RFC 3550

中文文档

翻译说明

译者注：

本文档提供给那些有一定英语基础的读者，所以文中涉及到的专有词汇均以英文原词给出，给读者带来的不便请谅解。

本文档知识产权声明部分仅供参考，如有涉及到与知识产权有关的地方请参看原文或询问相关法律人式。

由于工作量太大，附录部分没有列入译者翻译之列，请谅解。

感谢罗总及工程师 **gale** 给予的支持和帮助。

由于译者水平有限和时间冲忙，文中一定有错译的地方，译者尽量将这种情况降到最小，但不排除译者本身就理解错误的情况，欢迎任何对本书有兴趣的读者提出任何质疑，邮箱：**xingkongft@163.com**，**QQ：601211088**。

Network Working Group H. Schulzrinne
Request for Comments: 3550 Columbia University
Obsoletes: 1889 S. Casner
Category: Standards Track Packet Design
R. Frederick
Blue Coat Systems Inc.
V. Jacobson
Packet Design
July 2003

RTP: 应用于实时应用的传输协议

备忘录的状态:

本文档讲述了一种 Internet 社区的 Internet 标准跟踪协议，它需要进一步进行讨论和建议以得到改 进。请参考最新版的“Internet 正式协议标准”(STD1)来获得本协议的标准化程度和状态。本备忘录的发布 不受任何限制。

摘要:

本文档阐述了实时传输协议（RTP)，RTP 提供了端到端的网络传输功能，适合于通过多播或单播网络服务的实时数据传输应用，例如：音频，视频等，RTP 不强调资源保留(does not address resource reservation) 而且并不保证实时服务 QOS(quality-of-service)， 实时控制协议(RTCP)对数据进行监控和提供最小的控制和鉴别功能。 RTP 和 RTCP 是网络层和传输层上面的独立协议。 本协议支持 RTP 级别的 translators and mixers。

本文档大部分和 RFC 1889(已废)。 在封包格式上并没有改动，(no changes in the packet formats on the wire)而仅仅是在怎样用本协议的规定和算法管理进行了改动，最大的改动是对定时算法的增强，当试图发送 RTCP 报文时，发现有很多个参与者同时加入一个会话(a session)时，提供最小化传输控制。

1. 介绍

本文档详细描述了实时传输协议(RTP)，RTP 为实时数据传输例如交互的音频和视频提供了端到端传输服务。服务包括有效载荷的类型确认，序列编码，时间戳和传呼监控(delivery monitoring)。典型应用时利用 UDP 的多路技术了校验和服务而在之上运行 RTP，两者都提供一定的传输控制功能。然而，RTP 还可以与其它适合的协议并用(参见第 11 章节)，如果底层网络支持多路分发，RTP 还可以提供数据给多路终点。

需要注意的是 RTP 不提供任何的机制以保证数据的实时性并且也不保证其它的 QOS(quality-of-service)，而是依赖底层的服务来提供这些功能，RTP 既不保证传输或者是阻止无序传输，也不假定底层网络是可信任的和传输报文是有序的。RTP 中的序列号允许接收器重建发送器发来的报文，但是序列号还可以用来确定报文的合适位置，例如：在视频解码中，序列中没有了需要解码的报文。

虽然 RTP 协议最初是为了满足多媒体会议的需求的而制定的，然而 RTP 协议也适用于连续数据的存储，交互式的分布仿真(interactive distributed simulation)，active badge，管理和测量应用。

本文档定义了 RTP，包含两个紧密联系的部分：

o 实时传输协议(RTP)，传输具有实时特性的数据。

o RTP 控制协议(RTCP)，监控 QOS 和传递会话中参与者的信息。而后者对于“宽松会话”(loosely controlled)来说是足够了，即没有明确的成员控制和建立，但是协议没有必要支持一个应用的控制需求的全部。在一个单独的会话控制协议中可能全部或部分包含了这项功能，这不在本文档讨论之内。

RTP 代表了一种新类型协议，遵循应用级框架(application level framing)和(integrated layer processing)，由 Clark 和 Tennenhouse [10]提出的。即 RTP 可以很容易的扩展[P5]以提供某种特定应用的信息，并且可以经常集成进某种应用处理而不是作为补充而成立单独层。RTP 是有意的做成不完整的协议框架。本文档详细描述了那些所有应用都适用的常用功能。不像常规协议那样为了使协议更加一般化而提供附加的功能或是为了增加剖析功能而提供一个选项机制。RTP 可以根据需要而修改或者增加头字段。例子见于 5.3 和 6.4.3。

因此，除了本文档外，特定应用还需要其它的文档(见第 13 章)：

o 一个剖面详述文档，定义了一系列有效载荷码字和它们映射格式(例，媒体编码)，也定义了为了某一 类特定应用而需要扩展和修改的 RTP。典型的就是仅仅基于一个 profile 而运行的应用。在 RFC 3551 [1]可以找到用于音频和视频的 profile。

o 有效载荷格式说明文档，定义了怎样在 RTP 中携带特定的有效载荷，例如音频和视频编码。

可以在[11]中找到关于实时服务和补充算法的讨论，也有关于一些 RTP 设计决议的背景讨论。

1.1 术语

本文档中关键的单词 "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" 可以在 BCP 14，RFC 2119 [2]中找到解释 and indicate requirement levels for compliant RTP implementations。

2. RTP Use Scenarios

以下章节描述了使用 RTP 的部分特性。选择的例子是用来阐明基于 RTP 的应用的基本操作，而不是限制 RTP 仅能用于此类应用。在这些例子中，RTP 是在 IP 和 UDP 之上 carried，遵循在 RFC 3551 中为音频和视频而建立的惯例(conventions)。

2.1 简单的多播音频会议

IETF 工作组利用 Internet 上的为了语音通信而提供的 IP 多播服务讨论了最新的协议文档，基于某种分配机制，工作组主席得到了多播组的地址和一对端口。其中的一个端口是为了音频数据而准备的，另一个是用来控制(RTCP)报文的。将地址和端口分发给与会者。基于安全考虑，可以将数据报文和控制报文加密，这将在第 9。1 节中讨论，在这种情况中，会生成密钥并将它分发给与会者。详细的分配和分发机制不在本文档讨论之内。

与会者例如在 20 毫秒的持续期内发送一段(chunk)音频数据。将 RTP 头加在每一段音频数据的前面，然后插入在 UDP 报文中。RTP 头标明何种类型的数据封装(例如 PCM，ADPCM 或 LPC)以利于发送器拆分报文。例如， 给低带宽的参与者进行调节或者对网络拥塞进行重新操作。

像其它网络一样，Internet 偶尔会丢失报文或对报文重排序，或延迟不定长时间。为了防止这些意外，RTP 头字段中含有时间信息和序列号允许接收器按照源报文重建时序。在本例中，讲话者每 20 毫秒持续的发送音频报文段。在会议中的每个源的 RTP 报文的时序是独立重建的。接收者也可以应用序列号来确定丢失了多少报文。

在会议期间，工作组的成员会有人离开或加入进来，所以了解谁在和他们的收听质量如何是有用的。出于这个目的，每一个音频应用的实体(instance)周期的在 RTCP 端口多点传送接收报告和它的使用者的名字。就收报告标明接收者接收到的当前讲话者的通信质量，也可以用来控制自适应编码。除了使用者名字外，还包含其它的鉴别信息。当有人离开时，site 会发送 RTCP BYE 报文(第 6.6 节)。

2.2 音频和视频会议

如果会议期间同时应用了音频和视频媒体，它们会作为独立的 RTP 会话(sessions)来传输。即，将会为了每一个媒介开一对 UDP 端口来独立传输 RTP

和 RTCP 报文和多播地址。在 RTP 级(level)，音频和视频会话并没有直接的联系，除非在两个会话中使用者在 RTCP 报文中使用同样的 distinguished(canonical) name，这样两个会话就可以联系起来了。

这样做的一个动机就是允许某些与会者可以选择仅仅接收某一媒介的数据。在 5.2 节有更详细的阐述。尽管这样，利用会话中的 RTCP 报文携带的时序信息可以重新同步播放音频和视频。

2.3 Mixer 和 Translator

到目前为止，我们已经假设了所有的 sites 希望接收同样格式的媒体数据，然而这经常是不适当的。考虑这样一种情况，与会者中的大部分人在高速网络链路中而某个地方的小部分与会者却只能低速率连接。可以在低带宽区域放置一个 RTP 级的中继(relay)，称作 Mixer，而不是强迫播所有人都用低带宽。Mixer 将接收的音频报文重新同步以重建发送者的恒定的 20 毫秒间隙，将这些重建音频流混合成单一流，并且将音频编码翻译给一个低带宽然后通过低速率链路形成低带宽的报文流。这些报文可以单播也可以多播给更多的接收者。RTP 头字段包含了用于 Mixer 识别混合报文中的源的方法，这样就收者就可以识别出正确的讲话者。

一些音频会议的与会者可能是由高速链路连接的但却不是直接通过多播直达的。例如，他们可能在一个应用级的防火墙的后面，而防火墙不准 IP 报文同过。对于这些 sites，混合就不是必需的了，而另一种 RTP 级的中继称作 Translator 就派上用场了。安装两个 translators，防火墙一面一个，外面的 translator 所有的接收到的多播报文经过安全连接直达到内部的 translator，内部的 translator 再一次作为多播报文传输给限制在 site 内部网络的多播组。

可以基于很多种目的而设计 Mixers 和 Translators。一个例子就是在独立的视频流中，视频 Mixer scales 个人的图像，然后将它组合进一个视频流来模拟组场景。其它的关于转换包括一组主机仅仅发送基于 IP/UDP 的报文给仅仅基于 ST-II，或者是从没有经过重新同步或混合而单独的源来的 packet-by-packet 编码流。在第 7 章中有关于 Mixer 和 Translator 操作的详细介绍。

2.4 层编码

多媒体应用可以根据就收者的能力或者是网络拥塞来调节传输速率。许多的补充协议将速率适应能力放在源端。由于不同种类的接收者需求不一样的带宽，这种方法并不能很好的用于多播传输。 结果导致一个 least-common denominator scenario，其中网格中最小的管道指示直播的质量和忠诚度。

因此，速率适应的职责可以放在接收端，合并一个通过层传输的层编码(a layered encoding with a layered transmission system)。在通过 IP 多播的 RTP 中，the source can stripe the progressive layers of a hierarchically represented signal across multiple RTP sessions each carried on its own multicast group。接收者可以适应网络的不同并且通过加入适合的多播子组而控制就收带宽。

在第 6.3.9 节和第 8.3 节及第 11 章中详细阐述带层编码的 RTP 的用法。

3. 定义

RTP payload：在 RTP 报文中的有效载荷，例如音频采样或者压缩的视频数据。payload 格式和解释不在本文档范围之内。

RTP packet：包含定长的头字段的数据报文，源端或者 payload 数据可能是空(见下面)。一些低层的协议可能需要定义一种 RTP 报文封装协议。代表性的例子就是协议下的一个报文包含一个单独的 RTP 报文，但是也可以包含几个 RTP 报文，这取决于所用的封装方法(见第 11 章)。

RTCP packet：一种包含与 RTP 数据报文很相似的定长头字段的控制报文，紧随头字段的是结构元素，因不同 RTCP 报文而具有不同结构。格式见第 6 章。通常情况下几个 RTCP 报文合在一起作为一个混合的 RTCP 报文在协议下传送；这项功能由每个 RTCP 报文中头字段中的长度域来指定。

Port："传输协议用来区分主机下的不同应用，它是抽象出来的，TCP/IP 协议可以识别用正整数的端口" [12] OSI 模型中的传输层所用的传输选择器(TSEL)等同于端口。RTP 依赖更低层来提供例如端口机制来在会话中提供多播的 RTP 和 RTCP 报文。

Transport address：网络地址与端口的组合，用来识别传输级的终点(endpoint)，例如一个 IP 地址和一个 UDP 端口。报文是从源端 transport address 传送到目的端 transport address 的。

RTP media type：一个 RTP media type 是可以在单个的 RTP 会话中携带的 payload 类型的合集。由 RTP Profile 根据 RTP payload 类型指定 RTP media types。

Multimedia session：普通组参与者的一系列并发的 RTP 会话。例如视频会议中(这是一个 multimedia session)可能包含一个音频 RTP 会话和一个视频 RTP 会话。

RTP session：一组参与者利用 RTP 来通信的组合(association)。一个参与者可以同时加入几个 RTP 会话中。在一个 multimedia session 中，除非编码将多个媒体编入单数据流中，否则每个媒介都会独立的在自己的 RTP 会话中传送自己的 RTCP 报文。参与者利用不同的目的 transport addresses 对(pair)来区分就收到的不同的 RTP 会话，其中 transport addresses 对包括一个网络地址和一对 RTP 和 RTCP 端口。在一个 RTP 会话中的所有参与者分享同一个目的 transport address 对，如同 IP 多播，而不像单波网络中每个参与者的地址和端口对各不相同。在单播的情况，参与者可能与其他者共享同一对端口或者每两个人用一对端口。识别每一个 RTP 会话是利用两个 RTP 中的 SSRC identifier(定义见下面)空隙。在一个 RTP 会话中的一组参与者都有具有接收任何参与者的 RTP 中的 SSRC identifier 或是 RTCP 中的 CSRC(定义见下面)。例如，考虑三方会议的情况，每两个人都用一对不同的单播 UDP 端口。如果其中的一位与会者给仅另一个与会

者发送 RTCP 反馈，这样会议就由三的独立的点到点的(point-to-point)RTP 会话组成。如果其中的一个与会者将他接收到的另一个与会者的数据所做的 RTCP 反馈给另外两个人，这时会议就是由一个多方 RTP 会话组成。后面的例子模拟了在 IP 多播通信时三方会议会发生的情况。

RTP 协议框架在这里定义一些改变，但是某种特定协议的或是应用设计会限制这些定义的变化。

Synchronization source (SSRC)：RTP 报文流的一个 source，由 RTP 头中定义的 32-bit 的 SSRC identifier 来标识，这样做是为了不依赖网络地址。所有从同一个同步源出来的报文都具有相同的时序和序列号间隔，这就使得一个接收组可以依靠这些同步源来进行重放。同步源可以是报文流的发送者，这些报文是从麦克风或是相机得到的信号源或是一个 RTP mixer(见下)。synchronization source 可能会改变数据的格式，例如，音频编码。SSRC identifier 在特定的 RTP 会话中必须是全局的随机值(见第 8 章)。参议这不必在多媒体会话中为了所有 RTP 会话使用同一个 SSRC identifier；SSRC identifiers 是由 RTCP 分配的(见第 6.5.1 节)。在一个会话中，如果参与者中从不同的视频源端生成多个流，那么每个必须标识为不同的 SSRC。

Contributing source (CSRC)：RTP 报文的一个 source，对由 RTP Mixer 所输出的混合信号有作用(见下)。Mixer 在 RTP 头中插入一系列的 SSRC identifier，用来生成某种特定的报文。这一系列的就叫 CSRC list。如音频会议中，Mixer 标识出输出的报文由哪个讲话者的构成，这就使得就收这可以知道现在讲话者是谁，即使所有的数据包都包含相同的 SSRC identifier(that of the mixer)。

End system：一种应用，产生或是接收(consume)在 RTP 报文中传送的目录。在特定的 RTP 会话中，一个 end system 可以充当一个或者几个 synchronization source，但是一般情况下是一个。

Mixer：一个中介系统，从一个或几个源端接收报文，用某种方式合成然后输出新的 RTP 报文，这中间可能会改变原来报文的格式。不同输入源端来的数据时序可能不同，Mixer 会做一些调整产生自己的时序。所有从 Mixer 输出的数据包将会标识 Mixer 作为它们的同步源。

Translator：一个中介系统，前向输出 RTP 报文。translator 包括没有经过 mixing 的转换编码设备，从多播到单播的复制品(replicators)，和应用级的防火墙滤波器。

Monitor：在 RTP 会话中接收由发送者发送的 RTCP 报文的应用，特别是接收报告，为分发监测估计当前的 QOS，错误诊断和长期的统计。monitor 功能可以集成进参加会话的应用中，但是也可以是独立的不另外参加或发送接收 RTP 数据报文的应用(因为它们在独立的端口)。这叫做第三方的 monitor。也有这样的情况，在会话中第三方的 monitor 接收 RTP 数据报文但是不发送 RTCP 报文或是其它的被希望的。

Non-RTP means：为了提供一个适用的服务而额外附加的协议或机制。特别是在多媒体会议中，需要一种控制协议来分发多播地址和密钥来加密，协调所用的加密算法，在 **RTP payload type** 值与 **payload** 格式之间定义动态的映射，这样的例子包括会话初始协议**(SIP)(RFC 3261 [13])**， **ITU** 建议的 **H.323 [14]**和应用 **SDP** 的应用**(RFC 2327 [15])**，例如 **RTSP(RFC 2326[16])**。对于一个简单的应用，可以应用电子邮件或是会议数据数据库。更详细的这样的协议或机制不在本文档之内。

4. 字节序，校正(Alignment)，时间格式

网络字节序携带所有的整数域，即，有用的字节为首。在[3]有更详细的传输的顺序描述。不像其它的常数是十进制的。

所有的头数据都被指派它的自然长度(natural length)，即，16-bit 域指派两个偏移(offset),32-bit 域所指派的必须能被 4 整除等。用来填充的字节取值为零。

Wallclock time (绝对日期和时间)是用网络时间协议(NTP)的时间戳格式来标识的，秒级的与 0h UTC on 1 January 1900 [4]有关。NTP 的时间戳使用 64-bit 无符号的定点表示，整数部分用前 32-bit，小数部分用后 32-bit 来表示。在一些情况下，可以用一种更简洁的表示法，即用中间的 32-bit，也就是说，低 16 位表示整数，高 16 位表示小数，高 16 位的整数位必须独立确定。

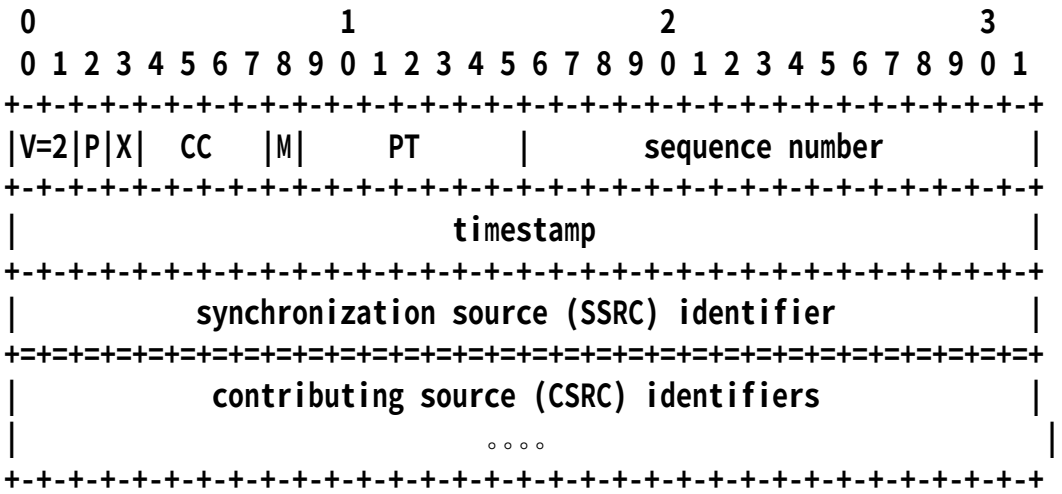
运行网络时间协议不需要其他的协议。也可以用其它的时间源或者根本不需要。(见第 6.4.1 节的 NTP 时间戳域详解)。运行 NTP 对来自不同的主机的同步流有益。

NTP 时间戳到 2036 年就会循环回零的状态，但是对于 RTP 来说，仅仅是利用了 NTP 时间戳的不同取值。只要时间戳对之间在 68 年内，用模块化的构架来减或比较就不用考虑 NTP 循环的情况。

5. RTP 数据传输协议

5.1 RTP 定长的头字段

RTP 头格式如下：



每一个 RTP 包中都有前 12 个字节，而 CSRC identifier 列表仅在 Mixer 插入时才有。域解释如下：

version (V): 2 bits

RTP 版本标识。说明中定义了两个版本(第一个 RTP 草案版本用 1 标识，在 "vat" 音频工具中最初是用 0 来标识的)。

padding (P): 1 bit

如果设定 padding，在报文的末端就会包含一个或者多个 padding 字节，这不属于 payload。最后一个字节的 padding 有一个计数器，标识需要忽略多少个 padding 字节(包括自己)。一些加密算法可能需要固定块长度的 padding，或者是为了在更低层数据单元中携带一些 RTP 报文。

extension (X): 1 bit

如果设定了 extension 位，定长头字段后面会有一个头扩展，在第 5。3。1 节中定义了格式。

CSRC count (CC): 4 bits

CSRC count 标识了定长头字段中包含的 CSRC identifier 的数量。

marker (M): 1 bit

marker 是由一个 profile 定义的。用来允许标识在像报文流中界定帧界等的事件。一个 profile 可能定义了附加的标识位或者通过修改 payload type 域

中的位数量来指定没有标识位(见第 5.3 节)。

payload type (PT)：7 bits

这个字段定一个 RTP payload 的格式和在应用中定义解释。profile 可能指定一个从 payload type 码字到 payload format 的默认静态映射。也可以通过 non-RTP 方法来定义附加的 payload type 码字(见第 3 章)。在 RFC 3551[1]中定义了一系列的默认音视频映射。一个 RTP 源有可能在会话中改变 payload type，但是这个域在复用独立的媒体时是不同的。(见 5.2 节)。

接收者必须忽略它不识别的 payload type。

sequence number：16 bits

每发送一个 RTP 数据报文序列号值加一，接收者也可用来检测丢失的包或者重建报文序列。初始的值是随机的，这样就使得 known-plaintext 攻击更加困难，即使源并没有加密(见 9。1)，因为要通过的 translator 会做这些事情。关于选择随机数方面的技术见[17]。

timestamp：32 bits

timestamp 反映的是 RTP 数据报文中的第一个字段的采样时刻的时间瞬时值。采样时间值必须是从恒定的和线性的时间中得到以便于同步和 jitter 计算(见第 6.4.1 节)。必须保证同步和测量保温 jitter 到来所需要的时间精度(一帧一个 tick 一般情况下是不够的)。 时钟频率是与 payload 所携带的数据格式有关的，在 profile 中静态的定义或是在定义格式的 payload format 中，或通过 non-RTP 方法所定义的 payload format 中动态的定义。如果 RTP 报文周期的生成，就采用虚拟的(nominal)采样时钟而不是从系统时钟读数。例如，在固定比特率的音频中，timestamp 时钟会在每个采样周期时加一。如果音频应用中从输入设备中读入 160 个采样周期的块，the timestamp 就会每一块增加 160，而不管块是否传输了或是丢弃了。

对于序列号来说，timestamp 初始值是随机的。只要它们是同时(逻辑上)同时生成的，这些连续的的 RTP 报文就会有相同的 timestamp，例如，同属一个视频帧。正像在 MPEG 中内插视频帧一样，连续的但不是按顺序发送的 RTP 报文可能含有相同的 timestamp。

不同媒体流的 RTP timestamp 可能有不同的速率，通常有独立的随机 offset。虽然这些 timestamp 对于从建单流的时序是足够的，但是直接比较不同 timestamp 来获得同步是不够的。因此，对于每一个媒介来说，RTP timestamp 与采样时刻是通过与从参考时钟而来的 timestamp 作对比而建立时间戳对(pair)，系统时钟 timestamp 是数据数据被采样的时刻的 timestamp。系统时钟被所有媒体共享以便同步。timestamp 对不是在每一个数据报文中传送，而是在低速率的 RTCP SR 报文中传送见第 6.4 节。

