oRTP 分析

岳维功

一：关于 oRTP

oRTP 是一款开源软件，实现了 RTP 与 RTCP 协议。

目前使用 oRTP 库的软件主要是 linphone（一款基于 IP 进行视频和语音通话的软件）。oRTP 作为 linphone 的 RTP 库，为基于 RTP 协议传输语音和视频数据提供保障。

二：源代码的构建框架

类似于 mediastream2 中的 filter，在 RTP 中也有比较重要的一个结构，就是 payload type，该结构用于指定编码类型，以及与其相关的时钟速率、采样率等一些参数，参见下图。

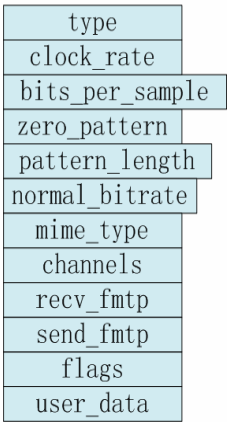

图 2-1

实际上在 RTP 的包头就有专门的域用来定义当前传输的数据是什么编码类型的。在代码中，不同的媒体类型有不同的 payloadtype 结构体与之对应，像 h263，g729，MPEG4 等。因为每种编码都有其独有的特点，而且许多参数也不一样，所以 RTP 包头中使用 payload 域标记负载的类型，一方面接收端可以就此判断负载的类型，从而选择对应的解码器进行解码播放；另一方面，代码在进行时间戳等方面的计算时可以更加方便一点。

Payloadtype 结构体定义了 payload 的许多属性，比如是音频还是视频数据，时钟采样率，每次采样的比特数，正常的比特率，MIME 类型，通道等等。代码中已有常见音视频编解码器对应的 payloadtype 结构体实现，应用程序在初始化 oRTP 库时，可以根据自己的需求，选择其中的一部分添加到系统中。所有系统当前支持的 payload 类型都被放在一个数组中，由全局变量 av_profile 这个结构体实例统领，如下图所示：

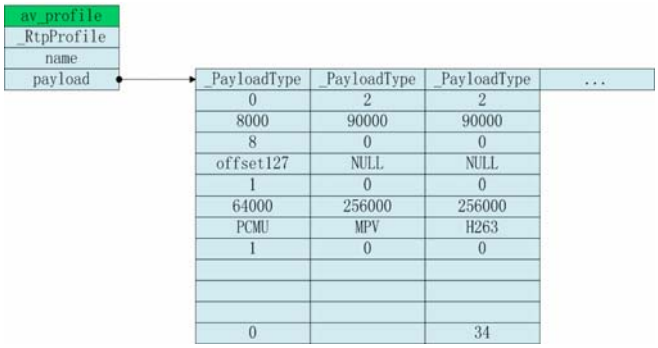

图 2-2

这些 payloadtype 结构体在 payload 数组中的位置就是以编码类型的定义为索引的。编码类型值的定义在 RFC3551 第六部分“payload type definitions”进行了描述。

Avprofile.c 文件定义了所有的 payload type。而有关 payload type 和 profile 的操作在文件 payloadtype.c 文件中实现。

除了 payloadtype 结构体外，一个更重要的结构体是 rtpsession。该结构体即是一个会话的抽象，与会话相关的各种信息都定义在该结构体上或者能够通过该结构体找到。要使用 oRTP 进行媒体数据的传输，需要先创建一个会话，之后所有数据的传输都在会话上完成或基于会话完成。rtpsession 结构体的定义如下：

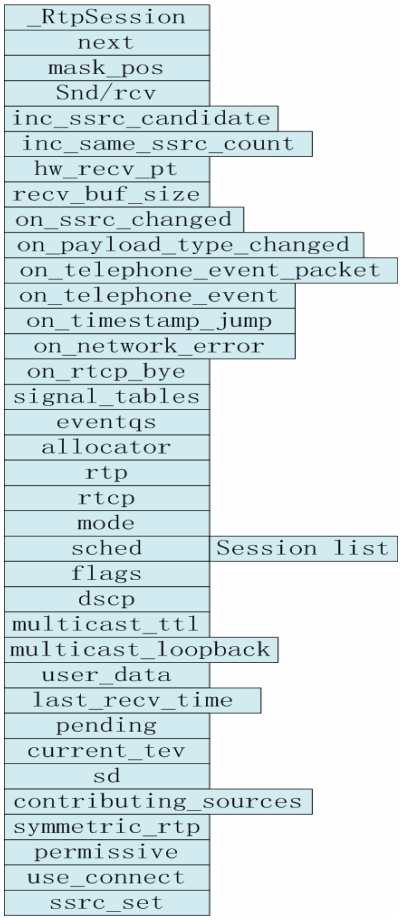

图 2-3

可以看到，这是一个非常大的结构体，从侧面说明了要维护的与会话相关的量还是比较多的。关于该结构体的比较详细的说明会在后面给出。

Session 的初始化通过接口 rtp_session_init 完成，外部获得一个新的 session 是通过调用接口 rtp_session_new 完成。关于 session 的其他有关配置和获取信息的操作都可以在文件 rtpsession.c 中找到定义。

使用 oRTP 进行数据传输时，可以在一个任务上完成多个会话流的接收和发送。这得益于 oRTP 中调度模块的支持。要使用调度模块，应用需要在进行 oRTP 的初始化时对调度进行初始化，将需要调度管理的会话注册到调度模块中，这样当进行接收和发送操作时，先向调度询问当前会话是否可以进行发送和接收，如果不能进行收发操作，则处理下一个会话。这有点类似 I/O 接口上的 select 操作。调度模块使用的数据结构主要为 rtpscheduler，如下图所示：

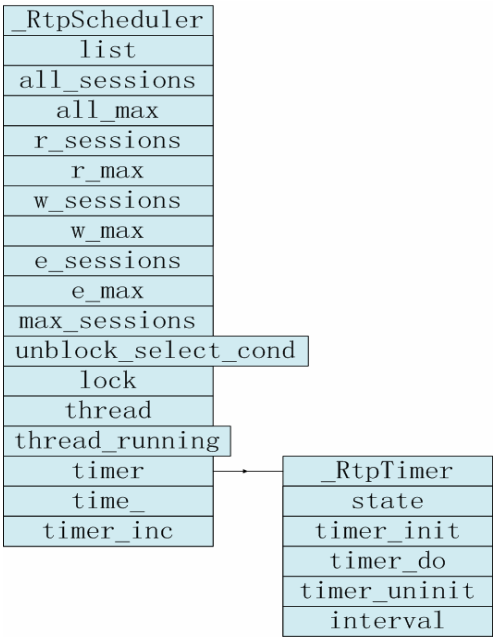

图 2-4

List 保存了所有要处理的会话，r\w\e 的意义类似于 select，在这里分别代表接收、发送以及异常。posixtimer.c，rtptimer.c，scheduler.c，sessionset.c 等文件实现了调度模块。

数据在底层实际的接收和发送是通过 socket 接口完成的，这些实现在 rtpsession_inet.c 文件中。

为了方便将 oRTP 移植到不同平台上，oRTP 实现了对操作系统接口的封装，包括常用的任务的创建及销毁，条件变量及互斥锁，进程间的管道通信机制等。这些在 port.c 文件中实现。

除了操作系统相关的接口外，oRTP 为了便于内部操作，实现了部分数据结构，一个是双向链表，在文件 utils.c 中；一个是队列，在文件 str_utilis.c 文件中。链表的实现比较简单，队列的实现相对较复杂一点。首先，队列数据结构由三部分组成：队列头、消息块以及数据块，图示如下：

