

11-Webrtc中RTCP使用及相关指标计算

RTCP报文类型

RTCP报文格式

200 SR: Sender Report RTCP Packet

201 RR: Receiver Report RTCP Packet

Fraction lost

Jitter

last SR和DelaySinceLastSR

音视频同步

SR和RR的区别

205 RTP-FEED BACK

NACK

Transport-CC

206 PAYLOAD-FEEDBACK

REMB

207 XR

RRT/DLRR

SS

Chrome 指标查看

SRS的处理

发布码流

处理rtcp包

处理on_rtcp_sr

处理on_rtcp_xr

回发rtcp包

SrsRtcRecvTrack::send_rtcp_rr()

SrsRtcRecvTrack::send_rtcp_xr_rrtr()

SrsRtcConnection::send_rtcp_fb_pli

SrsRtcPublishStream::send_periodic_twcc()

SrsRtcConnection::check_send_nacks

播放码流

处理rtcp包

SrsRtcPlayStream::on_rtcp_nack()

SrsRtcSendTrack::on_nack

SrsRtcPlayStream::on_rtcp_ps_feedback请求I帧

SrsRtcConnection::on_rtcp_feedback_remb没有处理

回发rtcp包

SrsRtcPlayStream::do_request_keyframe

名词解释补充

第一类：关键帧请求

第二类：重传请求

第三类：码率控制

参考

零声学院：音视频高级课程：<https://ke.qq.com/course/468797?tuin=137bb271>

本文版权归腾讯课堂 零声教育所有，侵权必究。

待续版本，此版本为过程版本，仅供进度比较快的小伙伴先学。

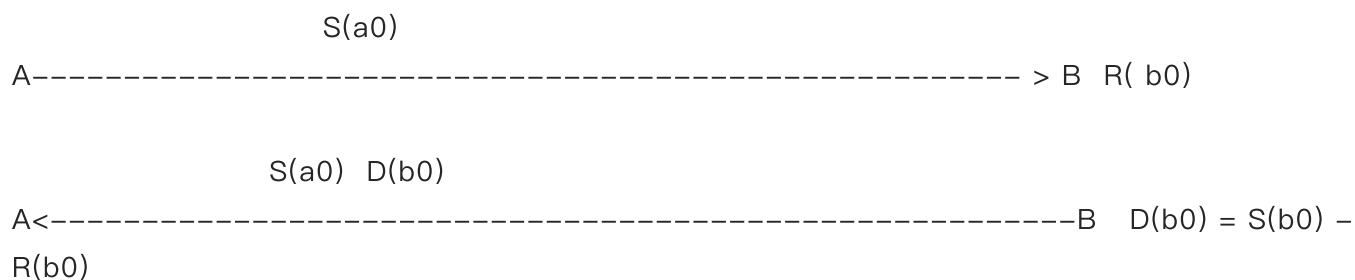
这一章涉及的概念非常多：

- SR的作用
 - NTP用于同步
- RR的作用
 - 计算RTT，用来确定丢包时何时触发重传
 - 计算丢包率
- SR和RR的区别
- RTT(Round-Trip Time): 往返时延怎么计算。在计算机网络中它是一个重要的性能指标，表示从发送端发送数据开始，到发送端收到来自接收端的确认（接收端收到数据后便立即发送确认），总共经历

的时延。

- 客户端计算
 - 作为发布者
 - 作为播放者
 - 服务器端计算
 - 作为接收者
 - 作为发送者
-
- 获取RTT有什么作用
 - 发送PLI包的目的
 - 什么时候发送NACK
 - 对端检测到丢包，请求重传，向发送端请求重传
 - 收到nack请求后，怎么处理？

大写S代表发送时间，R代表接收时间，D代表延时



1. A先发一个数据包，在本地记录下发送时间 $S(a_0)$,并把 $S(a_0)$ 放在数据包中发给B
2. B收到数据包，在本地记录下数据接收时间 $R(b_0)$,并把数据包发回A，发送时把本地处理延时时间 $D(b_0)$ 也记录到数据包中， $D(b_0) = S(b_0) - R(b_0)$
3. A收到B的回应后，记录到数据接收时间 $R(a_0)$
4. $rtt = R(a_0) - S(a_0) - D_0$