采样瞬时值作为 RTP timestamp 是因为这对于传输的终点来说是可知的并且它对于所有媒体，独立的编码延迟，或者其它的处理来说是一个通用的定义。目的是为了允许所有的同时采样的媒体能够同步表示

(presentation)。

应用所传输的数据是已经存储的而不是实时采样的，利用一个虚拟的从 wallclock 得到的表示时间来确定下一帧或是其它的存储单元的表示时间。在这种情况下，RTP timestamp 会为每个单元提供表示时间。即每一个单元的 RTP timestamp 会根据 wallclock time 来确定虚拟的表示时间。就收者的真正的再现会发生在一些时间后。

事先录音的现场音频会议很好的说明了选择采样瞬时值作为参考点的的重要作用。在这个假定中，视频必须被同步利用 RTP 传送以便于讲述者”看”。RTP 传送的一帧视频的"采样瞬时值"是由参考自己的时间戳与 wallclock 时间来确定什么时刻再现给讲述者。对于包含讲述者语音的音频 RTP 报文的采样瞬时值来说，是由参考音频采样时刻的 wallclock 时刻确定的。如果两台主机是由 NTP 来同步的，也可以通过这两台的主机传送音视频。就收者可以利用 在 RTCP Sr 报文中的 timestamp 对中的 RTP timestamp 来重现音视频。

SSRC：32 bits

SSRC 域识别同步源。为了防止在一个会话中有相同的同步源有相同的 SSRC identifier， 这个 identifier 必须随机选取。生成随机 identifier 的算法见目录 A.6。虽然选择相同的 identifier 概率很小，但是所有的 RTP implementation 必须检测和解决冲突。第 8 章描述了冲突的概率和解决机制和 RTP 级的检测机制，根据唯一的 SSRC identifier 前向循环。如果有源改变了它的源传输地址，就必须为它选择一个新的 SSRC identifier 来避免被识别为循环过的源(见第 8.2 节)。

CSRC list：0 to 15 items，32 bits each

CSRC list 表示那些在本报文中对 payload 作了贡献的源。号数是由 CC 域定的。如果有多于 15 个贡献源，只有 15 个源可以被标识。CSRC identifier 是由 Mixer 利用贡献源的 SSRC identifier 插入的。例如，对于音频报文，所有混合在一起的源的 SSRC identifier 被例出来，以便就收者识别出正确的讲话者。

5.2 RTP 会话的多路复用

正如在 the integrated layer processing design principle [10]中描述的那样，为了协议处理更有效率，复用点的数量应最小化，在 RTP 中，复用技术是由目的传输地址(网络地址和端口号)提供的。例如，在音频与视频独立编码的远程电信会议(teleconference)中，每一个媒介都是由单独的带有自己的目的传输地址的 RTP 会话来携带的。

分开的音频和视频流不应该在一个 RTP 会话中携带也不应该基于 payload type 或 SSRC 域来解复用(是 should not be carried in a single RTP session and demultiplexed based on the payload type or SSRC fields)。不同的 RTP

媒体但是用同样的 SSRC 较差的报文会产生几种问题：

1。 假设，两个音频流公用相同的 RTP 会话，具有相同的 SSRC 值，其中的一个改变了编码因此需要一个不同的 RTP payload type，就是不能识别出到底是哪个流改变了编码。

2。 SSRC 是用来识别一个单时序和序列值间隙的。如果媒体时钟速率不同，交叉的多 payload type 需要不同的时隙和不同的序列间隙来通知哪个 payload type 承受了损失。

3。 RTCP 发送者和接受者报告在每个 SSRC 仅能描述单时序和序列号间隙，没有携带 payload type 域。

4。 RTP Mixer 不能将不兼容的交叉 媒体流合并成单一流。

5。 在一个 RTP 会话中携带多个媒体流就不会出现下面几种情况：利用不同的网络路径或者分配的网络资源；希望接收子媒体，例如视频超出了带宽的承受能力而至接收音频，接受方对不同的媒体进行不同的处理，而用独立的 RTP 会话就可以执行单或多媒体处理。

每一个媒体用一个 SSRC 但是在一个 RTP 会话中发送它们可以避免上述的前三种情况，但是不能解决 后两种情况。

另一方面，在一个 RTP 会话中具有不同 SSRC 值的同一媒体的相关源的多路复用对多播会话来说是很标准的。上述问题就不会出现：例如，RTP Mixe 可以合并多个媒体音频源，对它们进行同样的处理。这对具有不同 SSRC 值得相同媒体的复用也是很合适的，这样就不存在后两个问题了。

5.3 RTP 头的 Profile-Specific 修正

现有的 RTP 数据报文头对一般的 RTP 所支持的应用来说已经是很完善的了。然而为了与 ALF 设计原理保持一致，头字段可以修剪或是根据附加条款的定义而增补，仍旧允许监测和记录。

- marker bit 和 payload type 域携带 profile-specific 信息，但是它们是在定长头字段中分配的，因为很多应用需要它们，不然就的额外增加 32-bit。包含这些域的字段可以根据需要而作出修改，例如增或减 marker bit。如果有很多的 marker bit，应该关注那些最重要的位，因为 profile-independent 监测就可以观测报文丢失和 marker bit 的关系了。

- 特定 payload format 所需要的附加信息，例如视频编码应该在有效载荷段携带。这可能会是在经常出现在有效载荷段的开始的头部或是由数据类型的保留值来标识。

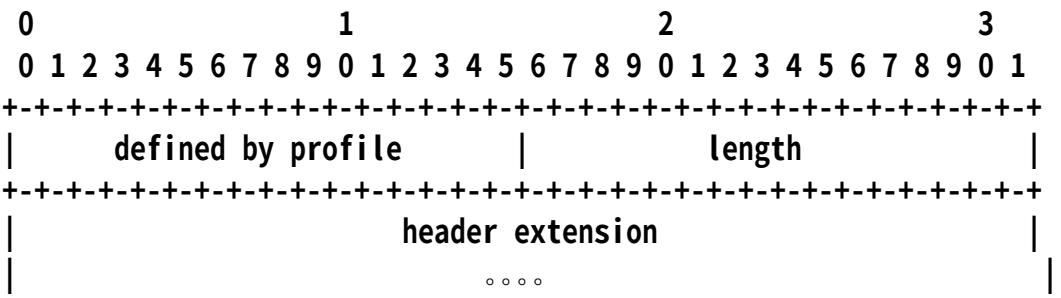
o 如有应用需要与 payload format 无关的附加功能的话，在应用下面运行的 profile 就应该定义固定的域来追踪 SSRC 域。应用可以直接访问附加域，而 profile-independent monitors 或是记录者仍旧可以通过前 12 个字节来识别 RTP 报文和进行相应的处理。

如果是所有 profile 都需要的附加功能的话，那就需要另外定义一个 RTP 版本了。

5.3.1 RTP 头字段扩展

新的独立的有效载荷格式需要在 RTP 数据报文头中携带附加的信息，这就需要一种扩展的机制。设计这种机制的目的就是为了那些没有进行头字段扩展的 implementation 可以忽略这些。

需要注意的是这种头字段的扩展仅仅是为了有限的应用。更具有潜在价值的做法是用前面描述的方法。例如对定长头字段的 profile-specific 扩展代价更小因为它不需要条件或是在变长的区域（location）。特定 payload format 需要的附加信息不应用这个头扩展而是携带在报文有效载荷区。



如果 RTP 头中的 X bit 是 1 的话，就必须在 RTP 头附加上变长的头扩展，如果有 CSRC list 的话就是在它后面。头扩展中包含一个 16-bit 的长度域用来对扩展中的 32-bit 字个数计数，把 four-octet 的头扩展（有效长度是零）排出在外。只能在 RTP 数据头中附加一个扩展。为了允许具有不同头扩展的交互操作和为了允许多于一个头扩展的特定操作（multiple interoperating implementations to each experiment independently with different header extensions, or to allow a particular implementation to experiment with more than one type of header extension），头扩展的前 16 位是用来鉴别 identifier 或是参数的。格式是由执行操作下面的 profile 来定义的。RTP 不定义任何的头扩展。

6. RTP 控制协议--RTCP

RTP 控制协议(RTCP)是基于在会话中的对所有参与者周期传输的控制报文的，与数据报文使用相同的分发机制。协议必须提供对数据报文和控制报文的复用，例如用 UDP 的不同端口号。RTCP 有下面四个功能：

1。 最基本的功能是对提供分发数据的质量反馈。这是 RTP 作为一个完整的传输协议所必须的部分，与其它的提供流或拥塞控制的传输协议有关（见第 10 章中关于拥塞控制需求分析）。反馈对控制自适应编码有直接的影响[18，19]。进一步的 IP 多播实验表明它对从接收者的反馈的作出分发错误诊断也很重要。将就受报告反馈给所有人就使得关注问题的人知道问题时本地的或是全局的。利用像 IP 多播的分发机制，可以使像服务提供者这样的实体(entity)收到反馈和作为第三方监测诊断网络问题的所在。这种反馈的功能有 RTCP sender 和 receiver report 实现，见第 6.4 节。

规范的

2。 RTCP 为 RTP source 携带一个稳定的传输级的 identifier，叫 canonical name 或 CNAME，
见 6.5.1。因为 SSRC identifier 可能因为发现了冲突或是程序重起而改变，接受者需要 CNAME 追踪每一个参与者。接受者也可以用 CNAME 来协调从一个参与者而来的一系列相关 RTP 会话的数据流，例如同步音频和视频。Inter-media 同步需要 NTP 和 RTP timestamps 包含在发送者发送的 RTCP 报文中。

3。 前两个功能需要所有参与者都发送 RTCP 报文，因此必须控制速率使得 RTP 可以让更多的参与者加入。每个人都对所有人发送他的控制报文，这就使得人人都可以独立的观察参与者的数量。这个数量用来计算发包的速率见第 6。2 节。

4。 这个功能是可选的，传达最小的会话控制信息，例如在用户界面显示参与者的身份。这最有可能用在 "loosely controlled"会话中，因为参与者不需要身份控制或是参数传递就可以进入或退出。RTCP 扮演一个直达接受者的便利通道的角色，但是不需要支持所有应用需要的控制通信的全部要求，需要一种更高一级的会话控制协议，这不在本文当讨论范围之内。

1-3 功能应用在所有情况中，特别是在 IP 多播的情况。RTP 应用设计者应该避免那些只能工作在单播模式不能提供更多的数量的机制。接受者和发送者各自控制 RTCP 的传输。见第 6.2 节，例如单向链路不可能从接受者那里收到反馈。

非标准的注释：在一种叫 Source-Specific Multicast(SSM)的多播路由方法中，一个"channel"只有一个发送者(一个源地址，组地址对)，接收者不能与其他的 channel 成员(except for the channel source)利用多播直接通信。本建议仅通过第 6。2 提供的关闭接收者的 RTCP 来调节 SSM。将来会修改文档可以支持接收者

6.1 RTCP 报文格式

为了携带不同的控制信息，本说明定义了不同的 RTCP 报文：

SR： 发送者报告，从参与者来的关于活动的(active)发送者的传输和接收统计

RR： 接收者报告，从参与者来的关于活动的(active)发送者的接收统计和与 SR 中关于活动的发送者的超过 31 个源的报告结合。

SDES: 源描述条款，包括 CNAME

BYE： 参与者结束标识

APP： Application-specific 功能

RTCP 报文与 RTP 数据报文一样都有一个固定的段， 接着的是根据报文类型可变长的数据结构，但是都是由 32-bit 的边界来结尾的。包含 Alignment 和长度域使得 RTCP 报文"stackable"。多个 RTCP 报文可以不经分离器(separator)干涉就可以极连形成可以被低层协议发送的单 RTCP 报文，例如 UDP。混合报文中不会有明确的个体 RTCP 报文的计数，因为低层的协议希望提供一个全长来确定混合报文的末端。

每一个在混合报文中的个体 RTCP 报文都可不必按顺序或是报文结合就可单独处理。然而，为了协议的功能，必须作出一些限制：

o 接受统计(在 SR 或 RR 中)依照带宽限制所做的统计最大化而派发出去，因此，每一个周期传送的混合报文必须包含一个报告报文。

o 新的接收者需要 CNAME 尽快地识别源来协调媒体进行如唇同步等，除了为了部分加密外（见 9.1），每一个混合的 RTCP 报文必须包含 SDES CNAME 。

o 第一次出现在混合报文中的报文类型号必须被限定为增加第一字节的，(to increase the number of constant bits in the first word and the probability of successfully validating RTCP packets against misaddressed RTP data packets or other unrelated packets)。

因此，所有的 RTCP 报文必须包含至少两个个体的报文，它们具有如下格式：

Encryption prefix：除非混合报文是通过第 9.1 节的方法加密的，否则必须被一个随机的 32-bit 量来重新赋值edrawn)。如果加密需要填充，它必须被填充到混合报文的最后一个报文中。

SR or RR：如附录 A.2 中描述的，混合报文中的第一个 RTCP 报文必须是一个报

告报文以便于确认头。即使没有数据发送或者发送，也必须是这样的。这种情况下必须发送空的 RR 即使其它的 RTCP 报文是 BYE。

Additional RRs：如果接受统计的源数超过了 31，多余的 RR 报文 follow the initial report packet。

SDES：每一个混合 RTCP 都包含带有 CNAME 项 的 SDES 报文，排除第 9.1 节所述的。其它的源描述项 视特定应用根据带宽限制而定（见第 6.3.9 节）。

BYE or APP：其它的 RTCP 报文类型，包含尚未定义的，没有固定的顺序，只是 BYE 是由 SSRC/CSRC 传送的最后一个字节。报文类型可能不只一次出现。

除非混合的 RTCP 报文如第 9.1 节描述那样是为了部分编码而分片的，个体 RTP 参与者在每一个报告间隔内只能发送一个混合的 RTCP 报文，这样做是为了每一个参与者估计的 RTCP 带宽是准确的（见 6.2）。如果提供的 RR 不够更多的源而又不能超过网络路径所能传输的最大的单元(MTU)，那么只有适合 MTU 的子序列可以被包含进每个间隔内。通过对多间隔的循环选择子序列，这样所有的源都可以被报告。

建议 Translator 和 Mixer 合并 individual RTCP packets from the multiple sources they are forwarding into one compound packet whenever feasible in order to amortize the packet overhead(见第 7 章)。由 Mixer 生成的 RTCP 混合报文的一种可能情况见 Fig。1。如果混合的报文总长超过了网络路径的 MTU，它将被分成更小的混合报文来分开传输。这并没有削弱 RTCP 带宽的估值，因为每一个混合的报文至少代表了一个参与者。注意每一混合报文都是由 SR 或 RR 报文开始的。

implementation 忽略它不识别的 RTCP 报文类型。附加的 RTCP 报文类型需要向 the Internet Assigned Numbers Authority (IANA)注册（见第 15 章）。

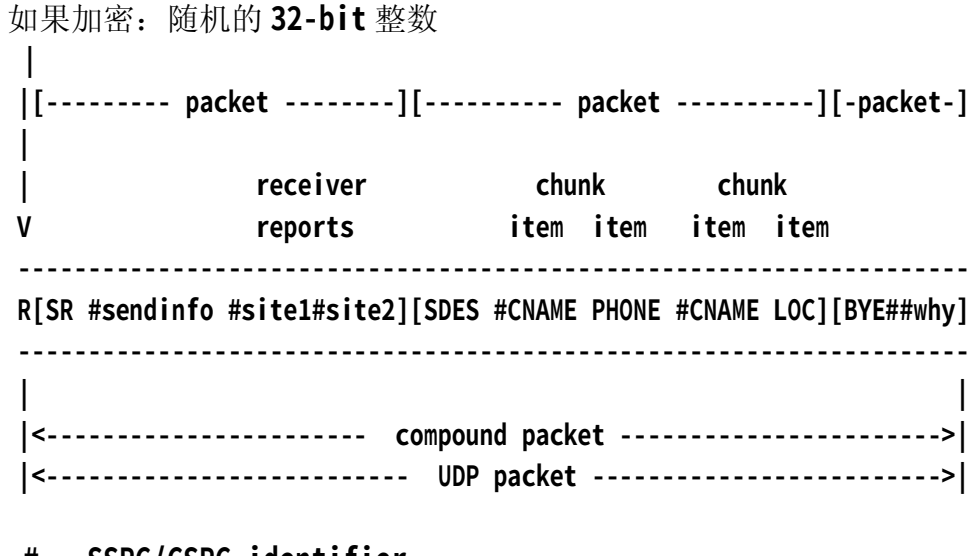


Figure 1：RTCP 混合报文的一个例子

6.2 RTCP 传输间隔

RTP 允许应用根据会话来从几个参与者到上千个自动调节。例如，在音频会议中数据通信量(data traffic)很自然的自我限定因为某个时间只有一个或两个讲话者，所以多播分发的情况下，每一个链路的数据速率相对的与参与者数量是不相关的。然而控制通信量(control traffic)并不是自我调节。如果从每一位参与者来的接收报告以恒定速率发送，那么控制通信量就会与参加者的数量成线性关系。因此必须动态的根据 RTCP 报文之间的间隔来调节速率。

对于每个会话来说，通常是假定数据通信量隶属于被参与者分的“会话带宽”。这个带宽是预定的和被网络限定的。如果没有预定，根据环境会有其它的限制确定“合理的”最大量给会话用。那就是会话带宽。给予代价或是对会话可提供的网络带宽的先验知识来选择会话带宽。与媒体编码关系很少，但是会话带宽会限制编码选择。通常，会话带宽是期望并发的发送者的虚拟(nominal bandwidth)带宽的总和。对于 teleconference 音频，这个值是一个发送者的带宽。对于层编码，每一个层对应一个具有自己会话带宽参数的 RTP 会话。

当调用了一个媒体应用时，会话带宽参数期望是由会话管理应用来提供的。但是媒体应用会根据单发送者数据带宽发送一个默认值以便于会话的编码。应用也可以根据多播范围规则或是其它的标准来强迫带宽限定。所有的参与者必须使用同样的会话带宽值以便于计算相同的 RTCP 间隔。

对于控制和数据通信量的带宽计算包括低层的传输和网络协议（例如 UDP 和 IP）。因为那是资源预留系统需要知道的。同时也希望应用知道在使用哪些协议。链路级的头不在计算内，因为当传输时，不同的链路级的头会封装进报文。

限定 control traffic 为会话带宽的一小部分(small)并且是可知的：一小部分是因为主要的携带数据的传输协议功能就不会被削弱；可知是因为这样可以在给定的资源预留协议的带宽说明中包含进 control traffic，而且参与者可以独立计算自己的份额。The control traffic bandwidth is in addition to the session bandwidth for the data traffic。建议为 RTCP 所增加的会话带宽部分固定在 5%，1/4 的 RTCP 带宽分给发送数据的参与者，这样在有很多接收者却只有很少的发送者的会话中，新加进来的参与者就可以更快的接收到 CNAME。当发送者的比例超过 1/4 时，发送者会按他们的比例得到 RTCP 带宽。While the values of these and other constants in the interval calculation are not critical, all participants in the session MUST use the same values so the same interval will be calculated。 Therefore, these constants SHOULD be fixed for a particular profile。

profile 会说明 control traffic 带宽会是会话的一个单独参数而不是严格的带宽百分比。单独的参数能使速率自适应应用设定与“典型的”数据带宽一致的 RTCP 带宽，数据带宽是低于会话带宽参数指定的最大带宽。

profile 会进一步的指定将 control traffic 带宽分成两部分独立的会话参数，分配给那些活跃(active)的数据发送者和不活跃的(可能有误)；假定为 S 和 R。按照建议 RTCP 带宽的 1/4 分给数据发送者，
默认的两个参数就是 1.25%和 3.75%。当发送者的比例大于 S/(S+R)时，发送者就会按比例得到他们的份额。利用这两个参数可以为特定应用关闭 RTCP 接收报告，将非数据发送者的 RTCP 带宽设为零而将数据发送者的 RTCP 带宽设为非零，这样仍可为 inter-media 同步而发送发送者报告。并不推荐关闭 RTCP 接受报告因为在第 6 章开始所列的功能里是需要的，特别是接收质量反馈和拥塞控制。然而，对于运行在单向链路的系统或是不需要接收质量反馈的会话或是活动的接收者而言这样做是合适的，因为它们有其它的方法避免拥塞。

当参与者的数量很小和 traffic 没有按照大数理论来平滑，混合 RTCP 报文之间的间隔就应该有一个低的 bound 来避免报文雪崩。It also keeps the report interval from becoming too small during transient outages like a network partition such that adaptation is delayed when the partition heals。 在应用启动时，应该在第一个混合 RTCP 报文之前先加一段延迟来接收其它参与者的 RTCP 报文，这样报告间隔就会更快的收敛到正确值。这个延迟设定为最小间隔的一半，这样能对新加入者做出更快的反应。建议的固定的最小间隔是 5 秒。

Implementation 会调节最小 RTCP 间隔为一个更小的值，与会话带宽比例相反，有如下的限制：

o 对于多播会话，只有活跃的数据发送者可以使用简化的值来计算混合 RTCP 报文的间隔。

o 对于多播会话，非活跃的数据发送者也可使用简化的值，发初始混合报文之前的延迟可以为零。

o 对于所有的会话，计算参与者 timeout 间隔（见第 6。3。5 节）时应用固定的最小值，这样那些没有简化值得 implementation 就不会过早的 timed out。

o 建议的简化最小值为 360 除以会话带宽（kilobits/second） 秒，当带宽大于 72kb/s 时，这个值就会比 5 秒小。

第 6.3 节和附录 A.7 中描述的算法就是为了满足本节所述的目标而设计的。它计算混合 RTCP 报文间隔来分配允许的 control traffic 带宽。这就允许应用可以对小会话做出反应，例如，对所有参与者的身份鉴定是很重要的，还要自动的适应大的会话。算法具有下面的几个特性：

o 计算的 RTCP 报文间隔会依据组成员的数量线形的调节。这个线形的因素使得对所有者的 control traffic 常量设定成为可能。

o RTCP 报文间隔在[0.5，1.5]是随机取值的乘以计算的间隔来避免 unintended 同步[20]。第一个 RTCP 报文也被延迟一个随机值，这个值是在半个最小 RTCP 间隔内取值。

o 计算平均混和 RTCP 报文的长度的动态估值，包括发送的和接收的报文，与携带的控制信息量自动适应。

o 因为间隔的计算会参考组成员的数量，当一个使用者加入一个已有的会话或是许多使用者同时加入一个新会话时就会有不期望的重启影响。这些新加入者因为没有正确估计组成员数量，所以设定了一个很小的 RTCP 传输间隔。如果很多的人同时加入，这个问题会更严重，因此为了解决这个问题，设计了一种叫做“timer reconsideration”的算法，算法运行一个简单的回退(back-off)机制，当组成员增加时，它会阻止 RTCP 报文的传送。

o 当使用者因为 BYE 或是 timeout 离开会话时，组成员减少了，计算的间隔也会减少，用一种称为”reverse reconsideration”的算法来允许成员迅速的减少他们的间隔。

o BYE 报文相比其他 RTCP 报文来说会有不同的处理，当使用者离开时，会在下一个预定的 RTCP 报文中发送 BYE 报文。传输 BYE 遵循回退算法来避免大量的成员同时离开而造成 BYE 雪崩。