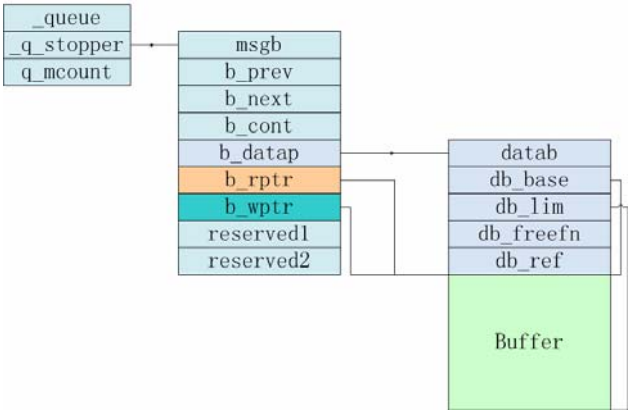

图 2-5

上图中从左到右依次为队列头，消息块和数据块。队列头指向消息块，消息块之间可以构成双向链表，这是队列的基本要素。消息块本身不带 buffer，数据是由专门的数据块来保存的，并被消息块指向。上图是一个初始化后的状态，消息块的读写指针都指向数据块的 buffer 的开始位置。数据块的 base 和 lim 指针则分别指向 buffer 空间的开始地址和结束地址处。向 buffer 中写入和读出数据后的状态变化如下图：

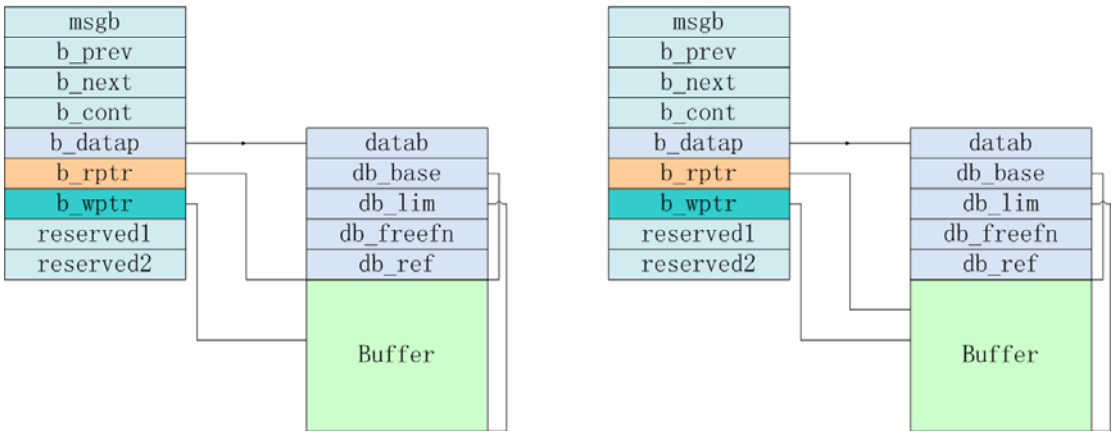

图 2-6

除了向队列添加消息块外，上述数据结构设计还支持向一个消息块添加新的消息块，这样可以支持一个消息块保存较大块的数据，如下图所示：

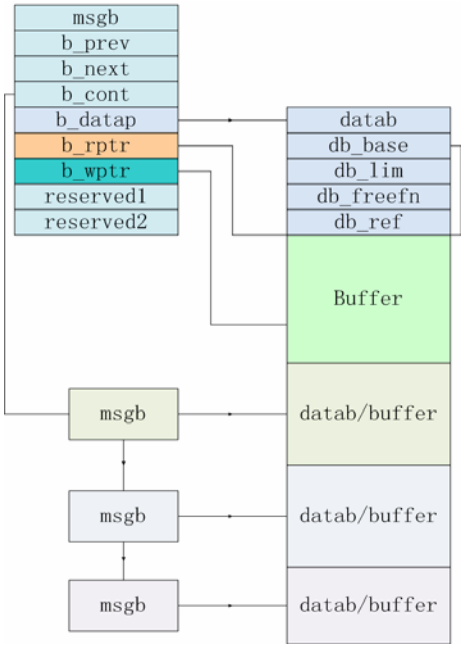

图 2-7

消息块的 b_cont 指针用于连接新的消息块。

在发送上层应用的 payload 数据之前，oRTP 会构造一个消息块，数据指针会指向 payload，这避免了数据拷贝。较低层的接口处理数据时依赖于消息块结构。接收后的数据从消息块中拷贝到用户 buffer。接收的 rtp 和 rtcp 包的解析处理函数在文件 rtpparse.c 和 rtcpparse.c 文件中实现。另外，rtcp.c 文件实现了 rtcp 数据包的构造处理。

在基于 ip 的音视频流传输中，防抖动能力是一个重要的特性，这在一定程度上能够保证用户有良好的体验。在 oRTP 中，是通过 jitter 模块完成这部分工作的。相关数据结构如下图所示：

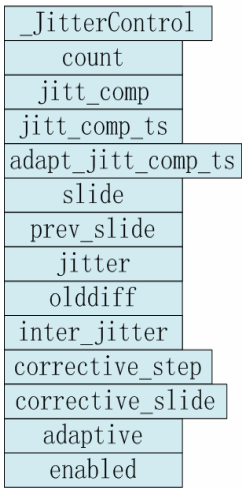

图 2-8

要使用 jitter 功能，需要使能 enabled 变量，如果要支持自适应补偿，则需要使能 adaptive 变量。

对于数据传输过程中产生的一些事件(比如 ssrc 发生改变,数据为 dtmf 数据等),在 oRTP 中是通过 signaltable（信号表）来处理的。signaltable 关联了事件类型与其上的回调函数。oRTP 使用 signaltable 处理如下一些事件:ssrc_changed(ssrc 发生改变),payload_type_changed（payload type 发生改变),telephone-event_packet(telephone event 包到达),telephone-event（telephone 事件),timestamp_jump（timestamp jump 事件),network_error（网络错误事件),以及 rtcp_bye（rtcp bye 包事件)。用户可针对这些事件注册回调处理函数，当底层接收函数接收到 rtp 包后，会对包进行检查，发现是上述某些事件的话，则触发回调函数的执行。rtpsignaltable.c 文件实现了对该表的操作，包括初始化，添加 callback 删除 callback 以及执行 callback。

oRTP 中对于事件的处理是基于事件结构体和事件队列的。队列用于存放事件结构体，结构体用于存放事件的数据。相关的处理在文件 event.c 中定义。特别的，对于 telephone 事件的处理放在 telephone_event.c 文件中，其中包括了如何构造用于传输 telephone_event 的 rtp 包，如何将 telephone 事件添加到包中，如何发送 dtmf 数据，以及接收到对应数据包后该如何处理。关于 telephone_event 的构成如下图所示：

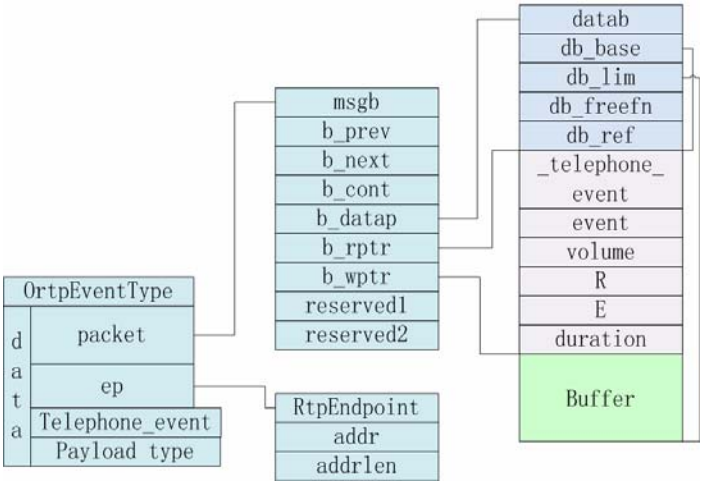

图 2-9

最左边的结构体是 rtp 包中存放的有关 telephone event 的数据，通过 packet 指针可以找到 telephone event 的详细信息。最终放入事件队列的也是 packet 指向的内容。

