
> ### 1. FEC 不会无限制地叠加
>
> - **带宽估计约束**
> 在 WebRTC 的拥塞控制（例如 GCC）中，发送端有一个“目标码率” (A(t))（由延迟控制和丢包控制共同决定）。当网络拥塞时，(A(t)) 会被下调到一个较低的水平，编码器+Padder/FEC 整体也只能“服从”这个限速。
> - 若为了添加 FEC 而无视带宽限制，结果只能是持续丢包或延迟暴涨，本身就让“过度使用检测器”频繁报警 → 进一步压低目标码率。
> - 因此系统会根据当前可用带宽、丢包率等信息，决定“能不能、以及该送多少 FEC”。在极度拥塞或弱网情况下，FEC 往往被关闭或仅保留非常少的冗余。
> - **自适应冗余比率**
> 一些实现中（如 SVC、ULPFEC 等），会根据当前丢包率、时延限制动态调节 FEC 冗余度。丢包严重但带宽依然可用时，可以适度提高冗余；如果带宽非常受限，则减少 FEC 甚至完全关闭，以避免雪上加霜。
>
> ---
>
> ### 2. 编码器本身会“降码率”或“降画质”
>
> - **视频编解码层的自适应**
> 除了在“FEC 或者 Padder”层面处理，实际的 WebRTC 系统也会让视频编码器因应弱网场景，降低分辨率、帧率或提高量化系数（降画质），从而**减小主流视频数据**的占用带宽。这样一来即便再加一点点 FEC 冗余，也不会超过总带宽限制，或者所加的冗余比率保持得更合理。
> - **保护关键帧 / 重要层**
> 若采用多层视频编码（SVC / simulcast），FEC 或冗余更可能只重点保护最基础的关键层（base layer），以保证最低画质持续不断，而对增强层则少做 FEC 或不做 FEC，这样既避免对总带宽的冲击，又能保证弱网下画面仍有基本流畅度。
>
> ---
>
> ### 3. “Padding”部分在弱网中也会相应减少
>
> - **Padding 目的**
> 是在编码器码率较低时“填充”达到目标发送速率，以帮助带宽探测。但如果网络真的无法承受更高速率（因为拥塞而降速），那么目标速率本身会降到足够低，Padding 也就不会大量发送；换言之，如果可用带宽只有 200kbps，但编码器内容需求已有 180kbps，Padding 只会填个很小差值或者甚至为零。
> - **对拥塞的反应**
> 当过度使用检测器判断网络极度拥塞，会压低 A_r；丢包控制器也会压低 A_s；最终发送端速率 (A) 自然就低。此时“Padder”实际上不会再去拼命塞包，因为那只会让延迟/丢包更糟糕。
>
> ---
>
> ### 4. WebRTC 中的实际应对思路
>
> 1. **协同自适应**
> - GCC 会随着拥塞升高迅速降低推荐码率；
> - 丢包控制也看到丢包高时迅速减速；
> - 编码器下调分辨率、帧率来减小主流码率；
> - FEC 冗余可自适应地减小甚至关闭；
> - Padding 在网络拥堵时基本也发不出去或被大大削减。
> 这样一来，不会出现“FEC 占满已非常狭窄的可用带宽”的极端。
> 2. **“弱网”场景常见体验**
> - 用户在网络很差时（上/下行都不佳），对端画质会明显降低、码率也低，有时甚至退化到非常模糊的分辨率甚至仅有音频维持。
> - 在这期间，FEC 或者 RTX 重传等往往也随之减少，因为带宽实在不足，再做大量冗余也没意义。很多时候 WebRTC 会优先保证音频的传输质量，让视频被进一步牺牲。
> 3. **策略性平衡**
> - 如果丢包率高但带宽也有一定空间，则会适度增大 FEC 比率，以提升抗丢包性能；
> - 如果带宽极度受限，冗余探测就会变得没有价值或得不偿失，一般就减少甚至停掉 FEC。
>
> ---

> **5. 结论**
>
> - **在极度拥塞或弱网中，引入 FEC 冗余确实可能使带宽压力更大，但 GCC（或其他自适应拥塞控制）不会允许它无限制地占用带宽**；
> - 当检测到网络性能极差时，系统会通过**降低主码率**（编码器降清晰度）、**减少或关闭 FEC**、**减少或停止 Padding**等多重方式共同协作，最终使得总体发送速率跌到一个更低的层面，以保护音视频的基本连通性与延迟。
> - 因此，在真正弱网环境中，FEC 不会“无限度地进一步恶化网络”，而是会在可能的条件下做有限度的纠错，否则干脆放弃，保障最核心的可用性。

- 为什么GCC会与TCP流产生并发竞争

> 在现实网络环境中，任何两个（或多个）不同的应用或传输协议只要共用同一条瓶颈链路，就会产生并发竞争。**即使你的 WebRTC 视频会议流本身基于 UDP + GCC**，也可能与系统中其它流量（包括大量使用 TCP 的应用）共享家庭宽带、办公室局域网的出口、或者移动数据连接等。具体原因包括：
>
> 1. **现代网络中的多种应用同时存在**
>   - 用户端常同时使用各种网络服务：视频会议（WebRTC / GCC）、文件下载（HTTP/TCP）、云存储同步（TCP）、网页浏览（HTTP/TCP）等等。
>   - 当这些不同类型的连接都走向同一个路由器或运营商出口，就会在瓶颈链路上进行竞争。
> 2. **TCP 在互联网上依然最常见**
>   - 大量常见应用（浏览器访问网页、后台软件更新、网盘同步、P2P、云游戏等等）多数基于 TCP 协议传输。
>   - 即使你的音视频流用 UDP/GCC，其他进程或用户照样会有各种 TCP 流量和你共享线路。
> 3. **GCC 流本身并不隔绝外部流量**
>   - GCC（Google Congestion Control）并不是像 MPLS 等专有隧道一样隔离带宽；它只是一种在 UDP 上的端到端拥塞控制算法，依然要和所有其他走同一网络的流共享公共路由器缓冲或 Wi-Fi/AP 出口。
> 4. **同一局域网/路由器出口**
>   - 比如在家用环境下，家中路由器带宽有限，同时有人在开视频会议（GCC 流），有人在看 YouTube（TCP 流），还可能在下载大文件（TCP），便会在出/入口带宽上互相竞争。
> 5. **并发流量是评估拥塞控制的重要现实场景**
>   - 正因为现实网络里 UDP 流不可能只与 UDP 流竞争，拥塞控制协议设计者需要确保在面对传统 TCP 流量时能有一个“公平且稳定”的表现：既不被 TCP 流“挤死”，也不无限制地抢占 TCP 流的带宽。
>
> 综上，**GCC 与 TCP 流产生并发竞争**是源自真实网络环境中多协议、多应用共同使用同一物理/逻辑瓶颈带宽的普遍现象，而并非 GCC 自己“主动要和 TCP”竞争。这是为什么在论文和实际评估里，会着重测试“一路 GCC 流 + 一路或多路 TCP 流”的场景，以检查 GCC 在真实场景中与常见 TCP 流量共存时的性能和公平性。