这种算法可用于所有参与者都允许发送的会话中。在这种情况下，会话带宽参数是单个发送者带宽乘以成员数量，RTCP 带宽是 5%。

算法的详情见后续章节，目录 A。7 中举了一个例子。

6.2.1 Maintaining 会话成员数量

计算 RTCP 报文取决于参加会话的参与者站点(site)的数量估计。当侦听到加入的站点时，就会在计数上加一，而且为每个站点创建一个由 SSRC 或 CSRC identifier(见第 8.2 节)的表来跟踪。直到收到携带新的 SSRC 的多报文时才认为新的实体有效（见附录 A。1），或者直到收到包含 CNAME 的 SDES RTCP 报文，当收到 RTCP BYE 报文与相应的 SSRC identifier 时就会从表中删掉实体。也有这样的情况，就是某些报文在 BYE 之后姗姗而来，这将造成实体被重建。实体会标记为已经接收了 BYE 然后在延迟一定时间后被删除。

如果几个 RTCP 报告间隔（建议是 5）没有收到 RTP 或 RTCP 报文，参与者可以标记另一个站点为非活跃的或尚未有效时删除它。这对报文损失由一定的用处。所有的站点有有相同的乘法器值，计算 RTCP 报文间隔为大概的相同值，以使 timeout 正常的工作。因此，特定的 profile 会有固定的乘法器。

对于有很多参与者的会话，建立一个存储所有人的 SSRC identifier 和状态信息的表示不现实的。 implementation 可以像[21]中描述的那样使用 SSRC 采样来减少存储需求或是其它的算法来达到要求。关键是不能低估组大小虽然可能高估。

6.3 RTCP 报文发送和接收规则

这里描述了如何发送 RTCP 报文和接收了 RTCP 报文之后怎么做等。可以在多播和多点单播环境下运行的 implementation 必须满足在第 6。2 节的要求。这样的 implementation 可能用在这里定义的算法来满足要求或是用其它相似性能的算法。限制带两方的单播情况下运行的 implementation 应该使用随机的 RTCP 传输间隔，避免相同环境下运行的非有益的多实体同步，但是可能忽略第 6.3.3 节，第 6.3.6 节和第 6.3.7 节的 "timer reconsideration"和 "reverse reconsideration"算法。

为了遵循这些规则，会话参与者必须坚持(maintain)几条状规定：

Tp：上次传输 RTCP 报文的时间；

tc：当前时间；

tn：下一个 RTCP 报文预定的传输时间；

pmembers：上次重计算的会话成员的估计数量；

members：最大的(most)当前估计的会话成员数量；

senders：最大的当前会话中的发送者的数量；

rtcp_bw：目标 RTCP 带宽，即所有会话中的成员用的带宽（B/s)。在应用启动时提供的"session bandwidth"参数的一部分。

we_sent：如果应用前两个(the 2nd previous)RTCP 报告发送了，设定标志为 true。

avg_rtcp_size：混合 RTCP 报文的平均长度(B)，单个参与者所有的收发 RTCP 报文。长度包括低层传输和网络协议头。（如 UDP 和 IP）见第 6。2 节。

initial：如果应用尚未发送 RTCP 报文，就设定标志为 true。

大多数规则在报文传输间用了"calculated interval"。这个间隔在以下章节中会将到。

6.3.1 计算 RTCP 传输间隔

为了保持伸缩性，会话中的单个参与者报文的平均间隔应该和组大小一致。这个间隔被称为 calculated interval。它是由合并上述几条状态而得到的。

calculated interval T 的确定是如下：

1。 如果发送者的数量小于等于成员的 25%，间隔是根据参与者是否是发送者来定(基于 we_sent)。如果参与者是发送者(we_sent true)，常量 C 被定为平均的 RTCP 报文长度(avg_rtcp_size)除以 RTCP 带宽的 25%(rtcp_bw)，常量 n 定为发送者的数量。如果 we_sent 不是 true，C 定为平均 RTCP 报文长度除以 RTCP 带宽的 75%。n 定为接收者的数量(members-senders)。如果发送者的值大于 25%，发送者和接收者一起处理。C 定为平均 RTCP 报文长度除以整个 RTCP 带宽，n 为成员的总数。RTP profile 可以说明 RTCP 带宽明确的由两个独立参数(称作 S 和 R)定义。这种情况下，25%的部分变成了 S/(S+R)和 75%部分变成了 R/(S+R)。如果 R 是零，发送者的百分比不会超过 S/(S+R)，必须避免分母为零的情况。

2。 如果参与者还没有发送 RTCP packet(初始变量为 true)，常量 Tmin 定为 2.5 秒，否则为 5 秒。

3。 计算的间隔 Td 在 Tmin 和 n*C 之间取最大值。

4。 计算的间隔 T 取值为：按在 0。5 和 1。5 之间的平均分布的值乘以 Td。

5。 T 然后除以 e-3/2=1。21828 来补偿因为 timer reconsideration 算法收敛的 RTCP 带宽值低于期望值所造成的影响。

这个程序(procedure)最总导致间隔是随机的统计平均会收敛在发送者 25%的 RTCP 带宽，剩下的给接收者。如果发送者大于 1/4，这个程序就会平分带宽。

6.3.2 初始化

刚加入会话时，参与者设定 tp，tc 为 0，发送者为 0，pmembers 为 1，成员为 1，we_sent 为 false，rtcp_bw 为指定的会话带宽的，initial 为 true，avg_rtcp_size 为第一个 RTCP 的可能值。T 过后计算，第一个报文与定为 tn = T。这意味着设定了传输定时，期满时间为 T。应用可以用享用的办法来执行这个定时。

参与者把自己的 SSRC 加入到成员表中。

6.3.3 接收 RTP 或是 Non-BYE RTCP 报文

当参与者的 SSRC 没有在成员表中而又收到他的 RTP 或 RTCP 报文时，就会将 SSRC 加入表中，参与者被判为有效后立刻更新成员值见第 6.2.1 节。也会对在有效的 RTP 中的 CSRC 进行同样的处理。

当参与者的 SSRC 没有在发送表中而又收到他的 RTP 报文时就会将 SSRC 计入表中更新发送者的值。

对每个接收到的混合 RTCP 报文，更新 avg_rtcp_size 值：

avg_rtcp_size = (1/16) * packet_size + (15/16) * avg_rtcp_size

其中 packet_size 是刚接收到的 RTCP 报文的大小。

6.3.4 接收 RTCP BYE 报文

除了第 6.3.7 节描述的情况：当发送了一个 RTCP BYE，如果接收的报文是 RTCP BYE 报文，SSRC 与成员表对比，如果在，从表中删除实体，成员值更新。SSRC 然后与发送表对比，如果在，从表中删除实体发送者更新。

此外，为了使 RTCP 报文的传输率更适应组成员的变化，当接收到 BYE 报文使成员值比 pmambers 少时，应该执行"reverse reconsideration"算法：

- 通过下面的公式来更新 tn：

  tn = tc + (members/pmembers) * (tn - tc)

- 通过下面的公式来更新 tp：

  tp = tc - (members/pmembers) * (tc - tp)。

- 下一个 RTCP 报文提前到 tn 来发送。

- 设置 pmembers 值等于成员值。

在很多参与者同时离开会话而还有以下部分没有离开的情况下，这种算法就会持续一段时间的认为组成员值为零。然而算法会很快恢复正确的估计值。这种情况很少见，所造成的影响也不大所以对这个问题的考虑是第二位的。

6.3.5 时间到的 SSRC

参与者应该在偶尔的间隔期内查看是否其它的参与者时间到。参与者计算确定的（没有随机因素）Td，即设定 we_sent 为 false。没有在 tc-Mtd（M 是 timeout 乘法器，默认值为 5）发送 RTP 或 RTCP 报文的会话成员都认为是时间到。也即意味着它的 SSRC 将被移出成员表然后更新成员值。发送表也有相似的操作。没有再 tc-2T（在上两个 RTCP 报告间隔）的时间内发送 RTP 报文的成员（在发送成员表中）将被移出发送成员表，然后成员更新。

有成员时间到，就会执行第 6.3.4 节中的 reverse reconsideration 算法。

参与者必须至少每个 RTCP 传输间隔都执行这样的检查。

6.3.6 传输定时时间到

当传输定时时间到，参与者执行下述的操作：

o 传输间隔 T，包含随机因素，是通过第 6.3.1 节介绍的方法计算的。

o 如果 tp + T 小于等于 tc，传输 RTCP 报文。tp 设定为 tc 值，T 的另一个值像上步那样计算，tn 设定为 tc + T 值。传输定时设定为 tn。如果 tp + T 大于 tc，tn 设定为 tp + T 值。不发送 RTCP 报文。传输定时设定为 tn。

o pmember 设定为成员值。

如果有 RTCP 报文传输，初始值设定为 FALSE。此外，avg_rtcp_size 的值要更新：

```
avg_rtcp_size = (1/16) * packet_size + (15/16) * avg_rtcp_size
```

其中 packet_size 是刚刚传输的 RTCP 报文的大小。

6.3.7 传输 BYE 报文

当参与者想离开会话时，发送一个 BYE 报文来通知其他的参与者，当离开的数量超过了 50 时，离开者必须执行下述算法，避免 BYE 报文雪崩。这种算法代替正常的成员变量来计数 BYE 报文：

o 参与者决定离开时，tp 重设为 tc，当前时间，成员和 pmembers 初始为 1，initial 设为 1，we_sent 设为 false，senders 为 0，avg_rtcp_size 设定为混合 BYE 报文的大小。T 重新计算。BYE 报文与定为 tn = tc + T 时间再发送。

o 每当接收到 BYE 报文时，不管成员是否存在于成员表，成员值都自加 1，如果使用了 SSRC 采样，就不管 BYE SSRC 是否包含在采样中。接收到其它的 RTCP 或 RTP 报文成员不自加，同样的 avg_rtcp_size 仅仅在接收到 BYE 报文时才更新。当 RTP 报文到时不更新 senders；仍为 0。

o 然后采用传输正规 RTCP 报文一样的规则来传输 BYE 报文。

这就使 BYE 报文立刻得到传输，还控制了总带宽的使用。在最坏的情况，这导致 RTCP 控制报文使用两倍多正常的带宽(10%) -- 5%的 non-BYE RTCP 报文和 5%的 BYE。

不想等到上述机制允许才发送 BYE 报文的参与者可以直接离开而不发送 BYE

报文。这个参与者终会因为 ie 时间到而被其他的成员移出。

当参与者决定离开时，如果组成员估值小于 50，参与者会立刻发送 BYE 报文。也可以选择执行 BYE backoff 算法。

在任何一种情况，从来不发送 RTP 或 RTCP 的 参与者不准 BYE 报文。

6.3.8 更新 we_sent

如果参与者最近发送 RTP 报文，变量 we_sent 为 true，其它情况为 false。采用与管理发送表上的其他参与者同样的机制。如果参与者发送 RTP 报文而 we_sent 为 false，它将自己加进发送表中并且设定 sets we_sent 为 true。应该执行第 6.3.4 节描述的 reverse reconsideration 算法来减少发送 SR 报文的延迟。每次发送另一个 RTP 报文时，那个报文的传输时间在表中记载，应用正常的发送 timeout 算法--如果在 tc-2T 时间内没有发送 RTP 报文，参与者将自己从发送表中移出，发送者计数减一，设定 we_sent 为 false。

6.3.9 分配源描述带宽

本段定义了几个源描述(SDES)条款来增加 CNAME 条款，例如 NAME（个人名字），EMAIL(email address)。也提供了一种方法来定义新的 application-specific RTCP 报文类型。应用在给这个附加的信心分配控制带宽时应该谨慎些，它会降低接收报告和发送 CNAME 的速率，这就削弱了协议的性能。建议将不超过分配给单个用户的 20%的 RTCP 带宽分给附加的信息。此外，并不需要在每个应用中包含所有的 SDES 条款。应根据效率来指定带宽的一小部分。建议转化成静态的报告间隔的百分比应该基于条款的典型长度，而不是动态的估计。

例如，设计仅发送 CNAME，NAME，EMAIL 的应用，NAME 将会有比 EMAIL 更高的优先级，因为 NAME 将不断的在用户界面显示，EMAIL 仅在需要时显示。在每个 RTCP 间隔，就会发送带有 CNAME 的 RR 报文和 SDES 报文。

对于运行在最小间隔的小型会话，均值会是 5 秒。每第三个间隔（15 秒），SDES 报文中会包含额外的条款。这有 7/8 的时间是 NAME 条款，第八个时间（2 分钟）会是 EMAIL 条款。

通过共同的 CNAME 而捆绑的 cross-application，当多个应用利用 cross-application 而一致的运行，例如在一个多媒体会议中，附加的 SDES 信息可以在一个 RTP 会话中发送。其他的会话仅仅携带 CNAME。这种方法可以应用在多个会话的层编码配置中。（见第 2.4 节）。

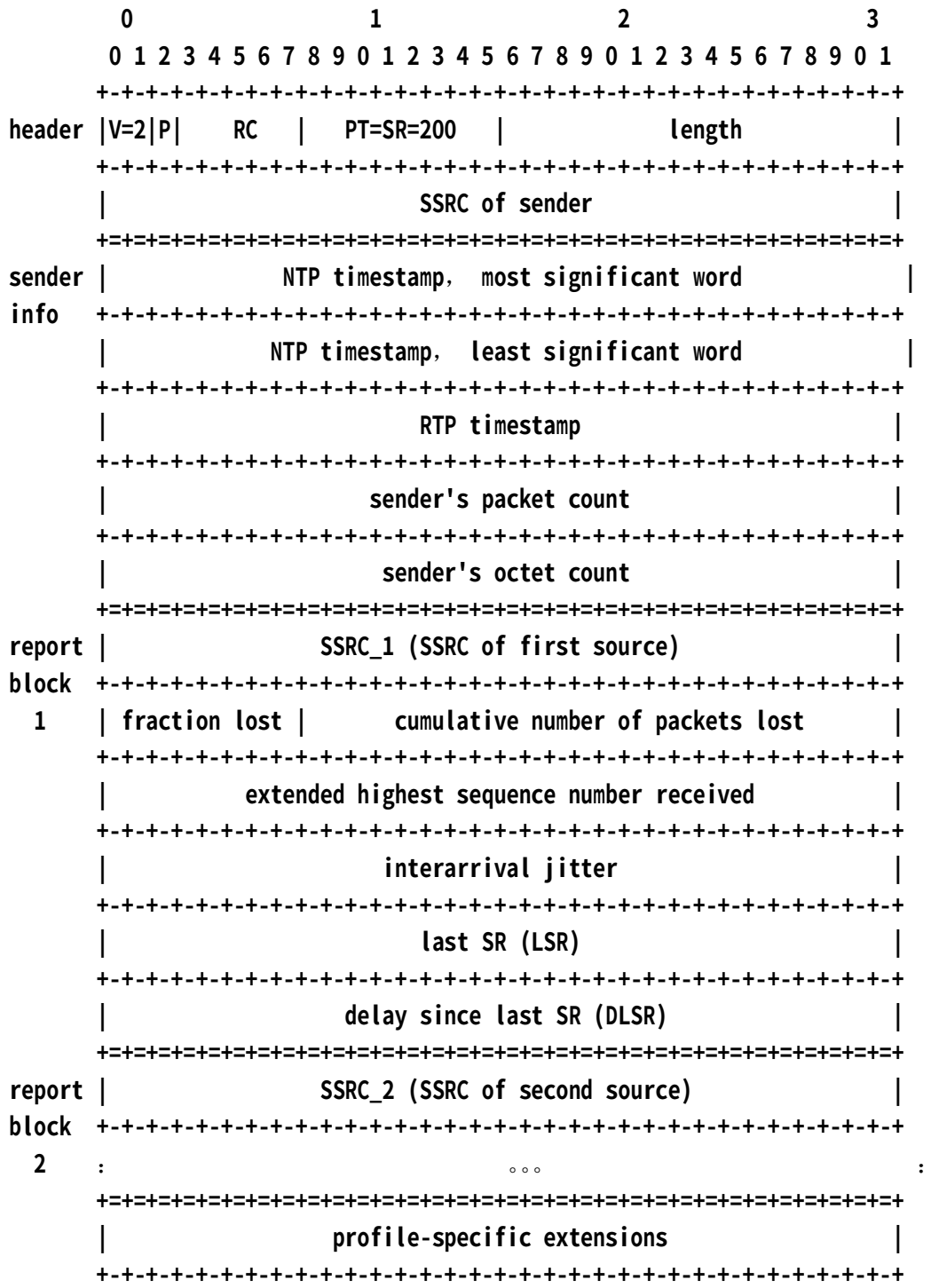
6.4 发送者和接收者报告

RTP 接收者利用 RTCP 报文来提供接收质量反馈，根据接收者是否也是发送者来采用两种格式的一种。除报文类型码外，发送者报告(SR)和接收者报告(RR)格式的唯一区别是，活跃的发送者的 SR 包含一个 20-byte 的发送者信息部分。自从最后一个报告或是前一个报告，如果节点在间隔期间发送任何的数据报文，都会发送 SR，其他的情况发送 RR。

SR 和 RR 格式都包含零个或更多的接收报告块，每一个块对应一个同步源，从那里接收者受到]]收到 RTP 数据报文。报告并不包括在 CSRC 表上的源。每一个接收报告块都提供关于本块中标记的特定源的接收报告的统计。因为在 SR 或 RR 中只有最大 31 个的接收报告块,在初始 SR 或 RR 来包含所有侦听到的源之后，其它的 RR 报文会被 stacked。如果一个混合 RTCP 报文不能装下所有必须的 RR 报文，那么仅仅适合一个 MTU 的子序列被包含进间隔内。子序列应该循环选择以使所有的源都可以被报告。

下面的部分定义了报告的格式，如果应用需要附加的反馈信息，它们怎么以 profile-specific 方式扩展，怎样用报告。关于 Translator 和 Mixer 接收报告详见第 7 章。

6.4.1 SR: Sender Report RTCP Packet



SR 报文由三个部分组成，如果定义了 profile-specific 扩展段，放在后面。第一个部分是 8 个字节长：

version (V): 2 bits

识别 RTP 版本，RTCP 报文也一样。本文档定义的是 2。

padding (P)：1 bit

如果设定了 **padding** 位，这个个别的 **RTCP** 报文在尾部包含一些附加的 **padding** 字段，不是控制信息但却包含在长度域中。**padding** 的最后一个字段是应该忽略的字段的计数，包括自己（四的倍数）。某些固定的块的加密算法需要 **padding**。在混合 **RTCP** 报文中，仅仅一个单独的报文需要 **padding**，因为混合的报文是如第 **9**。**1** 节所述的方法作为一个整体加密的。因此，仅需将 **padding** 加在最后一个单独的报文中，如果在那个报文中增加了 **padding**，**padding bit** 必须仅在那个报文中设置。这个规定协助头字段有效检查，见目录 **A.2**，允许检查那些没有设置正确的 **padding bit** 的前期 **implementation**，将 **padding** 加进左后的单独报文。

reception report count (RC)：5 bits

本报文中包含的接收报告块的数量，零有效。

packet type (PT)：8 bits

200 是标识本报文是 **RTCP SR** 报文。

length：16 bits

本 **RTCP** 报文长度（以 **32-bit** 形式）减一，包括头和 **padding**（减一是为了零长度有效和避免无限循环来寻找混合的 **RTCP** 报文，而 **32-bit** 字是避免对四的倍数的有效检查）。

SSRC：32 bits

对本 **SR** 报文的同步源标识。

第二个部分，发送者信息，是 **20** 字节，在每个 **SR** 报文中都存在。它是对从本发送者的数据传输的总结。

NTP timestamp：64 bits

当本报告发送时，标识 **wallclock time**（见第 **4** 部分），这样就可以和 **timestamp** 联起来使用 ，而 **timestamp** 是从接收报告中得到的，用来估量 **(measure)**那些接收者的 **round-trip** 传播。接收者期望 **timestamp** 的测量精度被限定为远小于 **NTP timestamp**。**timestamp** 测量的不确定性并不像标识因为可能不需要知道。在一个没有 **wallclock** 概念但有特定的 **system-specific** 时钟像 **"system uptime"**的系统，发送者可能用时钟作为参考来计算相关的 **NTP timestamp**。选择一个共同的时钟很重要，因为如果利用独立的 **implementation** 来生成单个多媒体会话流，所有的 **implementation** 都会用同样的时钟。在 **2036** 年之前，相对的或是绝对的 **timestamp** 在高位都是不同的，所以（有效的）对照会显示很大的区别；那时希望不需要相对的 **timestamp**。没有 **wallclock** 概念的发送者或是时间到**(elapsed time)**可以设定 **e NTP timestamp** 为零。

RTP timestamp：32 bits

与 **NTP timestamp** 保持一致**(**上**)**，在数据报文中与 **RTP timestamp** 有相同的单元和相同的随机偏移。这可对那些 **NTP timestamp** 同步的源进行 **intra-**和 **inter-**媒体同步，也可被媒体**-**独立接受这用来估计名义上的时钟频率。注意在

大多说情况下，在相邻的数据报文中，这个 timestamp 与 RTP timestamp 可能不相等。更合适的做法是，必须利用 RTP timestamp 计数器和真实时间的关系，从相应的 NTP timestamp 来计算，真实时间是通过周期的检查 wallclock 时间来确定的。

sender's packet count：32 bits

发送者传输的 RTP 数据报文的总数，从传输开始到本 SR 产生。如果发送者改变了 SSRC identifier，计数值重置位。

sender's octet count：32 bits

payload 字段的数量（即，不包括头或是填充）。如果发送者改变了 SSRC identifier，计数值重置位。这个域可用来估计平均 payload 数据速率。

第三部分包括根据发送者从上次报告侦听到的其它源的数量来确定零个或更多的接收报告块。每一个 接收报告块都含有从单个同步源的关于接收 RTP 报文的统计。当源因为冲突而改变了 SSRC identifier 时，接收者不携带统计。统计包括：

SSRC_n (source identifier)：32 bits

源的 SSRC identifier，本报告块的信息所属。

fraction lost：8 bits

上次 SR 或 RR 发送之后，从 SSRC_n 源的 RTP 报文丢失的部分，以定点数来表达，二进制的点在左边沿（相当于 loss 部分乘 256 之后取整）。这部分定义为丢失报文的数量除以期望的报文，像下段所应依的那样。implementation 见目录 A.3。如果因为复制而使损失为负，lost 部分设定为零。注意到接收者不能辨认是否有报文丢失，如果在上次报告间隔期间的报文丢失了，不会有没有接收报告来关注这个问题。

cumulative number of packets lost：24 bits

从接收开始，SSRC_n 源的 RTP 报文丢失的数量。定义为期望的减去实际接收的，其中接收的包括迟到的和复制的，如果有复制，丢失可能为负。期望的报文定义为由上次接收的序列号延伸出的序列号，如下所定义，减去接收到的初始序列号。目录 A.3 有详细算法。

extended highest sequence number received：32 bits

低 16 位包含从源 SSRC_n 接收到的 RTP 数据报文中最高的序列值和 16 位的扩展序列值，见目录 A。1。注意到如果开始时间差距很大，同一会话中的不同接收者会生成不同的序列值扩展。

interarrival jitter：32 bits

关于 RTP 数据报文 interarrival 时间的统计方差的估值，以 timestamp 单元来估值，表现为无符号整数。interarrival jitter J 定义为 D 的均方差，D 为接收者和发送者的间隔。像下面方程所示，等于两个报文的“相对传输时间”(the relative transit time)的差；相对传输时间是一个报文的 RTP timestamp