在使用 oRTP 提供的 rtp 库之前，需要先对其进行初始化，这部分的实现在 oRTP.c 文件中。oRTP 的初始化主要调用两个接口：ortp_init 和 ortp_scheduler_init。其中 ortp_init 完成了 payload 的注册，ortp_scheduler_init 完成了调度任务的初始化。

三：有关时间戳的说明

1 关于 RTP 传输中时间戳的说明（这部分来自于网络）

时间戳单位：RTP 协议中使用的时间戳，其单位不是秒之类的，而是以采样频率为基础的。这样做的目的就是为了使时间戳单位更为精准。比如说一个音频的采样频率为 8000Hz，那么我们可以把时间戳单位设为 1 / 8000。

时间戳增量：相邻两个 RTP 包之间的时间差（以时间戳单位为基准）。

采样频率：　每秒钟抽取样本的次数，例如音频的采样率一般为 8000Hz

帧率：　　每秒传输或者显示帧数，例如 25f/s

在 RTP 协议中并没有规定时间戳的粒度，这取决于有效载荷的类型。因此 RTP 的时间戳又称为媒体时间戳，以强调这种时间戳的粒度取决于信号的类型。例如，对于 8kHz 采样的话音信号，若每隔 20ms 构成一个数据块，则一个数据块中包含有 160 个样本（0.02×8000=160）。因此每发送一个 RTP 分组，其时间戳的值就增加 160。

如果采样频率为 90000Hz，则由上面讨论可知，时间戳单位为 1/90000，我们就假设 1s 钟被划分了 90000 个时间块，如果每秒发送 25 帧，那么，每一个帧的发送占多少个时间块呢？当然是 90000/25 = 3600。因此，我们根据定义“时间戳增量是发送第二个 RTP 包相距发送第一个 RTP 包时的时间间隔”，故时间戳增量应该为 3600。

关于 RTCP 中 NTP 时间戳的计算问题：从 1900 年到现在的经过的秒数赋值给 NTP 时间戳的高 32 位，这个时间的低 32 位通过当前获取的纳秒时间值计算得到。将 1 秒划分为 2 的 32 次方来表示，则一份子持续的时间大约位 232 皮秒。如果当前时间为 x 秒 232 毫秒，则 232 毫秒为 232000 微妙，232000000 纳秒，232000 000 000 皮秒，即 1000 000 000 多个 232 皮秒。也就是说在 NTP 时间戳的低 32 位划分的 2 的 32 次方个 232 皮秒块中占用了 1000 000 000 个块，转换为 16 进制表示为 3b9aca00，也就是说当当前时间的低位为 232 毫秒的话，NTP 时间戳的低 32 位就设置为 3b9aca00。

在 linux 系统中，我们常用的一个时间是 1970 年 1 月 1 日以来的时间所经过的秒数。在 RTCP 中，我们可以将当前所获得的上述时间加上 83AA7E80（十六进制）就是 1900 年 1 月 1 日以来所经过的秒数了。换为十进制，则为 2208988800。计算方法为（70 * 365 + 17) * 24 * 60 * 60。

2 代码中有关时间戳变量的说明

在数据的接收和发送过程中，用到了许多记录时间的变量。通过这些时间变量，oRTP 完成对 rtp 数据的流控功能。所有这些变量都定义在 rtpstream 结构体中，如下图所示：（这里只是截取了时间相关的变量）

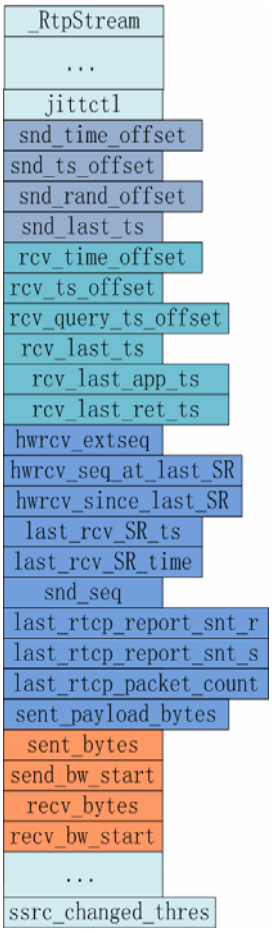

图 3-1

下面对这些变量的含义进行集中的说明：

uint32_t snd_time_offset;/*the scheduler time when the application send its first timestamp*/应用程序发送其第一个时间戳时的调度器时间

uint32_t snd_ts_offset;　/* the first application timestamp sent by the application */被应用程序发送的第一个应用程序时间戳

uint32_t snd_rand_offset; /* a random number added to the user offset to make the stream timestamp*/添加到用户 offset 上的一个随机数，用来产生流的时间戳

uint32_t snd_last_ts; /* the last stream timestamp sended */流上最后发送的时间戳

前述三个时间变量是 offset 结尾的，分别标记了第一个时间戳，包括调度器的时间偏移，在应用开始发送数据时，应用发送数据的时间偏移，也即是自己的时间戳，还有一个随机数用来添加到偏移上的，而第四个才是真正标记流里面当前最新发送的数据的时间戳。

uint32_t rcv_time_offset; /*the scheduler time when the application ask for its first timestamp*/应用程序询问其第一个时间戳时的调度时间，这里询问意指获取接收到的数据包—此应该指开始接收数据时的调度器时间

uint32_t rcv_ts_offset;　/* the first stream timestamp */第一个流的时间戳----此应该指第一个 rtp 包到来时其流上带的时间戳值

uint32_t rcv_query_ts_offset;　/* the first user timestamp asked by the application */被应用程序询问的第一个 user 时间戳—此应该指应用接收数据流时的时间

uint32_t rcv_last_ts; /* the last stream timestamp got by the application */应用程序得到的流

的最后一个时间戳—此应该指应用程序收到的最后一个 rtp 包的时间戳，是包里的时间戳值，而非应用自己的时间。

uint32_t rcv_last_app_ts; /* the last application timestamp asked by the application */被应用程序询问的最后一个应用程序时间戳—此处应该指应用收最后一个包时的应用时间，是应用按照 payload 类型及其采样率增长的时间戳记录，不是系统时间，也不是包里的时间

uint32_t rcv_last_ret_ts; /* the timestamp of the last sample returned (only for continuous audio)*/最后一个返回的采样的时间戳，仅仅对于连续的音频而言

接收相对于发送来讲存在一个问题，就是接收数据包时当前系统有个时间，数据包里面也有时间戳记录的时间，调度器也有记录时间。而对于发送，当前应用的时间就是给包的时间戳时间，这两个值对于发送来讲是一样的。

uint32_t hwrcv_extseq; /* last received on socket extended sequence number */在 socket 上最后接收的扩展的序列号

uint32_t hwrcv_seq_at_last_SR;每次发送报告包后，该变量更新为 hwrcv_extseq，因此是最近发送 rtcp 报告包时的最高扩展序列号。

uint32_t hwrcv_since_last_SR; 每收到一个 rtp 包，该变量加 1，在 rtcp 报告报构造好后，该变量就清为零，因此说明这个变量计数的是从上一个报告包以来接收的 rtp 包数目。

根据上面三个变量就可以计算出丢包率。首先，最近一次丢失包数（就是自从上一次 sr 或者 rr 发送以来）通过 hwrcv_extseq – hwrcv_seq_at_last_SR – hwrcv_since_last_SR 计算得到。但是丢包率为啥要除以 hwrcv_since_last_SR 比较奇怪。这个值是自从上一次发送报告包以来累计接收的包数。这个值不应该就是期望接收的包数。（最高序列号减去最初序列号）

累计包丢失数通过每次的丢包数累加得到。

uint32_t last_rcv_SR_ts; /* NTP timestamp (middle 32 bits) of last received SR */最后一个接收到的 sr 的 NTP 时间戳，取的是中间的 32bit。这个值也是报告包中上 LSR 值的来源。

struct timeval last_rcv_SR_time; /* time at which last SR was received */最后一个 sr 被接收到的时间，这个时间是用系统当前的时间来表示的。这个值记录了接收到最后一个 SR 时的系统时间，再发送当前报告包时，再次获取系统当前时间，然后二者相减，得到的值乘以 65536 得到以 1/65536 为单位的时间值。

uint16_t snd_seq; /* send sequence number */发送序列号。累加变量，保存会话的序列号的增长。

uint32_t last_rtcp_report_snt_r; /* the time of the last rtcp report sent, in recv timestamp unit */最后一个 rtcp 报告发送的时间，按照接收时间戳单位。程序中这个值是用 rcv_last_app_ts 变量的值来更新的。就是应用最后一次进行 rtp 接收时其时间戳增长到的值。不管收没收到就是这个值了？

uint32_t last_rtcp_report_snt_s; /* the time of the last rtcp report sent, in send timestamp unit */最后一个 rtcp 报告发送的时间，按照发送时间戳单位。程序中这个值是用 snd_last_ts 变量的值来更新的，就是应用最后一次进行 rtp 发送操作时其时间戳增长到的值。不管有没有发送 rtcp 报告包出去？

uint32_t rtcp_report_snt_interval; /* the interval in timestamp unit between rtcp report sent */按照时间戳单位表示的 rtcp 报告发送的间隔。这个值程序中使用默认时间值 5 秒与 payload 的 clockrate 的乘积来表示。是不是计算过于简单了？

uint32_t last_rtcp_packet_count; /*the sender's octet count in the last sent RTCP SR*/在最后发送的一个 rtcp sr 包中记录的发送者发送的 rtp 包总数。这个变量把这个值记录了下来。记

录这个值是为了实现协议中规定的：如果之前的 rtcp 包发送之后到当前再次发送 rtcp 包，这期间如果发送了 rtp 包，则发送 rtcp SR 报告包，否则只需发送 rtcp RR 包就可以了。

uint32_t sent_payload_bytes; /*used for RTCP sender reports*/用于 rtcp 发送者报告的 payload 字节数，数据来源。这个变量保存了从开始发送到发送这个 rtcp 报告包时发送的字节总数，但不包括头部和填充。

上面这些时间相关变量都是用于 rtcp 包的。

unsigned int sent_bytes; /* used for bandwidth estimation */用于带宽评估

struct timeval send_bw_start; /* used for bandwidth estimation */同上

上面两个变量用于计算发送带宽，start 记录的开始时间，sent_bytes 记录了发送的字节数，该值没调用 rtp 接口发送数据后都会进行累加更新。记录一次带宽值后，清为零，之后进行下一次带宽估计的计算。

unsigned int recv_bytes; /* used for bandwidth estimation */同上

struct timeval recv_bw_start; /* used for bandwidth estimation */同上

作用和处理逻辑都同上面发送部分。

四：调度的实现

要使用 oRTP 的调度功能，需要在初始化 oRTP 库时调用接口 ortp_scheduler_init 对调度模块进行初始化。在该接口中创建一个 RtpScheduler 类型的结构体__ortp_scheduler（参见图 2--4），并调用 rtp_scheduler_init 初始化它。

在 rtp_scheduler_init 中，分配定时器 posix_timer（rtptimer 类型结构体，参见图 2-4）挂载到调度结构体上。（定时器初始间隔设置为 POSIXTIMER_INTERVAL）。接着初始化 __ortp_scheduler 的其他部分，包括初始化互斥锁、条件变量等。在调度模块运行的整个过程中，相关操作都围绕该结构体，__ortp_scheduler 被定义为全局变量。

初始化完后调用 rtp_scheduler_start 启动调度任务。调度任务的执行体为 rtp_scheduler_schedule，参数为调度结构体自身。

调度任务执行后，首先初始化 timer。在这过程中将 timer 设置为运行状态，保存系统当前时间值。接着进入任务的 while 循环，遍历 scheduler 上注册的所有会话。如果不为空，说明应用有会话需要调度管理。此时会调用 rtp_session_process 进行处理。所有需要调度管理的会话按上述逻辑处理完之后，广播信号量 unblock_select_cond 唤醒所有因等待 select 而睡眠的任务，意即让这些任务去检查自己的会话是否需要进行处理了，这块后续还会说明。此时调度器完成了自己当前的工作开始准备进入睡眠状态，而其他的任务开始检查掩码结果以决定是需要进行数据的收发还是等待下次调度。

调度的睡眠是通过调用 timer 的 timer_do 接口来完成的，这里就是 posix_timer_do 接口。在该接口中，计算系统当前的时间，并和初始启动的时间（调度器初始化时保存）做差运算，结果转换为毫秒单位。posix_timer_time 记录了下一次调度器超时到达的时间，每次就让 posix_timer_time 减去系统当前时间与启动时间的差值，如果大于零，说明调度时间还没有到达，就调用 select 等待（posix_timer_time-差值）时间，然后重新获取系统当前时间，计算新的差值。流程图如下：

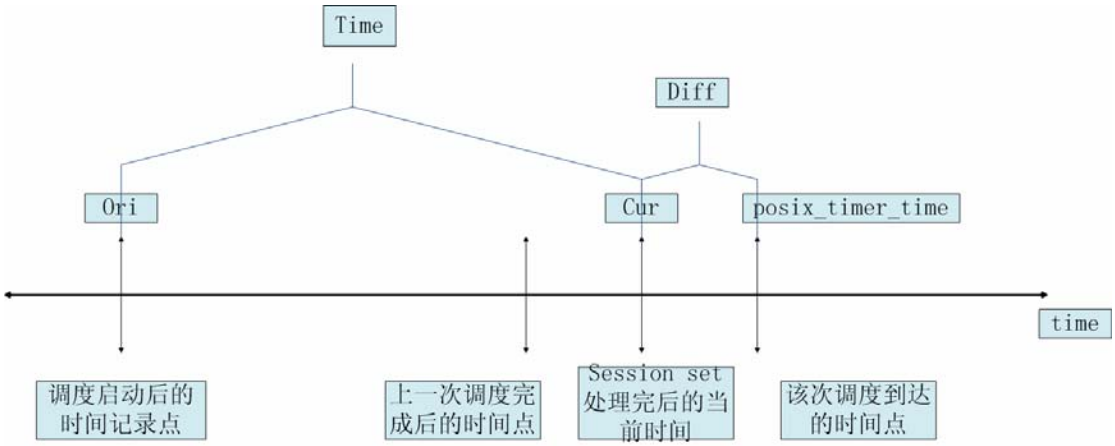

图 4-1

直观一点来说就是，调度器的调度精度由 POSIXTIMER_INTERVAL 确定，每次调度器运行，如果处理会话集合（session set）的时间超过该间隔，就会接着处理下次调度，如果没有用完，即剩余 diff 时间，这点时间就通过 select 系统调用耗掉。因此，调度器每次进行调度的时间点基本是确定的，diff 时间根据处理会话集合消耗时间的不同，每次的大小都是不一样的。

调度任务每次都基本上会在固定点检查所有需要由它来管理的会话，也就是应用添加到会话集合中的所有会话。如果在处理这些会话的过程中，时间超过了调度器设置的默认间隔，那么调度器处理完本次循环后会接着进行下一轮的循环，否则，会等待，直到下一个调度点时间到来。

调度器检查每个会话是通过 rtp_session_process 接口完成的。对于某一个会话，调用该接口将按如下逻辑进行处理：首先检查会话的发送部分的 waitpoint 结构体，将其时间与调度器当前时间进行比较（上述结构体中的时间是收发接口设置的需要唤醒的时间点）。如果该会话需要进行唤醒，也就是在等待唤醒，而且其等待的唤醒点也到了，（就是当前调度器时间已经超过了唤醒点）则清除需要进行唤醒的标识，然后在调度器结构体（调度器初始化时创建的全局变量）的 w_session 集合上将该会话的掩码位置置位，并通过条件变量唤醒该任务。同样的逻辑检查 r_session 集合。总的来看，调度器就是检查各个会话设置的唤醒点是否到了。如果到了则唤醒并设置其在集合中的掩码标志位。这样收发任务通过检查掩码标识位就知道是否可以继续进行收发了。一旦可以收发，应用会再次将这些掩码位置重新清除掉，这样在下次收发前就需要再次等待调度器进行检查并设置。