和到达接收者的时钟的差，在相同单元衡量。

如果 Si 是报文 i 的 RTP timestamp，Ri 是在 RTP timestamp 单元中报文 i 到达的时间，那么，对于报文 i 和 j，D 可以表达为

D(i, j) = (Rj - Ri) - (Sj - Si) = (Rj - Sj) - (Ri - Si)

每当从源 SSRC_n 接收到数据报文 i 时，都要计算 interarrival jitter，用这个差值 D 和上一个顺序到达的报文 i-1（不一定是连续的），

J(i) = J(i-1) + (|D(i-1, i)| - J(i-1))/16

每当处理一个接收报告时，都要采样当前的 J 值。

jitter 必须按照这里的公式计算，以允许 profile-independent monitor 对从不同 implementations 来的报告有正确的解释。算法是最佳的一阶 estimator，增益系数是 1/16，有很好的噪声抑制比和合理的收敛率[22]。目录 A.8 有一个例子。第 6.4.4 节有关于变长的报文持续期和传输前延迟的影响等讨论。

last SR timestamp (LSR)：32 bits

NTP timestamp（见第 4 章）的中间 32 位作为从源 SSRC_n 来的最近的 RTCP SR。如果尚未接收到 SR，域设置为零。

delay since last SR (DLSR)：32 bits

延迟定义为从接收到从源 SSRC_n 来的上一个 SR 到发送本接收报告块的间隔，表示为 1/65536 秒一个单元。如果尚未收到 SR，DLSR 域设置为零。

让 SSRC_r 表示接收者处理本接收者报告。源 SSRC_n 可以通过接收报告块到来时间 A 来计算 SSRC_r 的回环传播延迟。用上一 SR timestamp(LSR)域来计算总的回环时间 A-LSR，然后减去本域，这样回环传播延迟就是(A - LSR - DLSR)。在 Fig.2 中有解释。时间是用十六进制的表示，同样也换算成浮点的十进制表示。冒号“：”(Colon)表示一个 32-bit 域被分成一个 16-bit 整数部分和一个 16-bit 小数部分。

虽然某些链路可能有非对称的延迟，但这确是可以为一簇接收者的精确测量距离。

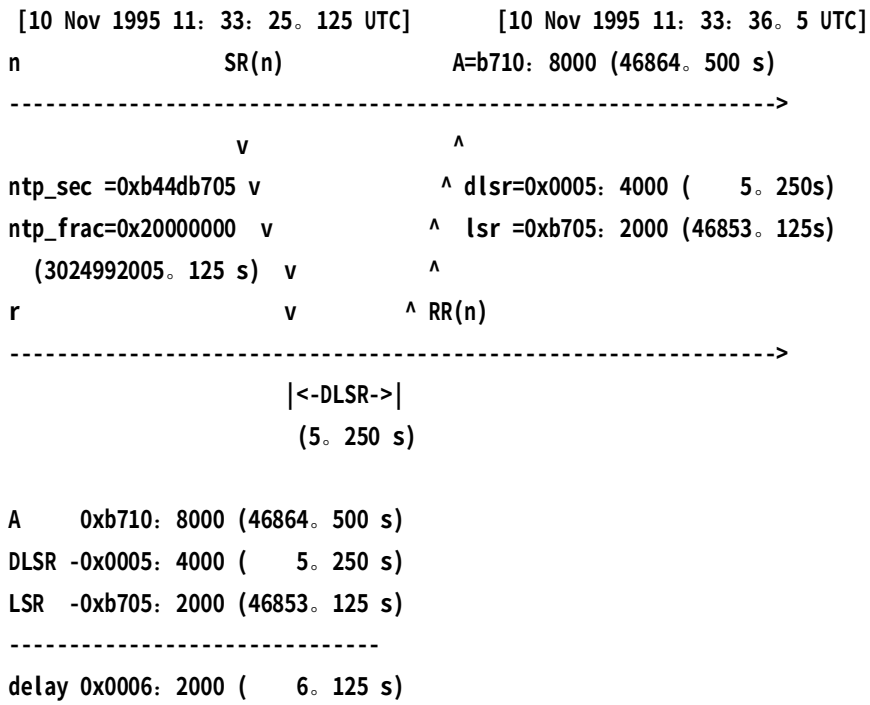
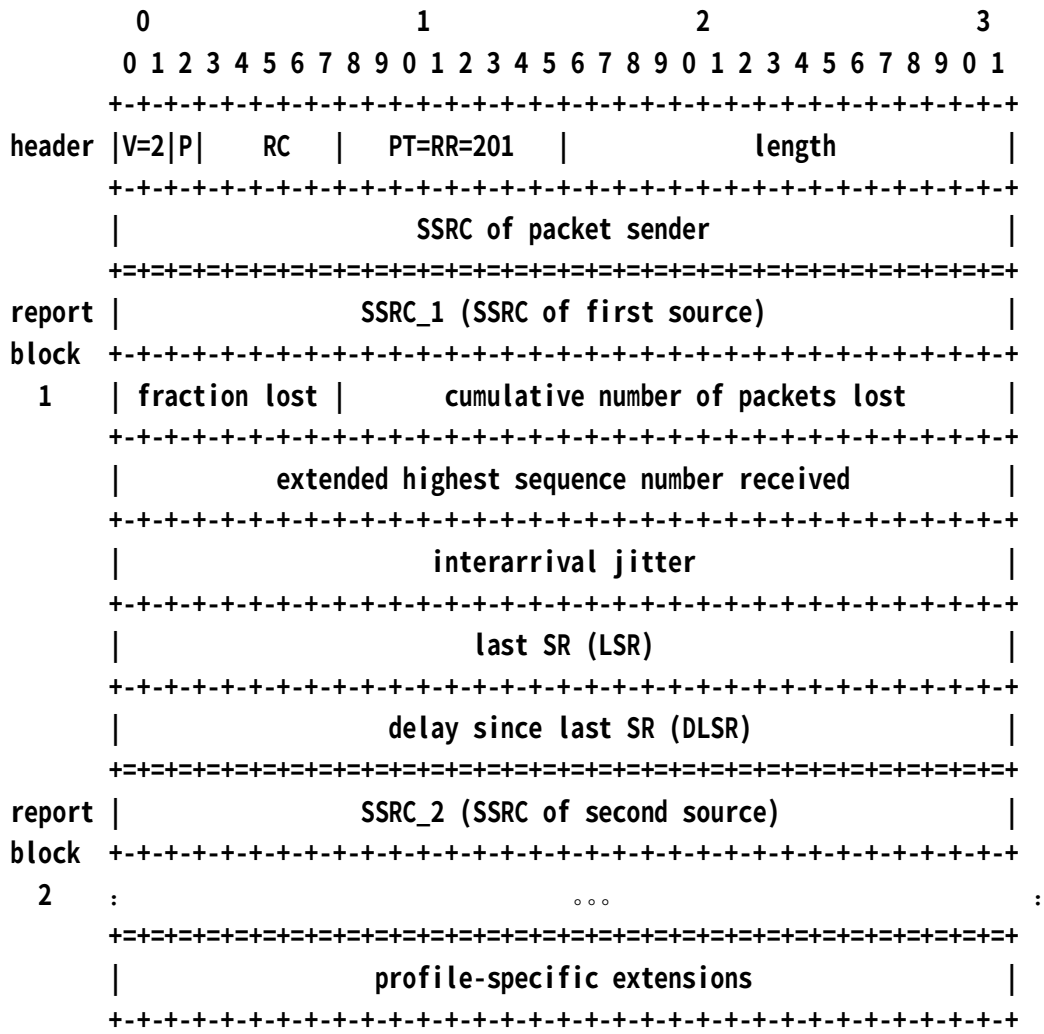


Figure 2: Example for round-trip time computation

6.4.2 RR: Receiver Report RTCP Packet



除了报文类型为 201 和 5 个字的发送者信息域（NTP 和 RTP timestamps 和 sender's packet 和 octet counts）删除外，RR 格式和 SR 是一样的。

如果没有数据传输或是接收报告，必须将空 RR 报文(RC = 0)放在混合 RTCP 报文头部。

6.4.3 扩展 SR 和 RR

如果需要有规律的报告接收者和发送者而携带附加的信息时，profile 应该对 SR 和 RR 定义 profile-specific 扩展。这种方法应优先于定义另一个 RTCP 报文，因为它需要更少的花费(overhead)：

- 更少的字段（没有 RTCP 头或 SSRC 域）；
- 更简单和更快的分解，因为运行在 profile 下的应用可以被编程为总是希

望接收报告后直接灸可以访问扩展域。

扩展是 SR 或是 RR 的第四个部分，是接收报告块之后的尾部。如果需要附加的发送者信息，那么对于发送者来说，它将先被包含在扩展段内，不会出现在接收报告内。如果包含有接收者信息，数据应该成为块组与以存的接收报告块组平行；即块号应该在 RC 域中标识。

6.4.4 SR 和 RR 的解析

接收质量反馈不仅要对发送者有用，也要对其他的接收者和第三方 monitor 有用，发送者根据反馈来更改传输；接收者能确定问题是局部的，区域性的还是全局的；网络管理员可以利用 profile-independent monitor 接收 RTCP 报文而不接收相应的 RTP 报文，分析他们网络的多播分发的性能。

发送者信息和接收者报告块都需要累积计数来计算两个报告的差别，衡量短时间或长时间的影响，对报文丢失提供恢复。可以用上两个报告的差值来估计当前的分发质量。包含 NTP timestamp 是为了利用两个报文的间隔的差值来计算速率。既然对数据编码 timestamp 与时钟率是独立的，就使程序执行 encoding-和 profile-independent 质量检测成为可能。

计算的一个例子：两个接收报告间隔的数据报文丢失率。累积的报文丢失的差值确定了间隔期间丢失的数量。上一个接收的扩展序列的差值指出了间隔期内期望的报文号数。两者等比率就是间隔期内报文丢失率(loss fraction)。如果两个报告是连续多，那么这应该与丢失域分数(fraction)相等，否则不一定。可以通过丢失率来除 NTP timestamp 的差值类获得丢失速率(rate)。单位是/s。接收的报文的数量是期望接收的减去丢失的。期望的报文的数量也可用来判断丢失估计的统计有效性。例如，5 个报文丢失 1 的影响远小于 1000 个丢失 200 个。

第三方 third-party monitor 可以利用发送者信息计算平均载荷速率和在没有接收报文的间隔期内的平均包率。两者等比率就是平均载荷大小。如果可以假定报文丢失与报文大小无关，那么特定接收者接收的报文数量乘上平均载荷长度（或是相应的报文长度）就是这个接收者的吞吐量（throughput）。

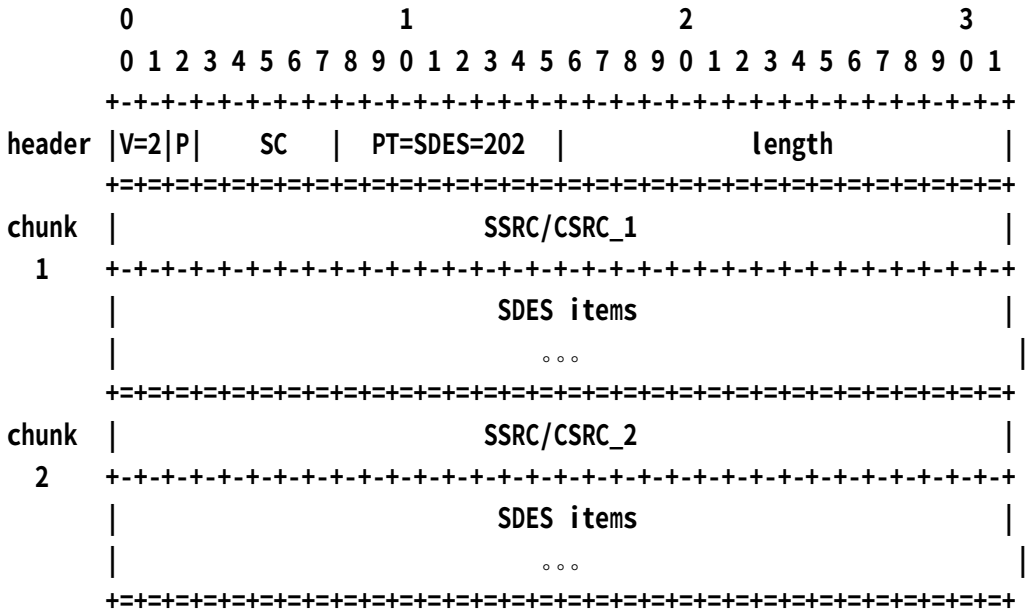
cumulative count 利用报文差值能对长期(long-term)报文丢失进行测量，除了 cumulative count 外，fraction lost 域提供对单个报告的短期(short-term)的测量。当会话的尺寸足够大以至于接收状态信息不可能存贮所有接收者或报告间隔变得足够长以至于从特定接收者只能接收一个报告，这就变得很重要了。

interarrival jitter 域提供了另一个网络拥塞的短期测量机制。报文丢失跟踪持续的拥塞而 jitter measure 跟踪短暂的拥塞。jitter measure 会在导致 packet loss 前先预示。interarrival jitter 域只是 jitter 在报告时间的一个快照，并不量化处理，而是从一个接收者时间上说是多个接收者空间上的报文作对比，例如，单网络内，同时。为了与不同接收者对比，jitter 是用同一公式

来对所有接收者计算的，这很重要。

因为 **jitter** 计算是基于 **RTP timestamp**，它表示报文第一个数据采样的时刻，在采样时刻与报文传输时刻的延迟中的方差会影响计算的 **jitter**。像这样的延迟的 **variation** 会在音频报文的变化持续期内发生。同时也会在视频编码中发生因为一帧的所有报文的 **timestamp** 是一样的，但却不是在同一时刻发送达。传输前 **Variation in delay** 确实减少了最为网络行为度量的 **jitter** 的精度，但是接收缓冲能 适应它也应考虑在内。当 **jitter** 计算作为对比测量，因为延迟中的 **variation** 所造成大常量部分应该减掉，这样只要网络 **jitter** 变化不太小，就都能检测出来。如果变化太小，这是不合逻辑地。

6.5 SDES： 源描述 RTCP 报文



SDES 报文是三级结构，由头，零个或多个 **chunk** 组成，每个 **chunk** 是由描述本 **chunk** 的源标识 **item** 组成。**item** 是独立的在子序列段中表示。

version (V)， padding (P)， length：
同 **SR**（见第 **6.4.1** 节)

packet type (PT)：8 bits
常量 **202** 标识本包是一个 **RTCP SDES** 报文。

source count (SC)：5 bits
本 **SDES** 中含有的 **SSRC/CSRC chunks** 的数量。零是有效的但没用。

chunk 是由一个 **SSRC/CSRC identifier** 及零个或更多的 **item** 组成，**item**

携带关于SSRC/CSRC的信息。chunk是由32-bit的边界开始的。item有一个8-bit的类型域，一个8-bit描述内容(text)的长度（不包括这两个字节的头）的字段计数 。text不能超过255字节，这与需要限制RTCP带宽消耗是一致的。

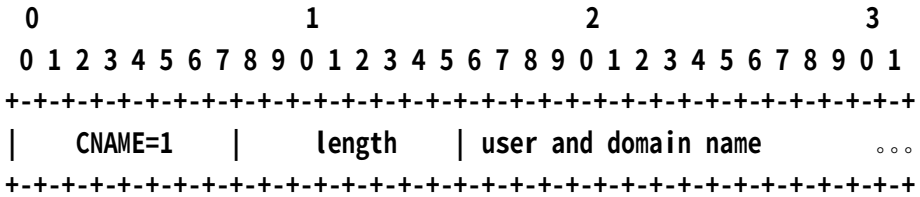
text使用UTF-8编码的，在RFC 2279 [5]中有详述。US-ASCII是这种编码的子集，不需要其他的编码方案。multi-octet编码是通过设定字符的最高有效位为零来标识的。

Item是连续的，即，并不是给item单个的填充32-bit边界。Text也不是以空(null)来结束的，因为某些multi-octet编码包含空字段。chunk中的Item表必须以一个或是多个空字节来结束，第一个空字节被译码为零类型的item来表示表尾。空类型字节之后没有长度字节，但是如果填充到下一个32-bit的边界，那就得附加空字节。这个填充是与RTCP头中由P位标识的填充是分离的(separate)。零item的chunk是有效的但无用。

终端系统发送一个包含它们自己源标识的SDES报文（同定长RTP头中的SSRC一样）。Mixer将包含chunk的SDES报文发给它接收SDES信息的贡献源，或是如果源数超过31，就发送多个完整的SDES报文。（见第7）。

SDES items的描述将在下一章展开。只有CNAME item是强制性的，这所述的某些item可能仅对特定的profile有用，但是所有的item类型都是从通用空间指派来提高共享或是简化
profile-independent应用。附加的item可以通过向IANA注册来在特定profile中定义（第15章）。

6.5.1 CNAME： 规范的终点标示 SDES Item



CNAME标识有以下特性：

o 因为如果发现冲突或者程序重起，随机分配的SSRC标识就会改变，CNAME item必须提供从SSRC标识到源标识(sender或receiver)的绑定。

o 同SSRC标识一样，CNAME标识应该在一个RTP会话中的所有参与者中是唯一的。

o 对那些在一系列相关RTP会话的同一参与者使用多媒体工具提供绑定，应与那个参与者固定。

- 为了方便第三方监测，CNAME 应该与程序或个人向配合定位源。

因此，如果可能的话，CNAME 应该是由算法而不是手动获得。为了满足这些要求，就得采用下面的格式，除非 profile 定义了交互的语法和语义。CNAME item 应该有"user@host"格式或如果在单用户系统没有 user 则只提供"host"格式。对于两种格式，"host"或是全授权的实时数据的发源地的主机名，在 RFC 1034 [6]，RFC 1035 [7]和 RFC 1123 中的第 2 章[8]中定义了格式；或是在 RTP 通信使用的界面的标准 ASCII 表示的主机数字地址。例如，IPv4 中的标准 ASCII 表示为"点分十进制"，也叫 dotted quad，在 IPv6 中地址是表示为一组十六进制由冒号分开当数字（见 RFC 3513 [23]）。其他的地址类型希望有 ASCII 表示为相互唯一就可以了。全授权的域名对人类很便捷，而且避免了发送附加 NAME item。然而在一些操作环境却很难或不可能得到可靠性。在这类环境下运行的应用应该使用 ASCII 表示地址。

例如"doe@sleepy。example。com"，"doe@192。0。2。89"或"doe@2201：056D：：112E：144A：1E24" 。没有使用者名字段系统，例"sleepy。example。com"，"192。0。2。89" 或"2201：056D：：112E：144A：1E24"。

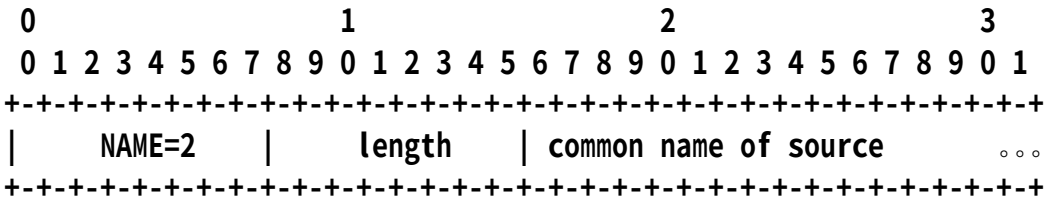
使用者名字应该使用这样一种格式，像程序"finger"或"talk"都可以使用，即，它应该是登陆名而不是个人名。域名不必与参与者的 email 地址相同。

如果应用允许使用者从同一主机生成多个源，本语法就不再给每个源提供唯一的标识。应用就得依赖 SSRC 来确定源，或者应用的 profile 给 CNAME 标识定义附加的语法。

如果应用各自创建 CNAME，CNAME 就不会像希望那样来提供绑定。如果需要交叉(cross-media)媒体绑定，那么每一个工具的 CNAME 都由调和工具配置为相同的值。

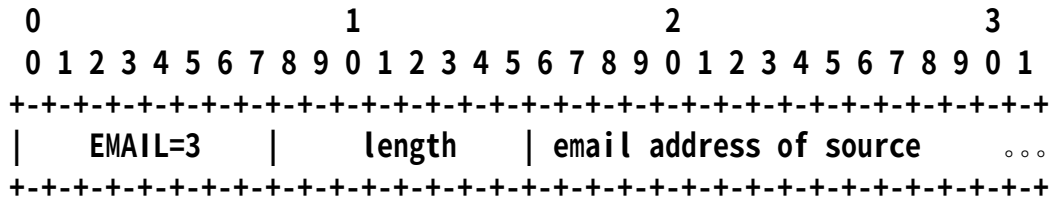
应用程序开发者应该意识到私有网络地址如在 RFC 1918 [24]中的 NET-10，可以创建不是全局唯一的网络地址。如果具有私有网络地址和非直接 IP 连接到公共因特网的主机通过 RTP 级的 Translator 将 RTP 报文前向发送到公共网路，就会导致 CNAME 的不唯一（见 RFC 1627 [25]）。处理这个问题就应该提供一种可使 CNAME 唯一的办法，这样就需要 Translator 转换私有地址的 CNAME 到公有地址，并且尽可能不要曝露私有地址。

6.5.2 NAME：用户名 SDES Item



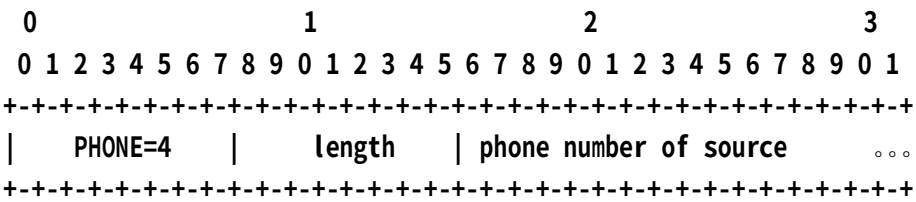
这是用来描述源的真实名字，例如 "John Doe，Bit Recycler"。用户可以用任何形式的名字，对于像会议这样的应用，这是最合适的形式，因此也会更频繁的发送这些 item 而不是 CNAME。Profile 可以建立这样的特性。NAME 值至少在会话期内应该恒定。在同一会话的所有参与者的唯一是不可靠的。

6.5.3 EMAIL：电子邮地址 SDES Item



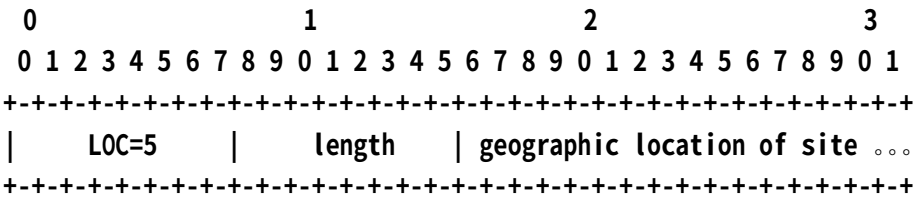
Email 地址定义的格式见 RFC 2822 [9]，例如"John。Doe@example。com"。EMAIL 值在会话期内是恒定的。

6.5.4 PHONE：电话号码 SDES Item



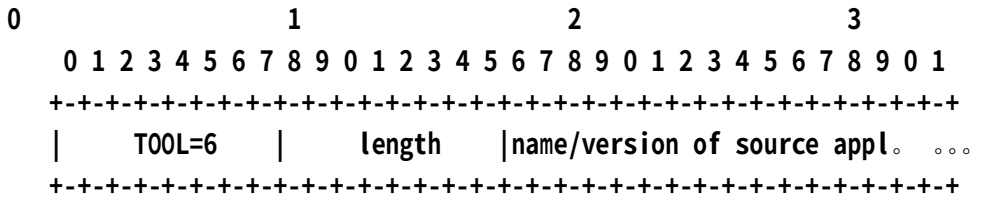
电话号码被编成格式为“+”取代国际码。例如，"+1 908 555 1212"是美国的号码。