上层应用通过调用接口 rtp_session_set_scheduling_mode 将一个 session 添加到调度器中。添加过程为先获得调度器全局数据结构，给会话的 sched 指针，即该会话的 sched 指针指向全局调度器数据结构；会话 flags 添加 RTP_SESSION_SCHEDULED，意即让调度器管理会话；最后调用 rtp_scheduler_add_session 接口将会话添加注册到调度器管理的会话集合上。rtp_scheduler_add_session 接口中，先将会话挂到调度器数据结构的会话链表上（调度器每次循环时就从该链表上获取要处理的会话），然后在 all_sessions 集合中找到一个空闲位置，记录该掩码位置，将当前会话在该集合中的掩码位置进行置位操作。这样调度器通过会话链表就可以找到要调度的会话，进而找到会话上记录的掩码位置，从而在集合中对会话进行设置。类似的，将会话从集合中移除的接口为 rtp_scheduler_remove_session，基本处理逻辑就是找到会话列表中的该会话，将其从链表中移除，并把它在集合中的位置清零。

上层应用检查是否需要收发数据是通过检查会话集合来完成。首先，应用调用 session_set_new 接口创建一个新的集合。在该接口中我们创建一个 SessionSet 结构体并将其初始化，后续的操作就在该结构体上完成。对于需要调度的会话，调用接口 session_set_set

将其在该集合中的掩码位设置为 1，也就是打上标识。应用在每次接收或者发送前，调用接口 session_set_select 检查是否可以发送或者接收。该接口会将 caller 挂起直到有事件到达。session_set_select 类似我们常用的系统调用 select，其使用方式也是类似的。

Session_set_select 是应用与调度器打交道比较重要的一个接口，下面看看它的实现：

首先调用 ortp_get_scheduler 获取到调度器全局结构体
进入 while（1）循环
如果接收集合不为空，（也就是要检查是否有接收事件），
调用 session_set_init 初始化一个临时存放结果的集合
调用 session_set_and 检查会话集合。处理基于三个量，一个是初始化时添加到调度中进行接收检测的会话集合 r_sessions（这个集合代表调度器可以处理那些会话），一个是用户调用 select 时进行检查的会话集合，也就是应用要处理的集合（这个集合代表用户要处理那些会话），一个就是当前调度处理的会话集合的最大值 all_max（调度器从小到大检查到 all_max 位置就检查了其需要检查的所有会话掩码位）。在处理中，集合就是一个数组，数组每一个元素的每一个 bit 位代表了一个会话。这样，以 all_max 为上限，检查每一个会话对应的 bit 位，将调度器结构体上的接收集合和用户集合进行与运算（注意：这里接收集合是调度器处理完的，其中被设置的会话表明有接收事件到达。），得到的结果既是调度器处理后可以接收的会话，也是在应用环境中添加了的要处理的会话，记为 result set。同时将接收集合中被添加到 result 集合中的位清除（因为已经获取了）。最终 session_set_and 接口返回 result 集合中被设置的 bit 位数，也就是实际可以处理的会话个数。
如果有会话的事件到达，即返回值大于零，将新的结果拷贝回用户集合，告知用户那些会话有事件到达了。
对于发送和 error 集合做同样类似的处理
如果最终三个集合处理完后有事件（不管是接收还是发送还是 error），则直接返回，
否则在条件变量上等待，直到调度器返回有事件到达。
跳到 While（1）进行下次循环处理

除了 session_set_select 接口供用户调用外，oRTP 还提供了带有超时处理的 select 接口：session_set_timedselect，该接口可以设置跳出时间，而不是像 session_set_select 那样为死等模式。

综合应用和调度器两部分处理，可以看出，调度器的精度（调度间隔）在一定程度上可以影响数据接收速度。因为如果本次检查会话上不能进行收发数据操作，那么下次的检查就必须等到下个调度点，即使在当前检查刚过数据就到来了，应用也得等到下次调度点，由调度器检查后应用才能知道，而这段时间数据就必须等待。这样的话，如果调度间隔过大，那么接收速度必然减慢。

应用在收发数据时，除了可以使用调度器管理会话外，还可以设置阻塞与非阻塞模式。关于调度器与阻塞模式的关系：如果使用调度器，可以不用管阻塞模式，即调度器可以工作在阻塞模式下，也可以工作在非阻塞模式下。如果要使用阻塞模式，则需要启动调度器，这是必须的，即阻塞模式必须工作在调度器使用的情况下。（因为阻塞功能的实现本身就依赖于调度器）。对于调度器启动并且为非阻塞模式，当数据不能收发时，上层任务可以在应用层做其他操作来等待。对于调度器启动并设置为阻塞模式，当数据不能收发时，上层应用任务会等待条件变量，该条件变量只有等到调度器 signal 之后，上层任务才能继续运行。所以，如果上层应用启动了多个发送或者接收端口，那么非阻塞模式下有一个或多个端口不能发送

或者接收时，会尝试其他端口是否可以发送，如果都不能使用，则可以空循环。而阻塞模式下，如果有一个端口被阻塞了，那么其他端口都无法进行数据的收发了，即必须等待该端口有事件并被调度器触发后才有机会进行其他端口的发送或者接收。所以，在多接收发送应用情况下不应使用阻塞模式。

在非阻塞模式下，应用的等待时间消耗在 session_set_select 接口中了。阻塞模式下，应用可能就阻塞在发送接收接口中了。

使用目前的库，存在一个问题，在使用调度的情况下打开阻塞模式，则会导致程序挂住。具体原因分析来在于，阻塞模式下，包发送时其唤醒时间点 packet time 在调度器 scheduler time 后面了，这样调度器检查时就认为不需要进行唤醒，因为此时已经比调度器 old 了。根本原因在于阻塞时等待调度器运行，导致调度器时间超过了 packet time。而非阻塞模式下，包会直接发送出去，这样其实包的暂缓发送是在下次，也即是下次 select 等待时，调度器赶上包的发送时间，然后唤醒包发送，而阻塞模式下下次 select 时调度器已经赶上了并超过了包的发送时间。

关于调度器与应用的关系如下图所示：

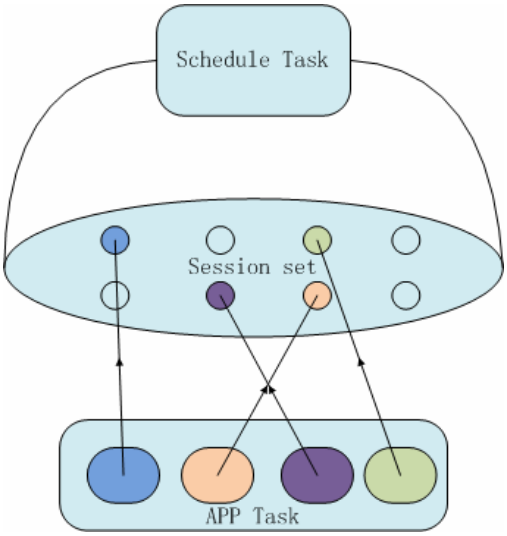

图 4-2

调度器检查 session set，唤醒到时间的接收流并设置掩码位。应用检查掩码位得到接收流是否被唤醒，然后进行接收处理，在接收处理中会清掉调度器设置的掩码位。

五：数据的接收和发送

1 发送过程：

应用发送数据时调用接口 rtp_session_send_with_ts 完成。参数为会话句柄，数据缓冲区地址，数据长度以及应用当前的时间戳。在该接口中，会先调用 rtp_session_create_packet 接口，根据缓冲区地址及数据长度，构造一个新的消息块，并根据会话信息初始化 rtp 头信息。完了将缓冲区中的数据拷贝到消息块中。最后以消息块为参数，调用 rtp_session_sendm_with_ts 接口进行数据发送。rtp_session_sendm_with_ts 调用更底层的函数 __rtp_session_sendm_with_ts，在该函数中完成具体的发送处理。下面具体分析该函数的实现：

如果发送还没有启动，也就是说当前是第一次启动，则 snd_ts_offset 变量首先被设置为应用当前开始的值。如果启动了调度，则 snd_time_offset 设置为调度器运行到现在的时间。这应该算是时间戳的初始化。

如果调度被启用了，则对部分时间戳做一些调整，如下：

首先计算包应该发送的时间，就是 packet time。计算方法为在发送第一个数据包时的调度器时间加上包的发送间隔，这个间隔根据应用当前给的时间与第一次的发送给的时间的差值除以 payload 的时钟速率计算得到，比如第一次发送的时间为 100，当前为 300，也就是说发送经过了 200 个单位，如果 payload 的 clock rate 为 10，则说明经过了 20 个时间戳单位，也就是说当前包的时间戳为调度器时间加 20。（packet time 实际上应该是将下一个包的发送时间转换为调度器时间，交给调度器让调度器来调度）如果计算的 packet time 与调度器当前的运行时间的差值小于 2 的 31 此方，并且二者不相等，则设置该等待点在 packet time 唤醒。（关于该比较，参见其他说明部分）

在发送数据前，RTP 的时间戳设置为应用传进来的当前的时间戳。snd_last_ts 时间戳也设置为应用当前给的时间戳。

之后就调用实际的发送接口 rtp_session_rtp_send 进行发送。该接口具体会调用 send 系统调用将数据包发送到网络的另一端。

发送完成后调用 rtp_session_rtcp_process_send 查看是否需要发送 rtcp 包，依据的原则是：

如果由应用程序询问的最后的时间戳减去以接收单位计算的最后一个 rtcp 包发送的时间大于 rtcp 报告包应该发送的时间间隔，或者最后发送数据包的时间戳与按照发送时间戳单位计算的最后一个 rtcp 报告包发送的时间的差值大于 rtcp 应该发送的间隔，就构造 rtcp 的发送者报告包发送。

在构造 rtcp 控制包的过程中，ssrc 源同步描述符采用 session 上的源同步描述信息，NTP 时间戳使用系统当前的时间加上 1900 到 1970 年间的秒数，实际上这个时间就是 1900 年当当前的秒数了（参见时间戳说明部分）。RTP 时间戳使用 snd_last_ts，也就是最后发送的流的时间戳。发送的包数和包字节计数使用 session 上 RTP 流上统计的计数。另外，如果数据包有被收到，则包含一个报告块，目前的设计也仅只包含一个报告块。数据包构造完成后直接发送。

如果会话当前的模式为 send-only，则调用 rtp_session_rtcp_recv 接收处理 rtcp 包。如果会话支持接收模式，则 rtcp 包的接收会在 rtp 接收过程中处理。

2 接收过程。

数据包的接收是通过调用 rtp_session_recv_with_ts 接口完成的。该接口实际上是调用 rtp_session_recvm_with_ts 从底层接收数据，将返回的消息块中的有效数据（不包含 rtp 头）拷贝到用户的 buffer 中。下面具体看 rtp_session_recvm_with_ts 的实现：

如果接收还没有启动，rcv_query_ts_offset 设置为应用给定的初始时间，也就是应用询问的时间，记录了一个开始时间偏移。如果发送没有启动或者为 recv-only 模式，则 session 的 last_rcv_time 设置为系统当前的时间。如果设置了调度器，那么 rcv_time_offset 设置为调度器启动后运行到当前所用的时间，这个作为接收的时间偏移。如果接收已经启动了，为了避免针对同一个时间戳连续多次接收，这里判断如果当前应用参数给的时间等于 rcv_last_app_ts 也即应用程序最近一次询问的时间戳，那么 read_socket 变量设置为 FALSE，避免连续接收。

接下来进入正常的处理流程，首先将 rcv_last_app_ts 设置为当前应用时间，也就是更新当前最后一次接收的时间。如果 read_socket 设置了，调用 rtp_session_rtp_recv 和 rtp_session_rtcp_recv 接口实际的从底层 socket 接收数据。

在 rtp_session_rtp_recv 中接收到数据后会调用 rtp_session_rtp_parse 对数据包进行解析。在 rtp_session_rtp_parse 中如果发现数据包是 telephone event 包，则会创建一个事件，将其发送到事件队列上，具体处理参见事件部分的说明。Jitter 中相关变量的更新也是在该接口中进行处理的，通过调用 jitter_control_new_packet 接口完成。最后将数据包放到接收队列上等

待进一步的处理。

从 rtp_session_rtp_recv 出来后，会检查会话的 telephone event 队列，如果不为空，则说明收到了拨号包，一方面需要调用注册的回调函数，另一方面则需要将其发送给事件队列。之后接收就返回了。如果该队列上没有包，则继续处理：

如果设置了接收同步标识，rcv_ts_offset 被设置为当前收到的 RTP 数据包中的时间戳。这作为流的第一个时间戳。rcv_last_ret_ts 变量则设置为当前应用给出的时间。这里仅仅是给一个初始的值。之后清掉同步标识。因此之前的偏移 rcv_ts_offset 记录了第一个 rtp 数据包的时间戳。后续到达的数据包将不再经过这里的处理逻辑。

调用接口 jitter_control_get_compensated_timestamp 计算流的时间戳。具体参见 jitter 模块说明。 如果 rtp 上的 jitter 控制是使能的，那么就会利用 jitter buffer 机制对数据包进行流控，否则，就直接从队列上取一个新的数据包。在 jitter 使能的情况下，如果 session 的 permissive 算法被启用了，那么就调用 rtp_getq_permissive 接口获取数据。在该接口中，判断如果计算出的流的时间戳与 rtp 数据包中记录的时间戳的差值小于 2 的 31 次方，就从队列中弹出一个包返回，否则返回空。如果没有启用 permissive 算法则调用 rtp_getq 接口按照正常方式接收数据包。在该接口中，我们返回时间戳等于或者早于计算的时间戳的数据包，如果这样的数据包不止一个，那么扔掉更老的包，也就是从队列上最先取出来的包，最后返回的就是最近一次取出的数据包。如果有两个数据包有相同的时间戳，那么只返回一个。另外，在该接口中如果有数据包也就是更老的包被丢弃了，那么会把丢弃的包数目记载到 reject 参数中返回。

如果上一步确有数据包返回，那么会对数据包中的时间戳进行更新，这部分参见对 jitter_control_update_corrective_slide 接口的说明。随后将 rcv_last_ts 时间戳更新为包原始到达时的时间戳值，即未更新前的值。接着调用 rtp_session_rtcp_process_recv 接口进行 rtcp 的接收处理。（之前是发送处理）触发条件和触发后时间量的修改同发送部分。如果最后一次 rtcp 的 sr 报告中的发送计数小于统计量中的发送包数的统计，则调用 make_sr 构造 sr 报告包，同时将之前的统计计数更新为统计量中保存的值。如果该值不小于，则说明不需要发送 rtcp 的 sr 报告包，但是如果同时接收的包数大于零，就是说有数据包被接收到，则调用 make_rr 构造 rtcp 的 rr 包。如果包构造成功，则调用 rtp_session_rtcp_send 发送包。

之后如果没有启动调度，则直接将包返回给上层，不需要再进行特殊处理，否则进行调度的处理。类似与发送部分，同样是根据应用给定的时间和应用第一次调用接收时的时间差值作为参数，调用 rtp_session_ts_to_time 接口计算出包的调度时间间隔。这个间隔加上应用询问第一个包时调度器运行的时间作为包的下次调度时间。如果这个时间在调度器当前的时间之后，则就将这个时间作为唤醒点，等待调度器调度。