6.5.5 LOC：用户地理定位 SDES Item



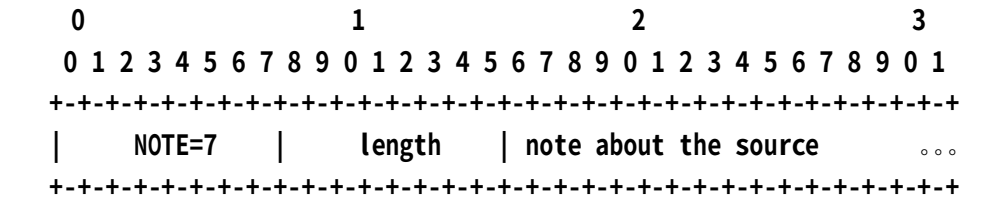
不同应用对应信息的详略程度。对于会话应用，字符串"Murray Hill， New Jersey"可能就足够了，而对于一个活跃的证件系统，字符串"Room 2A244， AT&T BL MH"可能才会合适。详略程度是由 implementation 和/或用户来定的，但是格式和内容可能是由 profile 来描述的。除了移动主机，LOC 值应该在会话期内是恒定的。

6.5.6 TOOL：应用或工具名 SDES Item



一个字符串，提供名字和可能的生成流的应用的版本。例如， "videotool 1。2"。这些信息对调试有用，与 Mailer 或 Mail-System-Version SMTP 头相似。在会话期内 TOOL 值应恒定。

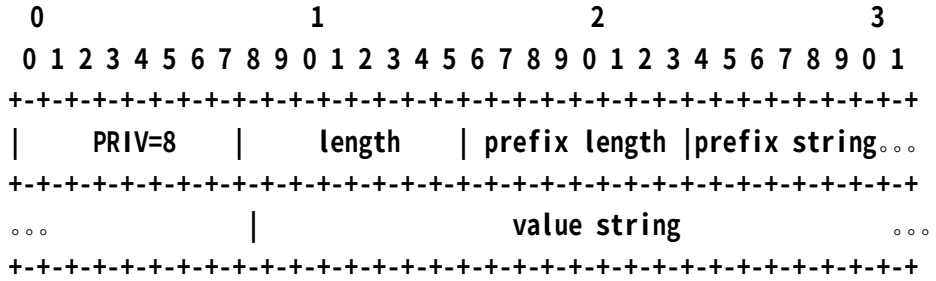
6.5.7 NOTE： 通知/状态 SDES Item



下面是这个 item 的建议语义，profile 可能明确定义了这些或是其它的语义。NOTE item 用来描述源的当前暂态消息，例如 "on the phone， can't talk"，或在语义中，本 item 可以用来传递讲话的标题。应该仅仅用它来携带例外的信息，不能被所有参与者循规蹈矩的利用，因为这样会降低发送接收报告和 CNAME 的速率，削弱协议的性能。特别是，不应在用户确认中作为 item 或是在 quote-of-the-day 中自动生成。

因为激活之后，显示 NOTE item 可能很重要，而其它 non-CNAME items 像 NAME 的传输速率就会减少，这样 NOTE item 就可以占用那部分 RTCP 带宽。当暂态消息不活跃时，NOTE item 应以同样的速率持续的传输一段时间但是以零长度的字符串来通知接收者。然而，如果接收者在很小的若干接收速率或是 20-30 RTCP 间隔内没有收到 NOTE item，接收者就应该考虑到 NOTE item 非活跃。

6.5.8 PRIV: 私人扩展 SDES Item

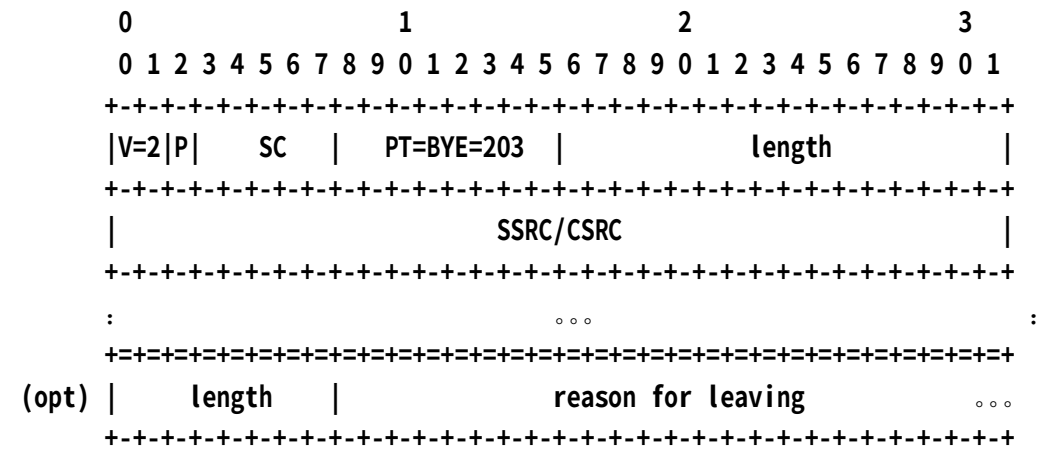


本 item 用来定义试验的或是 application-specific SDES 扩展。Item 包含一个由 length-string 对组成的前缀，然后是值串填充剩余的部分和携带需要的信息。前缀的长度是 8 比特。前缀字符串是由定义 PRIV item 来选的，一定要与本应用接收的其它的 PRIV item 不同。应用创建者可以选择使用应用名，如果需要，加上一个附加的子类型标识。建议根据它们代表的实体来选择，在实体内调整名字的使用。

前缀占用了 item 总长 255 字节的一些空间，所以前缀应该越小越好，这个设备和限制的 RTCP 带宽不应过载，也不必满足所有应用的所有通信控制需求。

SDES PRIV 前缀不会由 IANA 注册的，如果某些特定格式的 PRIV item 常用，它应由 IANA 注册成为政规定的 SDES item 类型，这样就不需要前缀了，这将使应用更为简单和提高传输效率。

6.6 BYE: Goodbye RTCP Packet



BYE 报文表示一个或多个源不再活跃。

version (V), padding (P), length:

同 SR 报文描述相同（见第 6。4。1 节）。

packet type (PT)： 8 bits

常量 203 标识本报文史一个 RTCP BYE 报文。

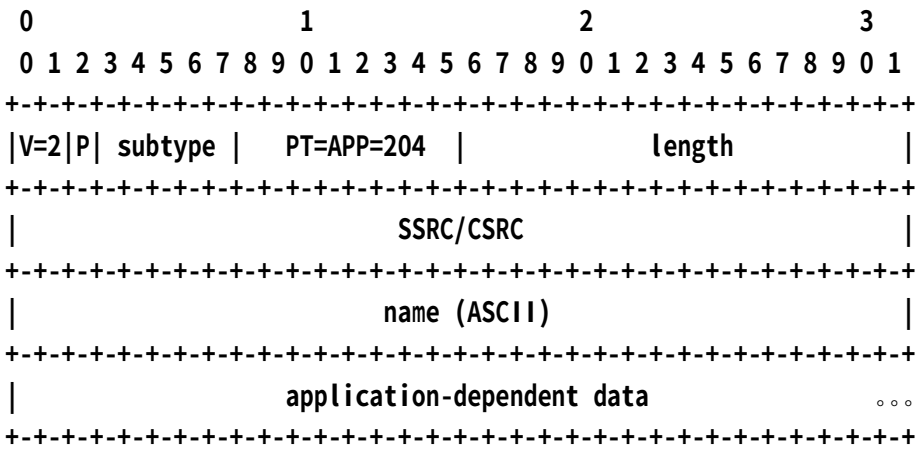
source count (SC)： 5 bits

本 BYE 报文中包含的 SSRC/CSRC identifier 的数量。零时有效的但没用。

发送 BYE 报文的规则见第 6.3.7 节和第 8.2 节。

如果 Mixer 接收到了 BYE 报文，Mixer 应该前向发送 BYE 报文而 SSRC/CSRC identifier(s)不变。如果 Mixer 关闭了，它应该发送 BYE 报文来报告它所处理等所有贡献源和自己的 SSRC identifier。BYE 报文包含一个字节到计数，随其后的是表示离开原因的文字字段，例如 "camera malfunction"或"RTP loop detected"。字符串与 SDES 有同样的编码方式。如果字符串将下一个 32-bit 边界填充了报文，字符串就不会以空结束。如果不是，BYE 报文必须给 32-bit 边界填充空字段。这个 padding 与在 RTCP 头由 P 位标识的是分开的。

6.7 APP： 应用定义的 RTCP 报文



APP 报文是给新应用或开发的新特性做试验用，不需要报文类型注册。忽略那些不识别的 APP 报文名字。测试之后如果证实可以打更宽的应用，建议从新定义 APP 报文，不要子类型和名字域，并且用 RTCP 报文类型向 IANA 注册。

version (V)， padding (P)， length：

与 SR 报文描述相同（见第 6。4。1 节）。

subtype：5 bits

可以作为子类型来允许一系列的 APP 报文在一个名字下定义，或是是为了任何的 application-dependent 数据。

packet type (PT)：8 bits

常量 **204** 标识本报文是 **RTCP APP** 报文。

name：4 octets

又定义 **APP** 报文系列的人来选择，应与其它的在本应用可能接收的 **APP** 报文不同，应用的创建者可以选择使用应用名字，然后调整子类型值的分发给那些给应用定义新报文类型的人。建议其他者根据他们代表的实体来选择名字，然后在实体内调整名字的使用。名字是解释成一个四个 **ASCII** 字符的序列，区分大写字母和小写字母。

application-dependent data：variable length

Application-dependent 数据可以也可以不出现在 **APP** 报文中。它是由应用解释而不是 **RTP** 本身，必须是 **32** 比特的倍数。

7. RTP Translators and Mixers

除了终端系统外，**RTP** 支持**"translators"**和**"mixers"**的概念， 可以把它们看成是 **RTP** 级的中间件。虽然这种意义上的支持给协议增加了复杂性，这些功能的要求已经明确的通过因特网上多播音视频试验而制定出来了。在第 **2.3** 节的关于 **Translators** 和 **Mixers** 的使用是在防火墙和低带宽连接的情况，这两者是可能存在的。

7.1 概括性描述

一个 RTP translator/mixer 连接了两个或多个传输级的"cloud"（云）。典型的，每一个云是由通用网络或是传输协议加上多播地址和传输级的目的端口或是单播地址和端口对定义的。（网络级的协议 translators，例如 IPv4 对 IPv6，会出现在 cloud 内但对 RTP 是不可见的）。一个系统可以为很多个 RTP 会话提供 Translator 或 Mixer 服务，但是逻辑上是独立地实体。

为了防止 Translator 或 Mixer 安装时带来的循环，应关注下下面的规则：

o 每一个在一个 RTP 会话中由 Translators 和 Mixers 连接的云，或者必须至少在一个参数（协议，地址，端口）上与其它云区分开来，或者在网络级与其它云隔离。

o 第一条规则派生出来的就是不能有很多的 Translators 或 Mixers 并联，除非根据安排他们前向的分割源。

同样地，所有的可以通过一个或多个 RTP Translators 或 Mixers 通信的 RTP 终端系统共用相同的 SSRC space，即，SSRC identifiers 在这些终端系统中是唯一的。第 8.2 节描述了冲突解决算法，SSRC identifier 是唯一的，循环也是检测了的。

对于不同目的和应用设计了不同的 Translators 和 Mixers，例如增加或移除加密，改变数据或底层协议的编码，在多播地址和一个或多个单播地址之间复制。Translators 和 Mixers 的区别是 Translator 独立的通过不同源的数据流，而 Mixer 将它们合并成为一个新流：

Translator：前向输出 RTP 报文而不改变它们的 SSRC identifier；这样即使来自不同源的报文通过了同一个 Translator，或是携带了 Translator 的网络源地址，接受者也可以识别出单个的源。一些类型的 Translator 可以不改变通过的数据，而有些则可能改变数据的编码和 RTP 数据载荷类型和时间戳。如果多媒体数据报文被重编码成一个报文，或是相反，Translator 必须重新给输出的报文安排新的序列号。损失包括相应地间隙。接收者不能检测到 Translator 的存在，除非它们通过其它方法知道源用了 payload type 或是传输地址。

Mixer：从一个或多个源接收 RTP 数据报文流，可能改变数据格式，以某种方式合并流，然后前向输出。因为多个源的时序的同步时不可能的，所以 Mixer 会在流之间调整时序，生成合并流的自己时序，它就是同步源。因此，所有 Mixer 前向输出的数据报文必须以 Mixer 自己的 SSRC identifier 标识。为了保存源的身份，Mixer 应该将他们的 SSRC identifier 插到 CSRC identifier 表中，如果 Mixer 自己也是源的话，就将自己的 SSRC identifier 明确的插到那个报文的 CSRC 表中。

对于特定应用，Mixer 没有在 CSRC 表中鉴别源也是可以接受的，然而这引进了危险，即包含这些源的循环不能被检测到。

在像音频这样地应用中，Mixer 相比于 Translator 的优势在于输出带宽被限制为一个源的带宽，即使有很多源在输入端是活跃的，这对低带宽的连接很重要。缺点是输出端的接收者不能控制那个源通过了那个源沉默了，除非执行了远端 Mixer 控制机制。Mixer 的同步信息的重生也意味着接收者不能对原始流进行 inter-media 同步，而一个多媒体 Mixer 可以。

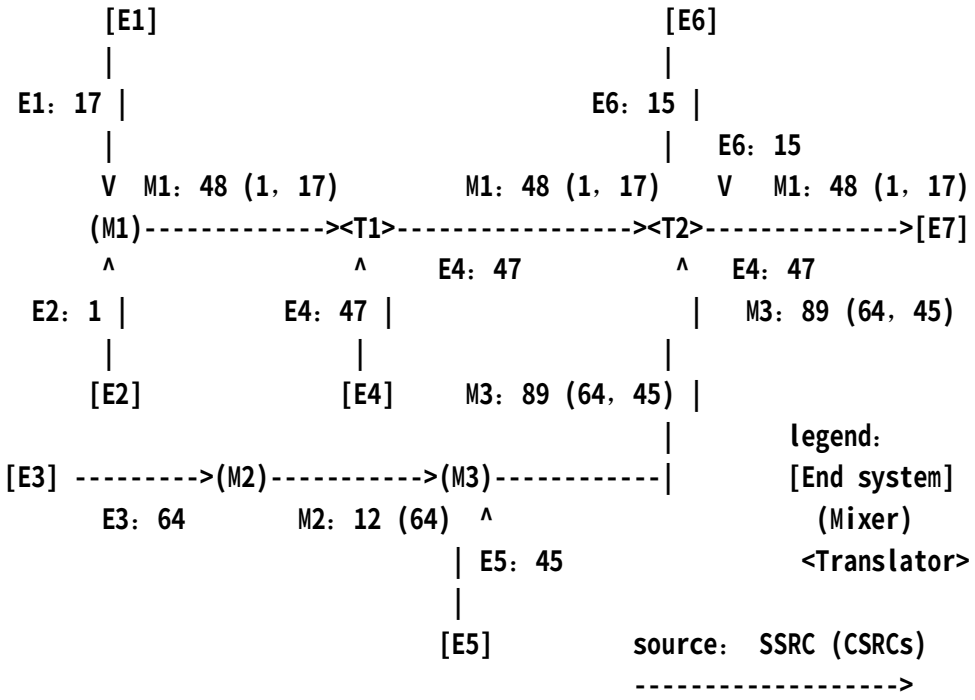


Figure 3: Sample RTP network with end systems, mixers and translators

Fig.3 给出了 Mixer 和 Translator，阐述它们对 SSRC 和 CSRC identifier 的影响。图中，终端系统是矩形（以 E 命名），translators 是三角形（以 T 命名）和 Mixer 是椭圆（以 M 命名）。符号 "M1: 48(1, 17)"指出报文是使由 Mixer M1 生成的，由 M1's (random) SSRC 的随机值 48 标识。两个 CSRC identifier 是 1 和 17，从 E1 和 E2 的报文中的 SSRC identifier 拷贝而来的。

7.2 在 Translator 中处理的 RTCP

除了前向输出的数据报文可能被改变了，Tanslator 和 Mixer 必须处理 RTCP 报文。在很多情况下，它们会将从终端来的混合 RTCP 报文分开，收集 SDES 信息和修改 SR 或 RR 报文。这些报文的中继是由报文的到来或是 Translator 或是 Mixer 自身的 RTCP 间隔定时触发的。

没有修改数据报文达 Translator，例如仅仅在多播地址和单播地址之间进行了复制，可能会同样简单的前向输出 RTCP 报文而不修改。以某种方式传输 payload 的 Translator 必须在 SR 和 RR 信息中生成相应的改变，这样就仍能反映数据的特定和接收质量。这些 Translator 不准简单的前向输出 RTCP 报文。一般来讲，Translator 不应将从不同源来的 SR 或是 RR 报文汇聚成一个报文，因为那样会减少根据 LSR 和 DLSR 域的传输延迟测量的精确性。

SR sender information： Translator 不生成自己的发送信息，前向输出从一个云接收来的 SR 报文，不改变 SSRC，如果转化要求，但是必须修改发送者信息。如果 Translator 改变了数据编码，就必须改变"sender's byte count"域。如果它也合并了几个数据报文成为一个输出报文，也必须改变"sender's packet count"域。如果改变了 timestamp 频率，必须改变在 SR 报文中的"RTP timestamp"域。

SR/RR reception report blocks： Translator 前向输出从一个云中接收到的接收报告。注意这些流的方向与数据相反。不改变 SSRC。如果 Translator 将很多的数据报文合并成一个输出报文，并且改变了序列号，它必须对报文丢失域和"extended last sequence number"域进行相反的操作，这可能很复杂。极端情况，没有很好的转换接收报告的方式，Translator 就会根本不传递接收报告或是基于自己接收的综合报告。一般的规则是对特定转换进行特殊对待。

Translator 不需要自己的 SSRC identifier，但可以选择分配一个来发送关于他所接收的报告。这会传给所有的连接的云，each corresponding to the translation of the data stream as sent to that cloud， 因为接收报告对所有参与者是多播得到。

SDES： Translator 不改变 SDES 信息前向输出它们从一个云中接收的信息，但是可能，例如如果带宽限制，就滤出 non-CNAME SDES 信息。CNAME 必须被前向输出来允许 SSRC identifier collision detection 工作。生成自己 RR 报文的 Translator 必须发送关于自己的 SDES CNAME 信息给那些接收了自己 RR 报文的云。

BYE： Translator 不改变的前向输出 BYE 报文。将要停止前向输出报文的 Translator 因发 BYE 报文给连接的云，并且包含以前前向输出给那个云的所有 SSRC identifier，如果它发送了自己的报告，就要包括 Translator 自己的 SSRC identifier。

APP： Translator 不改变的前向输出 APP 报文。

7.3 在 Mixer 中的 RTCP 处理

既然 Mixer 生成自己的新的数据流，它就根本不通过 SR 或是 RR 报文，而是

给两边生成新的信息。

SR sender information： **Mixer** 不通过从它混合的源的发送者信息，因为源流特性在混合式丢失了。 作为一个同步源，**Mixer** 应生成自己的关于混合数据流的发送者信息的 **SR** 报文，然后作为混合的流以相同的方向将它们发出。

SR/RR reception report blocks： **Mixer** 在每一个云中为源生成自己的接收报告。然后仅将它们发送给相同的云，不能将接收报告发给其它的云，也不能将接收报告从一个云前向输出给其它的云，因为源在那不是 **SSRC**（仅仅是 **CSRC**）。

SDES： **Mixer** 前向输出而不改变从一个云来的 **SDES** 信息，但是可能，例如如果带宽限制，可以滤出 **non-CNAME SDES** 信息。**CNAME** 必须被前向输出来允许 **SSRC identifier collision detection** 工作。（由 **Mixer** 生成的 **CSRC** 列表中的 **identifier** 可能与终端系统生成的 **SSRC identifier** 冲突。）**Mixer** 必须发送关于自己的 **SDES CNAME** 信息给那些接收了自己 **SR** 或 **RR** 报文的云。

既然 **Mixer** 不能前向输出 **SR** 或 **RR** 报文，它们就会从混合 **RTCP** 报文中提取 **SDES** 报文。为了使花费最小，从 **SDES** 报文来的 **chunk** 可能会被合并成一个单一的 **SDES** 报文，然后 **stack** 在从 **Mixer** 来的 **SR** 或是 **RR** 报文上。合并 **SDES** 报文的 **Mixer** 会比一个单一源消耗更多的 **RTCP** 带宽，因为混合报文会更长，但这是合适的因为 **Mixer** 代表了多源。同样的，想接收那样通过 **SDES** 报文的 **Mixer** 会以一个高于单源的速率传输 **RTCP** 报文，同样这也是正确的因为报文是来自多源。**Mixer** 两边的速率是不同的。

没有插入 **CSRC identifier** 的 **Mixer** 可能同样避免**(refrain from)**前向输出 **SDES CNAME**。这种情况下，两个云内的 **SSRC identifier space** 是独立的。早就指出，这种操作模式有循环不能被检测危险。

BYE： **Mixer** 必须前向输出 **BYE** 报文。将要停止前向发送报文的 **Mixer** 应发送 **BYE** 报文给连接的云，含有以前被前向输出给那个云的所有 **SSRC identifier**，如果发送了自己的报告，还应包含 **Mixer** 自己的 **SSRC identifier**。

APP： **Mixer** 对待 **APP** 报文是 **application-specific** 的。

7.4 级联的 Mixer

一个 **RTP** 会话可能就像 **Fig**。**3** 所示由 **Mixer** 和 **Translator** 构成。如果两个 **Mixer** 是级联的，如 **M2** 和 **M3**，**Mixer** 接收到的报文可能已经被混合了，包含一个有多个 **identifier** 的 **CSRC list**。第二个 **Mixer** 应该用已经混合的输入报文的 **CSRC identifer** 和没有混合的输入报文的 **SSRC identifier** 来为输出报文构

建 CSRC list。这在图中，如标有 M3：89(64，45)的 M3 所示。在没有级联的 Mixer 情况，如果 identifier 多于 15 个，多的不会被包含在内。

8. SSRC Identifier 分配和使用

RTP 头和 RTCP 报文很多域中携带的 SSRC identifier 是一个随机的 32-bit 号，在 RTP 会话内是唯一的。选择号时必须谨慎，以防同一网络或同时启动的会话选择相同的号。

对于 identifier 用局域网地址是不够的（例如一个 IPv4 地址），因为地址不唯一。因为 RTP translator 和 mixers 允许不同地址 space 的网络内操作 (interoperation) ，两个 space 内的地址分配类型就会有比随机分配更高的冲突率。

运行在一个主机的多个源也会有冲突。

仅仅简单调用 random()而没有小心的初始状态就得到一个 SSRC identifier 也是不够的。目录 A.6 中有一个关于怎样生成一个随机 identifier 的例子。

8.1 冲突的概率

既然 identifier 是随机选择的，两个或多个源就有肯能选择同一号。当很多源同时启动时，冲突的概率就很高。例如由特定会话管理事件自动触发的。假设源数是 N，identifier 长度是 L（在这是 32 比特）。 两源独立选择相同的值得概率可以接近 1 - exp(-N**2 / 2**(L+1))，当 N 很大时。对于 N=1000，概率是 10**-4。

一般的概率是比最坏的情况小得多。当一个新会话加入 RTP 会话，并且其它的源都有唯一的 identifier 时，the probability of collision is just the fraction of numbers used out of the space。同样设定源数 N，identifier 长度是 L，冲突的概率是 N / 2**L。对于 N=1000，
概率大约是 2*10**-7。

如果新源在发送第一个报文（不管是数据还是控制）之前有机会接收到从其它参与者发来的报文，冲突的概率就会锐减。如果新源跟踪其它的参与者（通过 SSRC identifier），在发送第一个报文之前，新源可以证实自己的 identifier 没有与接收到的任何源冲突，有则另选。

8.2 冲突解决和回环(loop)检测

虽然 SSRC identifier 冲突的概率很低，但是所有的 RTP implementation 必须准备检测冲突和采取适当的方式处理。如果一个源发现另一个源在用同样的 SSRC identifier，它必须给 identifier 发送一个 RTCP BYE 报文，然后另选一个新的。(见下解释，这步在一次循环内只执行一次)。如果接收者发现两个源冲突了，当能通过不同的源传输地址或 CNAME 检测出来时，接收者就会接收一个丢弃其它的。希望两个源解决冲突以使这样的形势不会持续下去。

因为在一个会话内随机的 SSRC identifier 是全局唯一的，也可以用它们来检测可能由 Mixer 或 Translator 引进到回环。一个环路导致数据和控制信息的复制，或是不修改或是混合，如下例：

- Translator 可能不正确的或者直接或通过 Translator 链前向输出一个报文给相同的多播组。在这种情况下，同样的报文会出现几次，从不同的网络源而来。

- 两个 Translator 错误的并行启动。即，两边具有同样的多播组，两者都前向的从一个多播组输出报文给另一个多播组。单向的 Translator 会生成两个副本。双向的 Translator 会形成回环。

- Mixer 可以通过直接的或通过另一个 Mixer 或 Translator 来发送报文给相同的目的来关闭回环。在这种情况下，源会以数据报文的一个 SSRC 或混合报文的 CSRC 的身份出现。

源可以发现自己的报文回环了，或是从另一个源来的报文回环了(一个第三方回路)。由于随机选择源 identifier 而引起的回环和冲突都会导致到来的报文拥有相同的 SSRC identifier，但是由于终端系统生成的报文或是中间系统而具有不同的源传输地址。

因此，如果源改变了源传输地址，也会选择一个新的 SSRC identifier 来避免被解释成回路源。(这不是必须的，因为在一些 RTP 应用中，在会话中源可能会改变地址。) Note 值得注意的是如果 Translator 重启，从而改变了源传输地址(例，改变了 UDP 源端口号)，那么所有的报文对接收者来说都是回环的因为最初的源用了 SSRC identifier 而且也不会改变。这个问题可以通过重启时固定源传输地址来避免的。任何一种情况，接收者在 timeout 之后都会解决这个问题的。

如果报文的所有副本都通过 Translator 或 Mixer 的话，在 Translator 或 Mixer 远端发生的回环或冲突不能用源传输地址检测到，然而当有从两个有相同的 SSRC identifier 不同的 CNAME 的 RTCP SDES 报文来的 chunk 时，仍能检测到冲突。

为了检测和解决冲突，尽管 implementation 可以为来自第三方源冲突的报文选择一个不同的政策，但 RTP implementation 必须包含一个与下所述相似的算法。下面描述的算法忽略了那些从与一个以建立的源相冲突的新源或回路的报文。它通过给旧的 identifier 发送 BYE 报文和选择一个新的来解决冲突。如果冲突是由参与者自己的报文回环而引起，算法只能选择一个新的 identifier，仅仅选择一次，然后再有冲突就忽略从回环源传输地址来的报文。这是用来避免 BYE 报文的雪崩。

算法需要维持一个由源 identifier 索引的表，包含从第一个 RTP 报文来的源传输地址，和第一个用那个 identifier 接收的 RTCP 报文，同时还有那个源的其它状态，以后需要两个源传输地址。例如 UDP 源端口号在 RTP 和 RTCP 报文中可能不同。然而，可以假设网络地址在两个源传输地址端是相同的。

在 RTP 和 RTCP 报文中接收到的 SSRC 或 CSRC identifier 要去查源 identifier 表，采用处理那个数据或是控制信息。报文中的源传输地址也要与相应地的表对比，如果不匹配，就要检测回环或冲突。对于控制报文。对于控制报文，每一个有自己 SSRC identifier 的元素(element)，例如一个 SDES chunk，需要一个单独的查找表。（在接收报告块中的 SSRC identifier 是例外，因为它标识为由报告者侦听到的源，那个 SSRC identifier 与报告者发送的 RTCP 报文的源传输地址无关。）如果没有发现 SSRC 或 CSRC，就创建一个新的实体。当接收到了一个 RTCP BYE 报文与相应的 SSRC identifier，并且由匹配的源传输地址确认，或者相对很长时间没收到报文，就会移除这些表实体。见第 6.2.1 节）。

注意，如果一个主机的两个源在接收者开始操作时用相同的源 identifier 传输，就会有这样一种可能，即从一个源接收到第一个 RTP 报文，而从另一个源接收到第一个 RTCP 报文。这就会导致用错误的 RTCP 报文关联错误的 RTP 数据，但是这种情况是很少见的并且可以忽视。

为了追踪参与者自己的数据报文的回路，implementation 必须也保持一个被证为有冲突的源传输地址表。如同在源 identifier 表中一样，必须两个源传输地址独立的追踪冲突的 RTP 和 RTCP 报文。注意冲突的地址表应该短，通常为空。表中的 element 存储了关于源地址和最近冲突报文接收的时间。当 10RTCP 报告间隔这样一段时间内没有冲突报文到来，element 就会被移出表（见第 6.2 节）。

对于所示的算法，假设参与者自己的源 identifier 和状态都包含在 source identifier 表中。可以重建算法来首先单独与参与者自己的源 identifier 作对比。

```
if (SSRC or CSRC identifier is not found in the source
        identifier table) {
```

```
    create a new entry storing the data or control source
        transport address,  the SSRC or CSRC and other state;
}

/* Identifier is found in the table */

else if (table entry was created on receipt of a control packet
         and this is the first data packet or vice versa) {
    store the source transport address from this packet;
}
else if (source transport address from the packet does not match
         the one saved in the table entry for this identifier) {

    /* An identifier collision or a loop is indicated */

    if (source identifier is not the participant's own) {
        /* OPTIONAL error counter step */
        if (source identifier is from an RTCP SDES chunk
            containing a CNAME item that differs from the CNAME
            in the table entry) {
            count a third-party collision;
        } else {
            count a third-party loop;
        }
        abort processing of data packet or control element;
        /* MAY choose a different policy to keep new source */
    }

    /* A collision or loop of the participant's own packets */

    else if (source transport address is found in the list of
             conflicting data or control source transport
             addresses) {
        /* OPTIONAL error counter step */
        if (source identifier is not from an RTCP SDES chunk
            containing a CNAME item or CNAME is the
            participant's own) {
            count occurrence of own traffic looped;
        }
        mark current time in conflicting address list entry;
        abort processing of data packet or control element;
    }

    /* New collision,  change SSRC identifier */
```

```
        else {
            log occurrence of a collision;
            create a new entry in the conflicting data or control
                source transport address list and mark current time;
            send an RTCP BYE packet with the old SSRC identifier;
            choose a new SSRC identifier;
            create a new entry in the source identifier table with
                the old SSRC plus the source transport address from
                the data or control packet being processed;
        }
    }
```

在这个算法中，从最近冲突的源地址来的报文会被丢弃，而保留原来的源地址来的报文。如果延长的周期内还没有从最初的源来的报文的话，表实体就会超时，新的源就可以接管。如果最初的源检测到冲突，并且改为新的源 **identifier**，这就有可能发生，但是通常情况，会从最初的源收到一个 **RTCP BYE** 报文，在没有到定时时间就将状态删除。

如果最初的源地址是通过 **Mixer** 接收的（即作为一个 **CSRC**），不久直接的接收了同样的源，建议接收者改向新源地址，除非丢弃在 **Mix** 中的其它的源。此外，像电话**(telephony)**这样的应用，像移动实体这样的源在一个 **RTP** 会话期间内会改变地址，**RTP implementation** 应修改冲突检测算法来接收从新的源传输地址来的报文。如果真正的冲突确实发生了，为了防止地址之间的 **flip-flopping**，算法应该含有一些机制来检测这种情况，避免转换。

当因为冲突而选择了一个新的 **SSRC identifier**，应先查源 **identifier** 表来确定候选的 **identifier** 是否已经使用。如果使用了，就应将生成一个新的候选者，然后重复以上操作。

数据报文对多播目的的回环会导致很严重的网络问题。所有的 **Mixer** 和 **Translator** 必须执行一个像这里给出的环路检测，使得它们可以打开环路。这会限制过剩的 **traffic** 到一个不超过一个最初 **Traffic** 的复制，允许会话继续来发现确定环路。然而，在极端的情况，**Mixer** 或 **Translator** 不能恰当的打开环路，终端系统就需要完全地停止发送数据或控制报文。这是由应用决定的。适当的标识错误状况。经过一个长的，随机的时间（分钟级的）后，周期的尝试传输。

8.3 Use with Layered Encodings

对于在单独的 **RTP** 会话中传输的层编码（见第 **2.4** 节）， 所有层的会话都应使用一个单独的 **SSRC identifier space**，核心**(core(base))**层应被用来分配 **SSRC** 和解决冲突。当一个源发现冲突了，仅仅在 **base** 层传输一个 **RTCP BYE** 报文，但在所有层都改变 **SSRC identifier**。

9. 安全

低层协议可能最终提供所有的 RTP 应用需要的安全服务，包括鉴定，完整性，机密性。这些服务对 IP[27]有详细的说明。因为用 RTP 初始音视频应用在 IP 层提供这些服务之前需要机密性服务，下面描述的机密性服务是为了使用 RTP 和 RTCP 而定义的。RTP 的新应用可以后向兼容的执行这个 RTP-specific 机密性服务，和/或它们可以选择执行其它的安全服务。这个机密性服务的花销是很小的，所以如果将来这个服务被其它服务取代了，代价也是很小的。

作为选择地，其它的服务，其它的服务的 implementation 和其它的算法会在将来为 RTP 定义。特别是，正在开发一个称作安全的实时传输协议(SRTP)[28]的 RTP profile ，当不受阻碍地离开 RTP 头时提供 RTP payload 的机密性，这样链路级头压缩算法就继续执行。希望 SRTP 是许多应用的正确选择。SRTP 建立在高级加密标准(AES)之上的，提供比这里描述的更强大的安全性。并没有声明这里所述的方法对特定的安全需要是合适的。profile 说明应用应该提供哪些服务和算法，和提供关于它们恰当应用的指导。

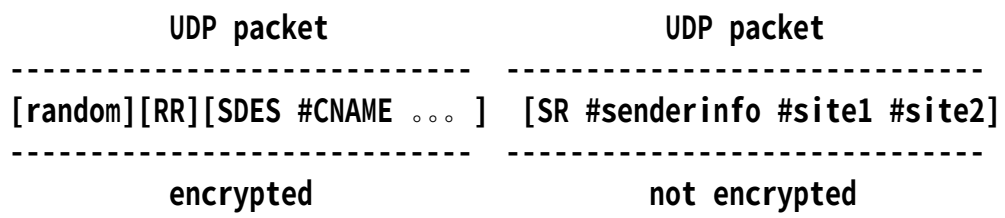
关键的分配和证书不在本文档讨论之内。

9.1 机密性

机密性意味着只有有意的接收者可以解码，对于其他者而言，机密性只是一堆无用的信息。内容的机密性是由加密完成的。

当想根据本章描述的方法来加密 RTP 或 RTCP 报文时，所有将要封装在一个低层报文的字段都是作为一个单元来加密的。对于 RTCP，必须在加密之前预先为每个单元计划一个 32-bit 随机数。对于 RTP，不需要预先计划前缀；相反，用一个随机偏移量(offset)来初始序列号和时间戳域。这个因为随机性不好而被认为是一个弱的初始向量(IV)。初次之外，如果子序列域，SSRC，可由敌方来操作，这种加密算法的缺点就更大了。

对于 RTCP， implementation 将混合报文中的个人报文隔离出来形成两个独立的混合报文，一个用来加密，另一个直接传。例如，SDES 信息将会被加密，而接收报告则直接传，因为这样才能配合与密钥无关的第三方 monitor。例中，如 Fig。 4 所示，SDES 必须附加在一个没有报告（和随机号）的 RR 报文上， 来满足所有混合报文以 SR 或 RR 开始的要求。SDES CNAME item 或是出现在加密的或是不加密的，但不会再两者中都出现，因为这会危及到加密的安全。



#: SSRC identifier

Figure 4: Encrypted and non-encrypted RTCP packets

接收者通过头或 payload 有效性检查来确认加密及使用正确的密钥。附录 A.1 和 A.2 中有关于这方面的例子。

为了与早先的 RFC 1889 中关于 RTP 的描述一致，默认的加密算法是数据加密标准(DES)算法(the Data Encryption Standard (DES) algorithm in cipher block chaining CBC) mode)，如在 RFC1423[29]中第 1。1 节描述的那样，除了 8 字节倍数的填充需要像第 5.1 节描述的 P 位那样外。初始向量为零因为随机值是由 RTP 头或是混合 RTCP 报文中的随机前缀来提供的。关于 CBC 初始向量的使用详见[30]。

这所描述的支持加密方法的 Implementation 应支持 CBC 模式 DES 算法作为默认的密码来使互操作的最大化。选择这个方法是因为在互联网上使用试验的音视频工具是简单的和现实的。然而，也发现 DES 是很容易被攻击的。

建议使用更强的加密算法来代替默认算法，如 Triple-DES。此外，安全的 CBC 模式需要每一个报文的第一块与密码块等长的一个随机的独立的 IV 异或(XOR)。对于 RTCP，是为每一个报文随机选择一个 32-bit 数来完成的，对于 RTP，timestamp 和序列号是随机数生成的，而连续的报文就不会独立的随机化。需要注意的是两者的随机化都是有限的。更高安全的应用应该考虑其它的，更常规的保护方法。其它的加密算法可以通过 non-RTP 方法来动态的在一个会话内定义。特别是， 开发了基于 AES 的 SRTP profile[28]来考虑 known plaintext 和 CBC plaintext 操作，会是将来不错的选择。

如上所述，可以在 IP 级或是 RTP 级来选择加密，profile 可以为加密的编码定义附加的 payload 类型。这些编码必须说明怎样处理 padding 和加密的其它的方面。这个方法允许仅加密数据而不管头。这对处理解密和解码的硬件特别有用，同时对链路级的 RTP 压缩和需要的低层头也是有价值的，payload 的有效性也是足够的因为头加码排除了压缩。

9.2 鉴定和信息的完整性

鉴定和信息的完整性服务不是在 RTP 级定义的，因为如果没有一个关键的管

理基础，这些服务是不能直接可行的。希望由更低层的协议来提供鉴定和信息完整性服务。

10. 拥塞控制

所有在因特网上的应用得传输协议都需要某种方式的地址拥塞控制[31]，RTP 也不为例，但是因为通过 RTP 传输的数据经常是无弹性的（以一个固定的或是控制的速率生成），控制拥塞的方式与其它的像 TCP 传输协议的不同。从某种意义上说，，无弹性减少了拥塞的机率因为 RTP 流不会膨胀到像 TCP 流那样消耗所有可提供的带宽，然而，无弹性也意味着当拥塞发生时，RTP 流不能任意的在网络减少自己的负载来减小拥塞。

既然 RTP 会在不同背景下有很广泛的应用，就不会有单一的拥塞控制机制来解决所有问题。因此应在每个 RTP profile 中恰定的定义拥塞控制。对于一些 profile，在拥塞是由工程来避免的环境中，限制这个 profile 的使用，一个适用性声明就足够了，也可能需要像根据 RTCP 反馈的数据速率适应这样的特定方法。

11. RTP over 网络和传输协议

本章描述了关于在特定网络和传输协议下携带 RTP 报文的细节。下述规则是适用的除非被本文档外 protocol-specific 的定义 所取代。

RTP 依赖低层协议来提供 RTP 数据和 RTCP 控制流的解复用，对于 UDP 和相似的协议，RTP 应用一个偶数的端口号地址，相应的 RTCP 流应用下一个更高的(odd)目的地址端口号。对于那些单端口号作为参数，并从这个号得到 RTP 和 RTCP 端口对的应用来说，如果提供了一个奇号，那么应用应该用下一个低的（偶）号还代替那个号，作为基地址对。对于那些 RTP 和 RTCP 目的端口号是通过明确的独立的参数（利用一个信号协议或其它的方法）来描述的应用，应用可以忽略那些端口号的奇偶和连续的限制，虽然最好应用奇偶对。RTP 和 RTCP 端口号必须不同因为 RTP 依赖端口号来对 RTP 数据和 RTCP 控制流来解复用。

在单播会话中，两方的参与者需要标识一个端口对来接收 RTP 和 RTCP 报文。参与这可用同一端口对。参与者禁止把引入的(incoming)RTP 或 RTCP 报文的源端口设定为流出的(outgoning)RTP 或 RTCP 报文的目的地址。当 RTP 数据报文双向传输时，每一个参与者的 RTCP SR 报文必须被送到另一个参与者设定的接收 RTCP 的端口。RTCP SR 报文合并流出数据的发送者信息和引入数据的接收报告信息。如果一方不积极的发送数据（见 6.4）， 就发送 RTCP RR 报文。

建议层编码应用（见 2.4 节）使用一系列的连续的端口号。端口号必须能区分开来，因为操作系统的不足而阻止对多个多播地址使用相同的端口，而对单播

而言，仅有一个准许地址。因此对于层 n，数据端口是 P + 2n，控制端口是 P + 2n + 1。当使用了 IP 多播，也要将地址区分开来，因为多播路由和组成员是被一个地址间隔尺度间隔开来的。然而，不能保证分配连续的 IP 多播地址，因为某些组可能需要不同的范围(scope)，由此可能分配为不同的组地址范围中。

前一段与 RFC 2327[15]中定义的 SDP 向抵触，RFC 2327 指出在同一会话描述中同时定义多地址和多端口是非法的，因为端口与地址的联合是模糊的。希望在修订的 RFC 2327 中取消这种限制来允许一对一对应来定义一个对等的地址和端口号。

RTP 数据报文没有包含长度域或是其它的描绘(delineation)，因此 RTP 依赖低层协议提供长度标识。RTP 报文的最大长度仅由低层协议限制。

如果携带 RTP 报文的低层协议提供连续字节流的提取而不是消息(messages(packets))，必须定义一个 RTP 报文的封装来提供帧机制，如果低层协议含有 padding 的话也同样需要帧机制，这样就不能确定 RTP payload 的扩展了。帧机制不在此处定义。

即使携带 RTP 的协议为了允许一个低层的协议数据单元（例如 UDP 报文）携带多个 RTP 报文而定义了帧机制，profile 也会描述一个帧机制。在一个网络或传输报文中携带多个 RTP 报文降低了头花销，并使不同流的同步简单化。

12. 协议常量摘要

本章包含了在本文档中定义的常量的列表。

RTP payload type(PT)常量是在 profile 中定义的而不是本文档。然而，包含有 marker bit(s)和 payload type 的 RTP 头字段必须回避保留值 200 和 201（十进制），为目录 A.1 中所描述的头确认程序区分 RTP 和 RTCP SR 和 RR 报文类型。对于本文档中定义的标准的一个 marker bit 和一个 7-bit payload 类型域，这些限制意味着 payload 类型 72 和 73 是保留的。

12.1 RTCP 报文类型

缩略词	名字	值
SR	sender report	200
RR	receiver report	201
SDES	source description	202
BYE	goodbye	203
APP	application-defined	204

这些类型值是在 200-204 之间选择的，为了与 RTP 报文或其它的不相关的报文作对比而改进的 RTCP 报文头有效性检查。当 RTCP 报文类型域与相应的 RTP 头字段作对比，这个范围与 marker bit（经常不在数据报文中）为 1，标准的 payload type 域（因为静态的 payload 类型一般是在低位定义的）的最高位为 1 是相对应的。这个范围的选择要与 0 和 255 保持一段距离，通常全 0 或全 1 是通用数据模式。

因为所有混合的 RTCP 报文必须以 SR 或是 RR 开始，作为奇/偶对来选择这些码字以允许 RTCP 有效性检查测试有掩码和值得最大位数。

附加的 RTCP 报文可能在 IANA 注册（见第 15 章）。

12.2 SDES 类型

缩略词	名字	值
END	end of SDES list	0
CNAME	canonical name	1
NAME	user name	2
EMAIL	user's electronic mail address	3
PHONE	user's phone number	4
LOC	geographic user location	5
TOOL	name of application or tool	6
NOTE	notice about the source	7
PRIV	private extensions	8

附加的 RTCP 报文可能在 IANA 注册（见第 15 章）。

13. RTP Profile 和 Payload 格式说明

特定应用的 RTP 完整描述需要一个或多个这描述的两种类型文档：profiles，和 payload 格式说明。

RTP 可被应用于具有不同要求的不同应用中。适应这些不同要求的灵活性是通过在主协议中允许多选择，然后选择合适的或在单独的 profile 中为特定的环境和类应用定义扩展。典型的是在一个特定的会话中应用仅运行在一个 profile 下，所以在 RTP 控制里既不需要明确指出哪一个 profile 在使用。一个音视频应用的 profile 的例子可在 RFC 3551 中找到。Profile 典型的标题是"RTP Profile for ...."。

第二个伴随的文档类型是 payload 格式说明，定义了怎样在 RTP 中携带特定类的 payload 数据，例 H。261 编码的视频。这些文档典型的名字是"RTP Payload Format for XYZ Audio/Video Encoding"。Payload 格式在多 profile 下是有用的，因此也被独立的定义为任何的特定的 profile。 如果需要，profile 文档负责指派一个默认的映射到 payload 类型值。

本说明内，下面的条款是在一个 profile 下的可能定义，但是这个表不是详尽的：

RTP data header： 在 RTP 数据头中含有 marker bit 和 payload 类型域的字段，可能被 profile 重定义以适应不同要求，例如增加或减少 marker bit（见第 5.3 节）。

Payload types： 假定已经包含一个 payload 类型域，profile 通常会定义一系列的 payload 格式(例如媒体编码)和一个默认的将那些格式映射到 payload 格式的静态映射。一些 payload 格式是参考单单独的 payload 格式说明来定义的，profile 必须说明使用了 RTP timestamp 时钟速率（见第 5.1 节）。

RTP data header additions： 如果与 payload 类型独立的 profile 类的应用（见第 5.3 节）需要一些附加的功能，可将附加域添加在定长的 RTP 数据头中。

RTP data header extensions： 如果允许在 implementation-specific 扩展的 profile 下使用那个机制，RTP 数据头扩展结构的前 16 位就必须定义(见第 5.3.1 节)。

RTCP packet types： 新的 application-class-specific RTCP 报文类型必须定义和在 IANA 中注册。

RTCP report interval： profile 应说明使用了第 6.2 节中建议的值，这个常量是用来计算 RTCP 报告间隔的。包括会话带宽的 RTCP 部分，最小报告间隔，发送者和接收者带宽间隔。如果证明是工作在可变(scalable)方式下，profile 会制定预定(alternate)的值。