接收和发送过程中各个时间戳值的关系如下图所示：

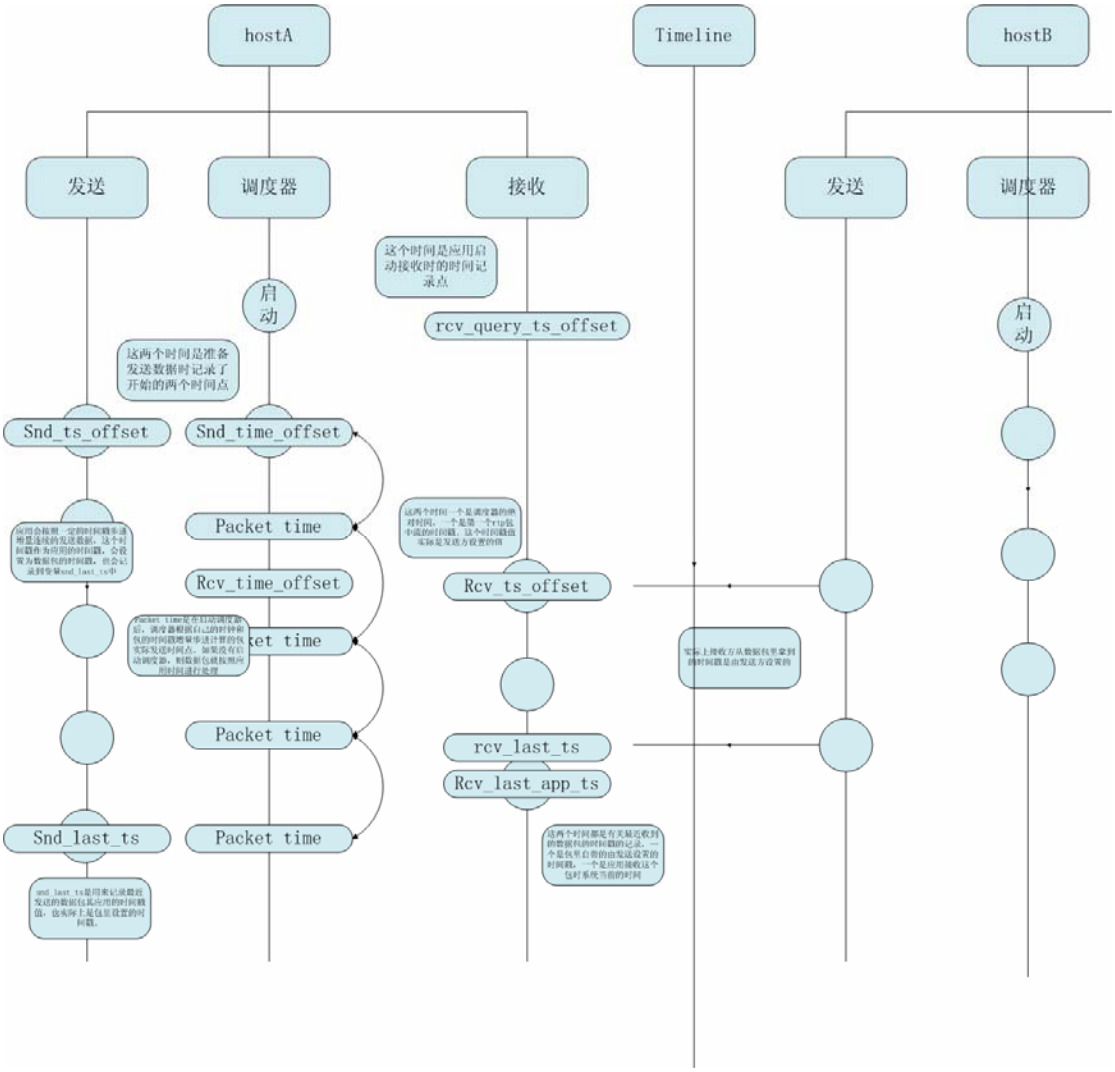

图 5-1

六：防抖动的实现

关于 jitter 结构体中部分变量的说明：（关于该结构体参见图 2-8）

其中 jitt_comp 为用户定义的防抖动补偿时间，jitt_comp_ts 为将其转化为时间戳单位的值，adapt_jitt_comp_ts 为使用自适应算法计算后的补偿时间值。

slide 为包期望接收时间和应用接收时间的差值的平均值，prev_slide 为上一次保存的 slide 值。

jitter 为 diff（最新得到的包的时间戳值与本地接收时间值的差，用于计算 slide）与更新后的 slide 的差值的平均值，olddiff 为上一次计算出的 diff 值，inter_jitter 为间隔抖动，参见 rtcp 协议（参考 1）。

corrective_step 和 corrective_slide 为校正步进值和校正滑动值，在更新包里带的时间戳时会用到。

adaptive 和 enabled 在介绍 jitter 结构体时已做说明。

在会话初始化的时候，会调用 rtp_session_set_jitter_buffer_params，该接口设置 jitter buffer 的参数。代码中实际上将默认的 jitter 时间设置为了 80 毫秒，也就是四个数据块的间隔（针对 8KHZ 音频采样数据而言），输入数据包的队列长度设置为了 100，也就是可以缓冲 100 个数据包。同时，打开了 jitter 的自适应（adaptive）特性，也就是 jitter 自适应补偿（adaptive

compensation）。

在实际中，用户也可以单独调用 rtp_session_set_jitter_compensation 设置 jitter 补偿时间，可以调用 rtp_session_enable_adaptive_jitter_compensation 单独设置是否打开自适应补偿功能。

在设置 jitter buffer 的时候，会调用接口 jitter_control_init 完成 jitter 的初始化。在该接口中，jitt_comp 设置为用户设置的值，该值就是补偿值。另外，调用 jitter_control_set_payload 将该补偿值转换为时间戳单位的值，设置给 jitter_comp_ts，转换依赖于 payload 的时钟采样率。校正步进值（corrective_step）设置为(160 * 8000)/pt->clock_rate;大部分音频采样率都是 8KHZ，所以应该是按照 160 的时间戳单位来校正。

要使用 jitter，需要使能 enabled 变量，要使用 adaptive，需要打开 adaptive 变量。

数据发送过程不需要 jitter 做什么控制，关键是在接收中。数据接收完后并不是直接交给上层应用，而是放到 buffer 中，其实就是队列。Buffer 的大小在 jitter 初始化部分设置，默认为 100（队列的长度），也就是可以缓冲 100 个包，这对一秒钟动辄百十来个网络包的媒体流来讲，其实也缓冲不了多少数据。另外，接收到的包只要解析通过，都先缓冲到队列中，如果包数目超过了队列大小，则移除最老的包，这也符合常理。后续为应用传递的包都是从队列上取出来的，所以取的也就是最老的包。在数据包是否需要取出来上传给应用就需要 jitter 来控制了。

对于已经缓冲到本地的数据包，没有 jitter buffer 控制的情况下我们直接将其返回，如果有控制，则需要判断包的时间戳，只有比给定时间戳老的包（早于给定时间戳到达）才上传给应用。那么这里控制包是否上传给应用，关键的因素就在于给定的时间戳值，这个值是怎么来的呢？在程序中，通过调用 jitter_control_get_compensated_timestamp 接口计算得到。基本计算式为：