SR/RR extension： 如果需要附加的信息来规则的报告发送者或接收者，就需要为 RTCP SR 和 RR 报文定义扩展部分（见第 6。4。3 章）。

SDES use： profile 会为 RTCP SDES 传输或整体排除而制定相对的优先级（见第 6.3.9）；一个为 CNAME item 定义的预先的(alternate)的语义或语法（见第 6.5.1 节）；LOC item 的格式（见第 6.5.5 节）；语义和 NOTE item 的使用（见第 6.5.7）；或在 IANA 注册的新的 SDES item 类型。

Security： profile 指出应用应该提供那种安全服务和算法，并会为正确使用提供指导（见第 9 章）。

String-to-key mapping： profile 指出怎样将用户提供的密码或通过语应

设为加密的密钥。

Congestion： **profile** 应指定与那个 **profile** 适应的拥塞控制行为。

Underlying protocol： 需要另用一个特定的低层网络或传输层协议来携带 **RTP** 报文。

Transport mapping： 指定将 **RTP** 和 **RTCP** 应设为传输级地址的映射，例如，**UDP** 端口，不同于第 **11** 章定义的标准的映射。

Encapsulation：定义 **RTP** 报文封装来允许多个数据报文在一个低层的报文中携带或是为不这样做的低层协议提供帧机制（见第 **11** 章）。

不指望新的 **profile** 会是所有应用必须的。在一类应用中，最好是扩展一个已存在的 **profile** 而不是新定义一个来使应用之间的互操作更容易，因为每一个仅在一个 **profile** 下运行。像定义附加的 **payload** 类型值或 **RTCP** 报文类型这样简单的扩展可以通过向 **IANA** 注册和在 **profile** 中的附录或在 **payload** 格式说明中公布描述。

14. 安全考虑

RTP 与低层协议承受同样的安全 **liabilities**。例如，冒名者可以伪造源或目的网络地址，或改变头或载荷。在 **RTCP** 内，可以用 **CNAME** 和 **NAME** 信息模仿另一个参与者。除此之外，**RTP** 可以通过 **IP** 多播来发送，**IP** 多播不给一个发送者提供直接的方法来知道发送的数据的所有接收者，因此 **no measure of privacy**。正确与否**(Rightly or not)**，使用者对关系到音视频痛心的秘密比在传统的网络通信形式更敏感**[33]**。因此，**RTP** 安全机制的使用是很重要的。这些机制在第 **9** 章中讨论。

RTP 级的 **Translator** 或 **Mixer** 可以用来允许 **RTP traffic** 到达防火墙后面的主机。适当的防火墙安全原理和实践（不在本文档范围），应在设计和安装这些设备和准许进入防火墙后面的 **RTP** 应用时考虑。

15. IANA 考虑

附加的 **RTCP** 报文类型和 **SDES item** 类型通过 **IANA** 注册。因为这些号空间是小的，允许不受限的注册新值是不谨慎的。为了请求回顾的便利和为了促进在多应用中共享使用，要求新类型的注册必须在 **RFC** 文档备份（**document**）或其它的永久的容易得到的参考，像另一个合作标准主要部分的产物（例，**ITU-T**）。其它的要求在建议**"designated expert**。**"**下也可接受。**(**与 **IANA** 联系获得现在的专家的联系信息）

RTP profile 说明应以"RTP/xxx"格式为 profile 向 IANA 注册一个名字，其中 xxx 是 profile 标题的缩写。这些名字是为了高级控制协议的使用，例如会话描述协议(SDP)， RFC 2327[15]， 参考传输方法。

16. 知识产权声明

关于知识产权的合法性或范围，或是其它的被声明为本文档所描述的属于执行或使用的权利，或任何这样权利下的能或不能的可用到的范围， IETF 没有任何的立场。也不代表它可以为确认这样的权利作任何努力。在 BCP-11 中可以找到关于标准轨迹(standards-track)和标准相关(standard-related)文档的 IETF 进程的信息。权利声明的副本和任何许可的确认是开放给公众的，可以通过 IETF 秘书处获得关于工具或使用者使用本文档的知识产权的普通的执照或许可的尝试结果。

IETF 邀请任何有兴趣的组织提出由于本标准执行所用到的任何产权，专利，专利应用或是其它的 所有权的质疑。请与 IETF 行政主管说明相关的信息。

17. 致谢

本备忘录是由 IETF 音/视频传输工作组主席史蒂芬●卡斯纳(Stephen Casner)和克林●珀金斯(Colin Perkins)主持下讨论得来的。当前协议的起源是 Network Voice Protocol 和 the Packet Video Protocol (Danny Cohen and Randy Cole)合 the protocol implemented by the vat application (Van Jacobson and Steve McCanne)。 Christian Huitema 提供了随机 identifier 生成器的想法。timer reconsideration 算法的扩展分析和仿真是由 Jonathan Rosenberg 做的。层编码的增加物是由 Michael Speer 和 Steve McCanne 完成的。

Appendix A - Algorithms

We provide examples of C code for aspects of RTP sender and receiver algorithms. There may be other implementation methods that are faster in particular operating environments or have other advantages. These implementation notes are for informational purposes only and are meant to clarify the RTP specification.

The following definitions are used for all examples; for clarity and brevity, the structure definitions are only valid for 32-bit big-endian (most significant octet first) architectures. Bit fields are assumed to be packed tightly in big-endian bit order, with no additional padding. Modifications would be required to construct a portable implementation.

```
/*
 * rtp.h    --    RTP header file
 */
#include <sys/types.h>

/*
 * The type definitions below are valid for 32-bit architectures and
 * may have to be adjusted for 16- or 64-bit architectures.
 */
typedef unsigned char    u_int8;
typedef unsigned short u_int16;
typedef unsigned int      u_int32;
typedef                   short int16;

/*
 * Current protocol version.
 */
#define RTP_VERSION       2

#define RTP_SEQ_MOD (1<<16)
#define RTP_MAX_SDES 255          /* maximum text length for SDES */

typedef enum {
      RTCP_SR      = 200,
      RTCP_RR      = 201,
      RTCP_SDES = 202,
      RTCP_BYE    = 203,
      RTCP_APP    = 204
} rtcp_type_t;
```

```
typedef enum {
        RTCP_SDES_END     = 0,
        RTCP_SDES_CNAME = 1,
        RTCP_SDES_NAME    = 2,
        RTCP_SDES_EMAIL = 3,
        RTCP_SDES_PHONE = 4,
        RTCP_SDES_LOC     = 5,
        RTCP_SDES_TOOL    = 6,
        RTCP_SDES_NOTE    = 7,
        RTCP_SDES_PRIV    = 8
} rtcp_sdes_type_t;

/*
 * RTP data header
 */
typedef struct {
        unsigned int version:2;     /* protocol version */
        unsigned int p:1;             /* padding flag */
        unsigned int x:1;             /* header extension flag */
        unsigned int cc:4;            /* CSRC count */
        unsigned int m:1;             /* marker bit */
        unsigned int pt:7;            /* payload type */
        unsigned int seq:16;          /* sequence number */
        u_int32 ts;                   /* timestamp */
        u_int32 ssrc;                 /* synchronization source */
        u_int32 csrc[1];              /* optional CSRC list */
} rtp_hdr_t;

/*
 * RTCP common header word
 */
typedef struct {
        unsigned int version:2;     /* protocol version */
        unsigned int p:1;             /* padding flag */
        unsigned int count:5;         /* varies by packet type */
        unsigned int pt:8;            /* RTCP packet type */
        u_int16 length;               /* pkt len in words, w/o this word */
} rtcp_common_t;

/*
 * Big-endian mask for version, padding bit and packet type pair
 */
#define RTCP_VALID_MASK (0xc000 | 0x2000 | 0xfe)
#define RTCP_VALID_VALUE ((RTP_VERSION << 14) | RTCP_SR)
```

```
/*
 * Reception report block
 */
typedef struct {
        u_int32 ssrc;                   /* data source being reported */
        unsigned int fraction:8;    /* fraction lost since last SR/RR */
        int lost:24;                    /* cumul. no. pkts lost (signed!) */
        u_int32 last_seq;               /* extended last seq. no. received */
        u_int32 jitter;                 /* interarrival jitter */
        u_int32 lsr;                    /* last SR packet from this source */
        u_int32 dlsr;                   /* delay since last SR packet */
} rtcp_rr_t;

/*
 * SDES item
 */
typedef struct {
        u_int8 type;                    /* type of item (rtcp_sdes_type_t) */
        u_int8 length;                  /* length of item (in octets) */
        char data[1];                   /* text, not null-terminated */
} rtcp_sdes_item_t;

/*
 * One RTCP packet
 */
typedef struct {
        rtcp_common_t common;           /* common header */
        union {
                /* sender report (SR) */
                struct {
                        u_int32 ssrc;           /* sender generating this report */
                        u_int32 ntp_sec;    /* NTP timestamp */
                        u_int32 ntp_frac;
                        u_int32 rtp_ts;     /* RTP timestamp */
                        u_int32 psent;          /* packets sent */
                        u_int32 osent;          /* octets sent */
                        rtcp_rr_t rr[1];    /* variable-length list */
                } sr;

                /* reception report (RR) */
                struct {
                        u_int32 ssrc;           /* receiver generating this report */
                        rtcp_rr_t rr[1];    /* variable-length list */
                } rr;
```

```
            /* source description (SDES) */
            struct rtcp_sdes {
                u_int32 src;          /* first SSRC/CSRC */
                rtcp_sdes_item_t item[1]; /* list of SDES items */
            } sdes;

            /* BYE */
            struct {
                u_int32 src[1];     /* list of sources */
                /* can't express trailing text for reason */
            } bye;
        } r;
} rtcp_t;

typedef struct rtcp_sdes rtcp_sdes_t;

/*
 * Per-source state information
 */
typedef struct {
        u_int16 max_seq;            /* highest seq. number seen */
        u_int32 cycles;             /* shifted count of seq. number cycles */
        u_int32 base_seq;           /* base seq number */
        u_int32 bad_seq;            /* last 'bad' seq number + 1 */
        u_int32 probation;          /* sequ. packets till source is valid */
        u_int32 received;           /* packets received */
        u_int32 expected_prior; /* packet expected at last interval */
        u_int32 received_prior; /* packet received at last interval */
        u_int32 transit;            /* relative trans time for prev pkt */
        u_int32 jitter;             /* estimated jitter */
        /* ... */
} source;
```

A.1 RTP Data Header Validity Checks

An RTP receiver should check the validity of the RTP header on incoming packets since they might be encrypted or might be from a different application that happens to be misaddressed. Similarly, if encryption according to the method described in Section 9 is enabled, the header validity check is needed to verify that incoming packets have been correctly decrypted, although a failure of the header

validity check (e.g., unknown payload type) may not necessarily indicate decryption failure.

Only weak validity checks are possible on an RTP data packet from a source that has not been heard before:

- RTP version field must equal 2.
- The payload type must be known, and in particular it must not be equal to SR or RR.
- If the P bit is set, then the last octet of the packet must contain a valid octet count, in particular, less than the total packet length minus the header size.
- The X bit must be zero if the profile does not specify that the header extension mechanism may be used. Otherwise, the extension length field must be less than the total packet size minus the fixed header length and padding.
- The length of the packet must be consistent with CC and payload type (if payloads have a known length).

The last three checks are somewhat complex and not always possible, leaving only the first two which total just a few bits. If the SSRC identifier in the packet is one that has been received before, then the packet is probably valid and checking if the sequence number is in the expected range provides further validation. If the SSRC identifier has not been seen before, then data packets carrying that identifier may be considered invalid until a small number of them arrive with consecutive sequence numbers. Those invalid packets MAY be discarded or they MAY be stored and delivered once validation has been achieved if the resulting delay is acceptable.

The routine update_seq shown below ensures that a source is declared valid only after MIN_SEQUENTIAL packets have been received in sequence. It also validates the sequence number seq of a newly received packet and updates the sequence state for the packet's source in the structure to which s points.

When a new source is heard for the first time, that is, its SSRC identifier is not in the table (see Section 8.2), and the per-source state is allocated for it, s->probation is set to the number of sequential packets required before declaring a source valid (parameter MIN_SEQUENTIAL) and other variables are initialized:

```
    init_seq(s, seq);
    s->max_seq = seq - 1;
    s->probation = MIN_SEQUENTIAL;
```

A non-zero s->probation marks the source as not yet valid so the state may be discarded after a short timeout rather than a long one, as discussed in Section 6.2.1.

After a source is considered valid, the sequence number is considered valid if it is no more than MAX_DROPOUT ahead of s->max_seq nor more than MAX_MISORDER behind. If the new sequence number is ahead of max_seq modulo the RTP sequence number range (16 bits), but is smaller than max_seq, it has wrapped around and the (shifted) count of sequence number cycles is incremented. A value of one is returned to indicate a valid sequence number.

Otherwise, the value zero is returned to indicate that the validation failed, and the bad sequence number plus 1 is stored. If the next packet received carries the next higher sequence number, it is considered the valid start of a new packet sequence presumably caused by an extended dropout or a source restart. Since multiple complete sequence number cycles may have been missed, the packet loss statistics are reset.

Typical values for the parameters are shown, based on a maximum misordering time of 2 seconds at 50 packets/second and a maximum dropout of 1 minute. The dropout parameter MAX_DROPOUT should be a small fraction of the 16-bit sequence number space to give a reasonable probability that new sequence numbers after a restart will not fall in the acceptable range for sequence numbers from before the restart.

```
void init_seq(source *s, u_int16 seq)
{
      s->base_seq = seq;
      s->max_seq = seq;
      s->bad_seq = RTP_SEQ_MOD + 1;      /* so seq == bad_seq is false */
      s->cycles = 0;
      s->received = 0;
      s->received_prior = 0;
      s->expected_prior = 0;
      /* other initialization */
}
```

```
int update_seq(source *s, u_int16 seq)
{
        u_int16 udelta = seq - s->max_seq;
        const int MAX_DROPOUT = 3000;
        const int MAX_MISORDER = 100;
        const int MIN_SEQUENTIAL = 2;

        /*
         * Source is not valid until MIN_SEQUENTIAL packets with
         * sequential sequence numbers have been received.
         */
        if (s->probation) {
                /* packet is in sequence */
                if (seq == s->max_seq + 1) {
                        s->probation--;
                        s->max_seq = seq;
                        if (s->probation == 0) {
                                init_seq(s, seq);
                                s->received++;
                                return 1;
                        }
                } else {
                        s->probation = MIN_SEQUENTIAL - 1;
                        s->max_seq = seq;
                }
                return 0;
        } else if (udelta < MAX_DROPOUT) {
                /* in order, with permissible gap */
                if (seq < s->max_seq) {
                        /*
                         * Sequence number wrapped - count another 64K cycle.
                         */
                        s->cycles += RTP_SEQ_MOD;
                }
                s->max_seq = seq;
        } else if (udelta <= RTP_SEQ_MOD - MAX_MISORDER) {
                /* the sequence number made a very large jump */
                if (seq == s->bad_seq) {
                        /*
                         * Two sequential packets -- assume that the other side
                         * restarted without telling us so just re-sync
                         * (i.e., pretend this was the first packet).
                         */
                        init_seq(s, seq);
                }
```

```
            else {
                s->bad_seq = (seq + 1) & (RTP_SEQ_MOD-1);
                return 0;
            }
        } else {
            /* duplicate or reordered packet */
        }
        s->received++;
        return 1;
}
```

The validity check can be made stronger requiring more than two packets in sequence. The disadvantages are that a larger number of initial packets will be discarded (or delayed in a queue) and that high packet loss rates could prevent validation. However, because the RTCP header validation is relatively strong, if an RTCP packet is received from a source before the data packets, the count could be adjusted so that only two packets are required in sequence. If initial data loss for a few seconds can be tolerated, an application MAY choose to discard all data packets from a source until a valid RTCP packet has been received from that source.
Depending on the application and encoding, algorithms may exploit additional knowledge about the payload format for further validation. For payload types where the timestamp increment is the same for all packets, the timestamp values can be predicted from the previous packet received from the same source using the sequence number difference (assuming no change in payload type).

A strong "fast-path" check is possible since with high probability the first four octets in the header of a newly received RTP data packet will be just the same as that of the previous packet from the same SSRC except that the sequence number will have increased by one. Similarly, a single-entry cache may be used for faster SSRC lookups in applications where data is typically received from one source at a time.

A.2 RTCP Header Validity Checks

The following checks should be applied to RTCP packets.

- RTP version field must equal 2.

- The payload type field of the first RTCP packet in a compound packet must be equal to SR or RR.

- The padding bit (P) should be zero for the first packet of a compound RTCP packet because padding should only be applied, if it is needed, to the last packet.

- The length fields of the individual RTCP packets must add up to the overall length of the compound RTCP packet as received. This is a fairly strong check.

The code fragment below performs all of these checks. The packet type is not checked for subsequent packets since unknown packet types may be present and should be ignored.

```
u_int32 len;        /* length of compound RTCP packet in words */
rtcp_t *r;          /* RTCP header */
rtcp_t *end;        /* end of compound RTCP packet */

if ((*(u_int16 *)r & RTCP_VALID_MASK) != RTCP_VALID_VALUE) {
    /* something wrong with packet format */
}
end = (rtcp_t *)((u_int32 *)r + len);

do r = (rtcp_t *)((u_int32 *)r + r->common.length + 1);
while (r < end && r->common.version == 2);
if (r != end) {
    /* something wrong with packet format */
}
```

A.3 Determining Number of Packets Expected and Lost

In order to compute packet loss rates, the number of RTP packets expected and actually received from each source needs to be known, using per-source state information defined in struct source referenced via pointer s in the code below. The number of packets received is simply the count of packets as they arrive, including any late or duplicate packets. The number of packets expected can be computed by the receiver as the difference between the highest sequence number received (s->max_seq) and the first sequence number received (s->base_seq). Since the sequence number is only 16 bits and will wrap around, it is necessary to extend the highest sequence

number with the (shifted) count of sequence number wraparounds (s->cycles). Both the received packet count and the count of cycles are maintained the RTP header validity check routine in Appendix A.1.

```
extended_max = s->cycles + s->max_seq;
expected = extended_max - s->base_seq + 1;
```

The number of packets lost is defined to be the number of packets expected less the number of packets actually received:

```
lost = expected - s->received;
```

Since this signed number is carried in 24 bits, it should be clamped at 0x7fffff for positive loss or 0x800000 for negative loss rather than wrapping around.

The fraction of packets lost during the last reporting interval (since the previous SR or RR packet was sent) is calculated from differences in the expected and received packet counts across the interval, where expected_prior and received_prior are the values saved when the previous reception report was generated:

```
expected_interval = expected - s->expected_prior;
s->expected_prior = expected;
received_interval = s->received - s->received_prior;
s->received_prior = s->received;
lost_interval = expected_interval - received_interval;
if (expected_interval == 0 || lost_interval <= 0) fraction = 0;
else fraction = (lost_interval << 8) / expected_interval;
```

The resulting fraction is an 8-bit fixed point number with the binary point at the left edge.

A.4 Generating RTCP SDES Packets

This function builds one SDES chunk into buffer b composed of argc items supplied in arrays type, value and length. It returns a pointer to the next available location within b.

```
char *rtp_write_sdes(char *b, u_int32 src, int argc,
                     rtcp_sdes_type_t type[], char *value[],
                     int length[])
```

```
{
        rtcp_sdes_t *s = (rtcp_sdes_t *)b;
        rtcp_sdes_item_t *rsp;
        int i;
        int len;
        int pad;

        /* SSRC header */
        s->src = src;
        rsp = &s->item[0];

        /* SDES items */
        for (i = 0; i < argc; i++) {
                rsp->type = type[i];
                len = length[i];
                if (len > RTP_MAX_SDES) {
                        /* invalid length, may want to take other action */
                        len = RTP_MAX_SDES;
                }
                rsp->length = len;
                memcpy(rsp->data, value[i], len);
                rsp = (rtcp_sdes_item_t *)&rsp->data[len];
        }

        /* terminate with end marker and pad to next 4-octet boundary */
        len = ((char *) rsp) - b;
        pad = 4 - (len & 0x3);
        b = (char *) rsp;
        while (pad--) *b++ = RTCP_SDES_END;

        return b;
}
```

A.5 Parsing RTCP SDES Packets

This function parses an SDES packet, calling functions find_member() to find a pointer to the information for a session member given the SSRC identifier and member_sdes() to store the new SDES information for that member. This function expects a pointer to the header of the RTCP packet.

```
void rtp_read_sdes(rtcp_t *r)
```

```
{
        int count = r->common.count;
        rtcp_sdes_t *sd = &r->r.sdes;
        rtcp_sdes_item_t *rsp, *rspn;
        rtcp_sdes_item_t *end = (rtcp_sdes_item_t *)
                                        ((u_int32 *)r + r->common.length + 1);
        source *s;

        while (--count >= 0) {
                rsp = &sd->item[0];
                if (rsp >= end) break;
                s = find_member(sd->src);

                for (; rsp->type; rsp = rspn ) {
                        rspn = (rtcp_sdes_item_t *)((char*)rsp+rsp->length+2);
                        if (rspn >= end) {
                                rsp = rspn;
                                break;
                        }
                        member_sdes(s, rsp->type, rsp->data, rsp->length);
                }
                sd = (rtcp_sdes_t *)
                         ((u_int32 *)sd + (((char *)rsp - (char *)sd) >> 2)+1);
        }
        if (count >= 0) {
                /* invalid packet format */
        }
}
```

A.6 Generating a Random 32-bit Identifier

The following subroutine generates a random 32-bit identifier using the MD5 routines published in RFC 1321 [32]. The system routines may not be present on all operating systems, but they should serve as hints as to what kinds of information may be used. Other system calls that may be appropriate include

- getdomainname(),
- getwd(), or
- getrusage().

"Live" video or audio samples are also a good source of random numbers, but care must be taken to avoid using a turned-off microphone or blinded camera as a source [17].

Use of this or a similar routine is recommended to generate the initial seed for the random number generator producing the RTCP period (as shown in Appendix A.7), to generate the initial values for the sequence number and timestamp, and to generate SSRC values. Since this routine is likely to be CPU-intensive, its direct use to generate RTCP periods is inappropriate because predictability is not an issue. Note that this routine produces the same result on repeated calls until the value of the system clock changes unless different values are supplied for the type argument.

```
/*
 * Generate a random 32-bit quantity.
 */
#include <sys/types.h>      /* u_long */
#include <sys/time.h>       /* gettimeofday() */
#include <unistd.h>         /* get..() */
#include <stdio.h>          /* printf() */
#include <time.h>           /* clock() */
#include <sys/utsname.h> /* uname() */
#include "global.h"         /* from RFC 1321 */
#include "md5.h"            /* from RFC 1321 */

#define MD_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final

static u_long md_32(char *string, int length)
{
      MD_CTX context;
      union {
            char     c[16];
            u_long x[4];
      } digest;
      u_long r;
      int i;

      MDInit (&context);
      MDUpdate (&context, string, length);
      MDFinal ((unsigned char *)&digest, &context);
```

```
        r = 0;
        for (i = 0; i < 3; i++) {
                r ^= digest.x[i];
        }
        return r;
}                                       /* md_32 */

/*
 * Return random unsigned 32-bit quantity.  Use 'type' argument if
 * you need to generate several different values in close succession.
 */
u_int32 random32(int type)
{
        struct {
                int         type;
                struct    timeval tv;
                clock_t cpu;
                pid_t      pid;
                u_long    hid;
                uid_t      uid;
                gid_t      gid;
                struct    utsname name;
        } s;

        gettimeofday(&s.tv, 0);
        uname(&s.name);
        s.type = type;
        s.cpu  = clock();
        s.pid  = getpid();
        s.hid  = gethostid();
        s.uid  = getuid();
        s.gid  = getgid();
        /* also: system uptime */

        return md_32((char *)&s, sizeof(s));
}                                       /* random32 */
```

A.7 Computing the RTCP Transmission Interval

The following functions implement the RTCP transmission and reception rules described in Section 6.2. These rules are coded in several functions:

- rtcp_interval() computes the deterministic calculated interval, measured in seconds. The parameters are defined in Section 6.3.

- OnExpire() is called when the RTCP transmission timer expires.

- OnReceive() is called whenever an RTCP packet is received.

Both OnExpire() and OnReceive() have event e as an argument. This is the next scheduled event for that participant, either an RTCP report or a BYE packet. It is assumed that the following functions are available:

- Schedule(time t, event e) schedules an event e to occur at time t. When time t arrives, the function OnExpire is called with e as an argument.

- Reschedule(time t, event e) reschedules a previously scheduled event e for time t.

- SendRTCPReport(event e) sends an RTCP report.

- SendBYEPacket(event e) sends a BYE packet.

- TypeOfEvent(event e) returns EVENT_BYE if the event being processed is for a BYE packet to be sent, else it returns EVENT_REPORT.

- PacketType(p) returns PACKET_RTCP_REPORT if packet p is an RTCP report (not BYE), PACKET_BYE if its a BYE RTCP packet, and PACKET_RTP if its a regular RTP data packet.

- ReceivedPacketSize() and SentPacketSize() return the size of the referenced packet in octets.

- NewMember(p) returns a 1 if the participant who sent packet p is not currently in the member list, 0 otherwise. Note this function is not sufficient for a complete implementation because each CSRC identifier in an RTP packet and each SSRC in a BYE packet should be processed.

- NewSender(p) returns a 1 if the participant who sent packet p is not currently in the sender sublist of the member list, 0 otherwise.

- AddMember() and RemoveMember() to add and remove participants from the member list.

- AddSender() and RemoveSender() to add and remove participants from the sender sublist of the member list.

These functions would have to be extended for an implementation that allows the RTCP bandwidth fractions for senders and non-senders to be specified as explicit parameters rather than fixed values of 25% and 75%. The extended implementation of rtcp_interval() would need to avoid division by zero if one of the parameters was zero.

```
double rtcp_interval(int members,
                     int senders,
                     double rtcp_bw,
                     int we_sent,
                     double avg_rtcp_size,
                     int initial)
{
        /*
         * Minimum average time between RTCP packets from this site (in
         * seconds).  This time prevents the reports from `clumping' when
         * sessions are small and the law of large numbers isn't helping
         * to smooth out the traffic.  It also keeps the report interval
         * from becoming ridiculously small during transient outages like
         * a network partition.
         */
        double const RTCP_MIN_TIME = 5.;
        /*
         * Fraction of the RTCP bandwidth to be shared among active
         * senders.  (This fraction was chosen so that in a typical
         * session with one or two active senders, the computed report
         * time would be roughly equal to the minimum report time so that
         * we don't unnecessarily slow down receiver reports.)  The
         * receiver fraction must be 1 - the sender fraction.
         */
        double const RTCP_SENDER_BW_FRACTION = 0.25;
        double const RTCP_RCVR_BW_FRACTION =
(1-RTCP_SENDER_BW_FRACTION);
        /*
        /* To compensate for "timer reconsideration" converging to a
         * value below the intended average.
         */
        double const COMPENSATION = 2.71828 - 1.5;
```

```
double t;                       /* interval */
double rtcp_min_time = RTCP_MIN_TIME;
int n;                          /* no. of members for computation */

/*
 * Very first call at application start-up uses half the min
 * delay for quicker notification while still allowing some time
 * before reporting for randomization and to learn about other
 * sources so the report interval will converge to the correct
 * interval more quickly.
 */
if (initial) {
        rtcp_min_time /= 2;
}
/*
 * Dedicate a fraction of the RTCP bandwidth to senders unless
 * the number of senders is large enough that their share is
 * more than that fraction.
 */
n = members;
if (senders <= members * RTCP_SENDER_BW_FRACTION) {
        if (we_sent) {
                rtcp_bw *= RTCP_SENDER_BW_FRACTION;
                n = senders;
        } else {
                rtcp_bw *= RTCP_RCVR_BW_FRACTION;
                n -= senders;
        }
}

/*
 * The effective number of sites times the average packet size is
 * the total number of octets sent when each site sends a report.
 * Dividing this by the effective bandwidth gives the time
 * interval over which those packets must be sent in order to
 * meet the bandwidth target, with a minimum enforced.  In that
 * time interval we send one report so this time is also our
 * average time between reports.
 */
t = avg_rtcp_size * n / rtcp_bw;
if (t < rtcp_min_time) t = rtcp_min_time;

/*
 * To avoid traffic bursts from unintended synchronization with
 * other sites, we then pick our actual next report interval as a
```

```
         * random number uniformly distributed between 0.5*t and 1.5*t.
         */
        t = t * (drand48() + 0.5);
        t = t / COMPENSATION;
        return t;
}

void OnExpire(event e,
                        int       members,
                        int       senders,
                        double rtcp_bw,
                        int       we_sent,
                        double *avg_rtcp_size,
                        int       *initial,
                        time_tp      tc,
                        time_tp      *tp,
                        int       *pmembers)
{
        /* This function is responsible for deciding whether to send an
         * RTCP report or BYE packet now, or to reschedule transmission.
         * It is also responsible for updating the pmembers, initial, tp,
         * and avg_rtcp_size state variables.    This function should be
         * called upon expiration of the event timer used by Schedule().
         */

        double t;         /* Interval */
        double tn;        /* Next transmit time */

        /* In the case of a BYE, we use "timer reconsideration" to
         * reschedule the transmission of the BYE if necessary */

        if (TypeOfEvent(e) == EVENT_BYE) {
                t = rtcp_interval(members,
                                  senders,
                                  rtcp_bw,
                                  we_sent,
                                  *avg_rtcp_size,
                                  *initial);
                tn = *tp + t;
                if (tn <= tc) {
                        SendBYEPacket(e);
                        exit(1);
                } else {
                        Schedule(tn, e);
                }
```

```
        } else if (TypeOfEvent(e) == EVENT_REPORT) {
            t = rtcp_interval(members,
                              senders,
                              rtcp_bw,
                              we_sent,
                              *avg_rtcp_size,
                              *initial);
            tn = *tp + t;
            if (tn <= tc) {
                SendRTCPReport(e);
                *avg_rtcp_size = (1./16.)*SentPacketSize(e) +
                     (15./16.)*(*avg_rtcp_size);
                *tp = tc;

                /* We must redraw the interval.    Don't reuse the
                   one computed above, since its not actually
                   distributed the same, as we are conditioned
                   on it being small enough to cause a packet to
                   be sent */

                t = rtcp_interval(members,
                                  senders,
                                  rtcp_bw,
                                  we_sent,
                                  *avg_rtcp_size,
                                  *initial);

                Schedule(t+tc,e);
                *initial = 0;
            } else {
                Schedule(tn, e);
            }
            *pmembers = members;
        }
    }

    void OnReceive(packet p,
                   event e,
                   int *members,
                   int *pmembers,
                   int *senders,
                   double *avg_rtcp_size,
                   double *tp,
                   double tc,
```

```
                    double tn)
{
    /* What we do depends on whether we have left the group, and are
     * waiting to send a BYE (TypeOfEvent(e) == EVENT_BYE) or an RTCP
     * report.    p represents the packet that was just received.    */

    if (PacketType(p) == PACKET_RTCP_REPORT) {
        if (NewMember(p) && (TypeOfEvent(e) == EVENT_REPORT)) {
            AddMember(p);
            *members += 1;
        }
        *avg_rtcp_size = (1./16.)*ReceivedPacketSize(p) +
            (15./16.)*(*avg_rtcp_size);
    } else if (PacketType(p) == PACKET_RTP) {
        if (NewMember(p) && (TypeOfEvent(e) == EVENT_REPORT)) {
            AddMember(p);
            *members += 1;
        }
        if (NewSender(p) && (TypeOfEvent(e) == EVENT_REPORT)) {
            AddSender(p);
            *senders += 1;
        }
    } else if (PacketType(p) == PACKET_BYE) {
        *avg_rtcp_size = (1./16.)*ReceivedPacketSize(p) +
            (15./16.)*(*avg_rtcp_size);

        if (TypeOfEvent(e) == EVENT_REPORT) {
            if (NewSender(p) == FALSE) {
                RemoveSender(p);
                *senders -= 1;
            }

            if (NewMember(p) == FALSE) {
                RemoveMember(p);
                *members -= 1;
            }

            if (*members < *pmembers) {
                tn = tc +
                     (((double) *members)/(*pmembers))*(tn - tc);
                *tp = tc -
                     (((double) *members)/(*pmembers))*(tc - *tp);

                /* Reschedule the next report for time tn */
```

```
                    Reschedule(tn, e);
                    *pmembers = *members;
                }

            } else if (TypeOfEvent(e) == EVENT_BYE) {
                *members += 1;
            }
        }
    }
```

A.8 Estimating the Interarrival Jitter

The code fragments below implement the algorithm given in Section 6.4.1 for calculating an estimate of the statistical variance of the RTP data interarrival time to be inserted in the interarrival jitter field of reception reports. The inputs are r->ts, the timestamp from the incoming packet, and arrival, the current time in the same units. Here s points to state for the source; s->transit holds the relative transit time for the previous packet, and s->jitter holds the estimated jitter. The jitter field of the reception report is measured in timestamp units and expressed as an unsigned integer, but the jitter estimate is kept in a floating point. As each data packet arrives, the jitter estimate is updated:

```
int transit = arrival - r->ts;
int d = transit - s->transit;
s->transit = transit;
if (d < 0) d = -d;
s->jitter += (1./16.) * ((double)d - s->jitter);
```

When a reception report block (to which rr points) is generated for this member, the current jitter estimate is returned:

```
rr->jitter = (u_int32) s->jitter;
```

Alternatively, the jitter estimate can be kept as an integer, but scaled to reduce round-off error. The calculation is the same except for the last line:

```
s->jitter += d - ((s->jitter + 8) >> 4);
```

In this case, the estimate is sampled for the reception report as:

```
rr->jitter = s->jitter >> 4;
```

Appendix B - Changes from RFC 1889

Most of this RFC is identical to RFC 1889. There are no changes in the packet formats on the wire, only changes to the rules and algorithms governing how the protocol is used. The biggest change is an enhancement to the scalable timer algorithm for calculating when to send RTCP packets:

- The algorithm for calculating the RTCP transmission interval specified in Sections 6.2 and 6.3 and illustrated in Appendix A.7 is augmented to include "reconsideration" to minimize transmission in excess of the intended rate when many participants join a session simultaneously, and "reverse reconsideration" to reduce the incidence and duration of false participant timeouts when the number of participants drops rapidly. Reverse reconsideration is also used to possibly shorten the delay before sending RTCP SR when transitioning from passive receiver to active sender mode.

- Section 6.3.7 specifies new rules controlling when an RTCP BYE packet should be sent in order to avoid a flood of packets when many participants leave a session simultaneously.

- The requirement to retain state for inactive participants for a period long enough to span typical network partitions was removed from Section 6.2.1. In a session where many participants join for a brief time and fail to send BYE, this requirement would cause a significant overestimate of the number of participants. The reconsideration algorithm added in this revision compensates for the large number of new participants joining simultaneously when a partition heals.

It should be noted that these enhancements only have a significant effect when the number of session participants is large (thousands) and most of the participants join or leave at the same time. This makes testing in a live network difficult. However, the algorithm was subjected to a thorough analysis and simulation to verify its performance. Furthermore, the enhanced algorithm was designed to interoperate with the algorithm in RFC 1889 such that the degree of reduction in excess RTCP bandwidth during a step join is proportional

to the fraction of participants that implement the enhanced algorithm. Interoperation of the two algorithms has been verified experimentally on live networks.

Other functional changes were:

- Section 6.2.1 specifies that implementations may store only a sampling of the participants' SSRC identifiers to allow scaling to very large sessions. Algorithms are specified in RFC 2762 [21].

- In Section 6.2 it is specified that RTCP sender and non-sender bandwidths may be set as separate parameters of the session rather than a strict percentage of the session bandwidth, and may be set to zero. The requirement that RTCP was mandatory for RTP sessions using IP multicast was relaxed. However, a clarification was also added that turning off RTCP is NOT RECOMMENDED.

- In Sections 6.2, 6.3.1 and Appendix A.7, it is specified that the fraction of participants below which senders get dedicated RTCP bandwidth changes from the fixed 1/4 to a ratio based on the RTCP sender and non-sender bandwidth parameters when those are given. The condition that no bandwidth is dedicated to senders when there are no senders was removed since that is expected to be a transitory state. It also keeps non-senders from using sender RTCP bandwidth when that is not intended.

- Also in Section 6.2 it is specified that the minimum RTCP interval may be scaled to smaller values for high bandwidth sessions, and that the initial RTCP delay may be set to zero for unicast sessions.

- Timing out a participant is to be based on inactivity for a number of RTCP report intervals calculated using the receiver RTCP bandwidth fraction even for active senders.

- Sections 7.2 and 7.3 specify that translators and mixers should send BYE packets for the sources they are no longer forwarding.

- Rule changes for layered encodings are defined in Sections 2.4, 6.3.9, 8.3 and 11. In the last of these, it is noted that the address and port assignment rule conflicts with the SDP specification, RFC 2327 [15], but it is intended that this restriction will be relaxed in a revision of RFC 2327.

- The convention for using even/odd port pairs for RTP and RTCP in

Section 11 was clarified to refer to destination ports. The requirement to use an even/odd port pair was removed if the two ports are specified explicitly. For unicast RTP sessions, distinct port pairs may be used for the two ends (Sections 3, 7.1 and 11).

- A new Section 10 was added to explain the requirement for congestion control in applications using RTP.

- In Section 8.2, the requirement that a new SSRC identifier MUST be chosen whenever the source transport address is changed has been relaxed to say that a new SSRC identifier MAY be chosen. Correspondingly, it was clarified that an implementation MAY

  choose to keep packets from the new source address rather than the existing source address when an SSRC collision occurs between two other participants, and SHOULD do so for applications such as telephony in which some sources such as mobile entities may change addresses during the course of an RTP session.

- An indentation bug in the RFC 1889 printing of the pseudo-code for the collision detection and resolution algorithm in Section 8.2 has been corrected by translating the syntax to pseudo C language, and the algorithm has been modified to remove the restriction that both RTP and RTCP must be sent from the same source port number.

- The description of the padding mechanism for RTCP packets was clarified and it is specified that padding MUST only be applied to the last packet of a compound RTCP packet.

- In Section A.1, initialization of base_seq was corrected to be seq rather than seq - 1, and the text was corrected to say the bad sequence number plus 1 is stored. The initialization of max_seq and other variables for the algorithm was separated from the text to make clear that this initialization must be done in addition to calling the init_seq() function (and a few words lost in RFC 1889 when processing the document from source to output form were restored).

- Clamping of number of packets lost in Section A.3 was corrected to use both positive and negative limits.

- The specification of "relative" NTP timestamp in the RTCP SR section now defines these timestamps to be based on the most common system-specific clock, such as system uptime, rather than

on session elapsed time which would not be the same for multiple applications started on the same machine at different times.

Non-functional changes:

- It is specified that a receiver MUST ignore packets with payload types it does not understand.

- In Fig. 2, the floating point NTP timestamp value was corrected, some missing leading zeros were added in a hex number, and the UTC timezone was specified.

- The inconsequence of NTP timestamps wrapping around in the year 2036 is explained.

- The policy for registration of RTCP packet types and SDES types was clarified in a new Section 15, IANA Considerations. The suggestion that experimenters register the numbers they need and then unregister those which prove to be unneeded has been removed in favor of using APP and PRIV. Registration of profile names was also specified.

- The reference for the UTF-8 character set was changed from an X/Open Preliminary Specification to be RFC 2279.

- The reference for RFC 1597 was updated to RFC 1918 and the reference for RFC 2543 was updated to RFC 3261.

- The last paragraph of the introduction in RFC 1889, which cautioned implementors to limit deployment in the Internet, was removed because it was deemed no longer relevant.

- A non-normative note regarding the use of RTP with Source-Specific Multicast (SSM) was added in Section 6.

- The definition of "RTP session" in Section 3 was expanded to acknowledge that a single session may use multiple destination transport addresses (as was always the case for a translator or mixer) and to explain that the distinguishing feature of an RTP session is that each corresponds to a separate SSRC identifier space. A new definition of "multimedia session" was added to reduce confusion about the word "session".

- The meaning of "sampling instant" was explained in more detail as part of the definition of the timestamp field of the RTP header in

Section 5.1.

o Small clarifications of the text have been made in several places, some in response to questions from readers. In particular:

- In RFC 1889, the first five words of the second sentence of Section 2.2 were lost in processing the document from source to output form, but are now restored.

- A definition for "RTP media type" was added in Section 3 to allow the explanation of multiplexing RTP sessions in Section 5.2 to be more clear regarding the multiplexing of multiple media. That section also now explains that multiplexing multiple sources of the same medium based on SSRC identifiers may be appropriate and is the norm for multicast sessions.

- The definition for "non-RTP means" was expanded to include examples of other protocols constituting non-RTP means.

- The description of the session bandwidth parameter is expanded in Section 6.2, including a clarification that the control traffic bandwidth is in addition to the session bandwidth for the data traffic.

- The effect of varying packet duration on the jitter calculation was explained in Section 6.4.4.

- The method for terminating and padding a sequence of SDES items was clarified in Section 6.5.

- IPv6 address examples were added in the description of SDES CNAME in Section 6.5.1, and "example.com" was used in place of other example domain names.

- The Security section added a formal reference to IPSEC now that it is available, and says that the confidentiality method defined in this specification is primarily to codify existing practice. It is RECOMMENDED that stronger encryption algorithms such as Triple-DES be used in place of the default algorithm, and noted that the SRTP profile based on AES will be the correct choice in the future. A caution about the weakness of the RTP header as an initialization vector was added. It was also noted that payload-only encryption is necessary to allow for header compression.

- The method for partial encryption of RTCP was clarified; in particular, SDES CNAME is carried in only one part when the compound RTCP packet is split.

- It is clarified that only one compound RTCP packet should be sent per reporting interval and that if there are too many active sources for the reports to fit in the MTU, then a subset of the sources should be selected round-robin over multiple intervals.

- A note was added in Appendix A.1 that packets may be saved during RTP header validation and delivered upon success.

- Section 7.3 now explains that a mixer aggregating SDES packets uses more RTCP bandwidth due to longer packets, and a mixer passing through RTCP naturally sends packets at higher than the single source rate, but both behaviors are valid.

- Section 13 clarifies that an RTP application may use multiple profiles but typically only one in a given session.

- The terms MUST, SHOULD, MAY, etc. are used as defined in RFC 2119.

- The bibliography was divided into normative and informative references.

References

Normative References

[1] Schulzrinne, H. and S. Casner, "RTP Profile for Audio and Video Conferences with Minimal Control", RFC 3551, July 2003.

[2] Bradner, S., "Key Words for Use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[3] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[4] Mills, D., "Network Time Protocol (Version 3) Specification, Implementation and Analysis", RFC 1305, March 1992.

[5] Yergeau, F., "UTF-8, a Transformation Format of ISO 10646", RFC 2279, January 1998.

[6] Mockapetris, P., "Domain Names - Concepts and Facilities", STD 13, RFC 1034, November 1987.

[7] Mockapetris, P., "Domain Names - Implementation and Specification", STD 13, RFC 1035, November 1987.

[8] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989.

[9] Resnick, P., "Internet Message Format", RFC 2822, April 2001.

Informative References

[10] Clark, D. and D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in SIGCOMM Symposium on Communications Architectures and Protocols , (Philadelphia, Pennsylvania), pp. 200--208, IEEE Computer Communications Review, Vol. 20(4), September 1990.

[11] Schulzrinne, H., "Issues in designing a transport protocol for audio and video conferences and other multiparticipant real-time applications." expired Internet Draft, October 1993.

[12] Comer, D., Internetworking with TCP/IP , vol. 1. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[13] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[14] International Telecommunication Union, "Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service", Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, July 2003.

[15] Handley, M. and V. Jacobson, "SDP: Session Description Protocol", RFC 2327, April 1998.

[16] Schulzrinne, H., Rao, A. and R. Lanphier, "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998.

[17] Eastlake 3rd, D., Crocker, S. and J. Schiller, "Randomness Recommendations for Security", RFC 1750, December 1994.

[18] Bolot, J.-C., Turletti, T. and I. Wakeman, "Scalable Feedback Control for Multicast Video Distribution in the Internet", in SIGCOMM Symposium on Communications Architectures and Protocols, (London, England), pp. 58--67, ACM, August 1994.

[19] Busse, I., Deffner, B. and H. Schulzrinne, "Dynamic QoS Control of Multimedia Applications Based on RTP", Computer Communications , vol. 19, pp. 49--58, January 1996.

[20] Floyd, S. and V. Jacobson, "The Synchronization of Periodic Routing Messages", in SIGCOMM Symposium on Communications Architectures and Protocols (D. P. Sidhu, ed.), (San Francisco, California), pp. 33--44, ACM, September 1993. Also in [34].

[21] Rosenberg, J. and H. Schulzrinne, "Sampling of the Group Membership in RTP", RFC 2762, February 2000.

[22] Cadzow, J., Foundations of Digital Signal Processing and Data Analysis New York, New York: Macmillan, 1987.

[23] Hinden, R. and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003.

[24] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. and E. Lear, "Address Allocation for Private Internets", RFC 1918,

February 1996.

[25] Lear, E., Fair, E., Crocker, D. and T. Kessler, "Network 10 Considered Harmful (Some Practices Shouldn't be Codified)", RFC 1627, July 1994.

[26] Feller, W., An Introduction to Probability Theory and its Applications, vol. 1. New York, New York: John Wiley and Sons, third ed., 1968.

[27] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

[28] Baugher, M., Blom, R., Carrara, E., McGrew, D., Naslund, M., Norrman, K. and D. Oran, "Secure Real-time Transport Protocol", Work in Progress, April 2003.

[29] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III", RFC 1423, February 1993.

[30] Voydock, V. and S. Kent, "Security Mechanisms in High-Level Network Protocols", ACM Computing Surveys, vol. 15, pp. 135-171, June 1983.

[31] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, September 2000.

[32] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[33] Stubblebine, S., "Security Services for Multimedia Conferencing", in 16th National Computer Security Conference, (Baltimore, Maryland), pp. 391--395, September 1993.

[34] Floyd, S. and V. Jacobson, "The Synchronization of Periodic Routing Messages", IEEE/ACM Transactions on Networking, vol. 2, pp. 122--136, April 1994.

Authors' Addresses

Henning Schulzrinne
Department of Computer Science

Columbia University
1214 Amsterdam Avenue
New York, NY 10027
United States

EMail: schulzrinne@cs.columbia.edu

Stephen L. Casner
Packet Design
3400 Hillview Avenue, Building 3
Palo Alto, CA 94304
United States

EMail: casner@acm.org

Ron Frederick
Blue Coat Systems Inc.
650 Almanor Avenue
Sunnyvale, CA 94085
United States

EMail: ronf@bluecoat.com

Van Jacobson
Packet Design
3400 Hillview Avenue, Building 3
Palo Alto, CA 94304
United States

EMail: van@packetdesign.com

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.