Ts = user_ts + slide –adapt_jitt_comp_ts

为了更好的理解上面的计算式，我们来看上述几个值是如何计算出来的。首先，user_ts，这是应用程序接收流时给出的时间戳，是基于应用接收的速度和 payload 类型计算出来的，典型的，对于采样率为 8KHZ 的音频数据来说，该时间戳的增加步进值为 160.(按照每 20 毫秒采样一次来算)。如果网络传输没有延迟，数据包处理不需要消耗时间，那么 user_ts 应该和 rtp 包里带的时间戳值是一致的。

slide，根据前面的介绍，为数据包期望接收的时间和实际本地接收的时间差值的平均值。每次接收到新的 rtp 包后该值即进行更新，新的 slide 值为之前值乘以 0.99 加上新计算的值乘以 0.01。

adapt_jitt_comp_ts 的计算依赖于 jitter 的值。Jitter 按前面介绍，为 diff 与更新后的 slide 的差值的平均值。同样是已计算得到的值乘以 0.99 加上新的到的值乘以 0.01 得到。如果数据包均匀到达，那么 diff 的值和 slide 的值应该就是相等的，这样 jitter 的值就为零。相反，如果 jitter 的值变化比较大，那么说明数据包每次到来的间隔参差不一，一定程度上反映抖动比较大。

inter_jitter 为间隔抖动，含义和计算参见 rtcp 发送间隔分析（参考 3）。从上面计算可以看出，inter-jitter 反映了两次间隔的抖动情况，而 slide 则反映了一个比较长期的均匀的抖动情况。

如果打开了 adaptive，slide 的值就会不断更新，并且每当收到 50 个包后，adapt_jitt_comp_ts 就会被更新。新值为 jitt_comp_ts 和 2*jitter 中较大的一个。

上述计算过程可参见接口 jitter_control_new_packet。

现在回过头来再看上面的计算式，slide 实际上是个小于零的值，因此我们实际上是努力将时间戳靠近包里自带的时间戳值上去的，补偿值在一定程度上起到了缓冲的作用。

关于数据包时间戳值的更新，如果从队列中取出了数据包，并且当前数据包的时间戳值与之前收到的数据包的时间戳值不一致，在打开 adaptive 的情况下，将对数据包的时间戳值进行校正，算法如下：

当前 slide 减去之前的 slide，如果差值大于校正步进值 correction_step，则校正滑动值 correction_slide 加上一个 correction_setp，prev_slide 更新为 slide 加上 correction_step 的值。如果差值小于 correction_step 的负值，则将其转换为正值后按照之前的方式进行相同的更新。之后将包里的时间戳修改为加上 correction_slide 后的值。（如此修改后后续还有用否？数据包已经交给上层了。）

从上面描述的机制来看，jitter buffer 机制利用缓冲在一定程度上能保证数据包以比较均匀的速度上传给应用。

七：事件的处理

在 rtp 会话上，保存了 signal table、表中的各个事件的回调函数以及事件队列。如果接收到 signal table 中所注册的事件信息，则调用注册的回调函数进行处理，而对于 telephone event 包，除了调用回调函数外，还需要将其发送到事件队列，以便上层应用进行进一步的处理。

八：其他需要说明的

Rtp 这块有关时间戳的比较计算，主要通过几个宏来完成。RTP_TIMESTAMP_IS_NEWER_THAN(ts1,ts2) 比较 ts1 小于等于 ts2，RTP_TIMESTAMP_IS_STRICTLY_NEWER_THAN(ts1,ts2)则比较 ts1 小于 ts2，没有等于关系。但是实际实现中，差值都是与 2 的 31 次方来比较，原理如下：

就是将差值结果强制转换为 uinit 后于 0x8000000 进行比较，这样大于等于零返回成功，小于零返回失败。还是简单的比较关系而已。

关于 rtcp 同步 rtp 流的问题：

从 rtp 传输过来的媒体流可能为音频流和视频流，二者采用了不同的编码方式和不同的分采样率，同时这两个流中都带有时间戳信息。如何将这两个流进行同步，这可以通过 rtcp 的报告包来完成。在 rtcp 的报告包中带有具有绝对的基于 NTP 的时间戳信息，同时也带有与 rtp 流中相同的采样时间戳信息，依据这两个时间戳信息就可以对同一个媒体流进行同步。Rtcp 中带有的唯一的 cname 信息可以将同一个媒体的音频和视频流信息关联起来，虽然这两个流使用不同的源描述符 ssrc。

关于 rtcp 发送的间隔：

这块程序中只是按照 5 秒的间隔来进行计算，是固定的。存在巨大的局限性，但是对于点对点的通信来讲，问题不是很大。另外测试 eyebeam 程序，发现是按 3 秒左右来发送 rtcp 包的，不知是计算得到还是固定值。

这块可以按照协议要求对 oRTP 进行改进。

关于 rtcp 包的接收发送规则：

目前从代码来看，每次发送一个 rtp 包后检查是否需要发送 rtcp 包。每次接收 rtp 包时同时检查是否可以接收 rtcp 包，完了后检查是否需要发送 rtcp 包。但是协议的规则不是这样。

关于 rtcp 包的构建问题：

Rtcp 包必须以复合包的形式向网络上传输，而且必须至少包含两个基本的包，第一个必须是发送者报告包或者接受者报告包，另外包括 sdes 源描述项包。

关于 rtcp 协议方面的问题，可以参考 rtcp 协议的说明文档，参考资料 1

九：使用 oRTP 库

oRTP 提供了测试程序来测试 oRTP 库，同时测试程序也是如何使用 oRTP 最好的例子。测试程序在源代码的 test 目录下，包括发送测试 rtpsend.c，接收测试 rtprecv.c，并行发送测试 mrtpsend.c，并行接收测试 mrtprecv.c，还有有关 telephone event 相关的测试。这里的说明只针对上述四个有关接收发送测试程序的测试结果。

程序中提供的测试例子，大部分都是针对音频数据的，时间戳增加值也都是设置为 160 了（如何计算得来，参见时间戳部分的说明），这在接收和发送视频数据时存在问题，需要做些修改，具体见问题列表。

海思 3716 平台上 oRTP 代码的编译：.

/configure --host=arm-hisiv200-linux【指定交叉环境】 --prefix=【指定安装目录】

Make

Make install

完了之后会在指定的目录下创建 include、lib 以及 share 三个文件夹。include 包含了我们要使用 oRTP 库的头文件，lib 目录包含了编译完的库，share 下是文档说明。

可以把 test 目录下的文件拿到安装目录下，跟库一起编译出来可执行文件进行测试。

测试问题列表：

1 测试程序默认为测试音频流，没有包含视频流，时间戳步进值为 160，就是按照 20ms 采样周期加 8KHZ 采样率计算出来的。这就导致接收和发送视频流的速度都特别慢。因此接收视频数据时需要调整步进值为 3600，即 90000 采样率加 25 帧每秒计算得来。实际使用过程中可根据测试效果进行调整。

2 单独调整第一条还不能完全解决速率问题，还需要调整 payload type。接收过程可以根据接收的类型发生改变（即会产生 payload type changed 事件），从而对此作出相应调整，故对接收过程影响不大，但是发送过程因为默认 payload 类型设置为了 payload_type_pcmu8000，导致发送速度很慢，（基于 8KHZ 采样率）调整为 33（payload_type_mp2v）即可。

3 33 在 oRTP 中默认并没有添加支持，可仿照 avprofile.c 文件中的实现单独添加。

4 测试中发现 multiple recv 测试程序接收不到数据，将文件操作接口换为 fopen，及 ascii 标准类型的文件读写接口即可，原因待查。

5 多会话接收目前是按照端口号不同来进行的。

6 关闭调度模式和 block 模式可以加速视频流的推送

7 在没有调度器的情况下，控制视频流发送速度，可以改善马赛克情况。其实发送过快也不行，用 vlc 播放测试来看。

8 在启动调度器的情况下，设置应用时间戳增量值（user_ts）或者调整该变量也可以调整视频流的发送速度，调整马赛克情况。

9 数据发送速率与采样率，user_ts 时间戳值增加以及 buffer 大小均有关系。

10 在并行发送和接收测试时（msend、mrecv），需要设置为非阻塞模式，否则程序可能会被卡死。

十：参考

1 RFC3550 RTP: A Transport Protocol for Real-Time Applications

2 RFC3551 RTP Profile for Audio and Video Conferences with Minimal Control

3 RTCP 发送间隔分析