在RFC3550中，除了定义了用来进行实时数据传输的 RTP 协议外，还定义了 RTCP 协议，用来反馈会话传输质量、用户源识别、控制 RTCP 传输间隔。在 WebRTC 中，通过 RTCP 我

们可以实现发送数据/接收数据的反馈，传输控制如丢包重传、关键帧请求，网络指标 RTT、丢包率、抖动的计算及反馈，拥塞控制相关的带宽 反馈，以及用户体验相关的音视频同步等等。为了让开发者获取以上数据指标，Webrtc 提供了统一的接口调用,如在GoogleChrome 中,可以通过 `RTCPeerConnection.getStats()`或者<chrome://webrtc-internals/> 查看以上指标。

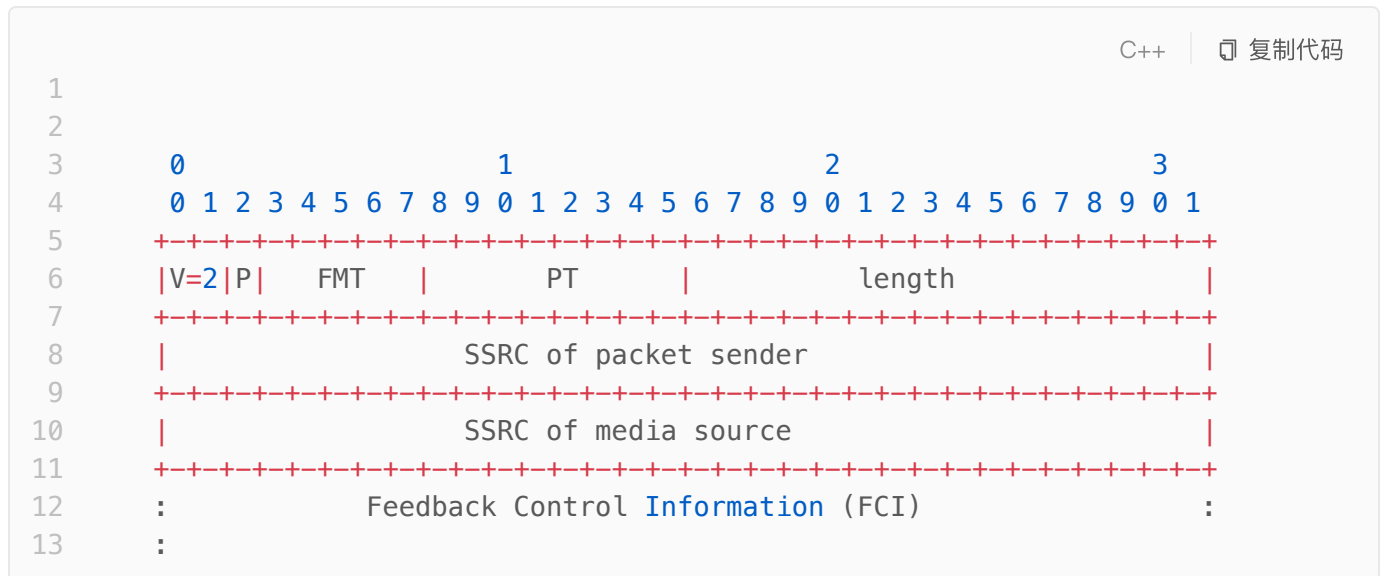
```
srs_kernel_rtc_rtcp.hpp
```

RTCP报文类型

目前 RTCP 主要定义了以下 8 种类型的报文，其中业务场景中主要用到 SR/RR/RTPFB/PSFB，接下来我们也将重点介绍这四种报文。未来如果有新类型的话，会继续从208-223中分配， 0/255目前禁止使用（对应PT值）。

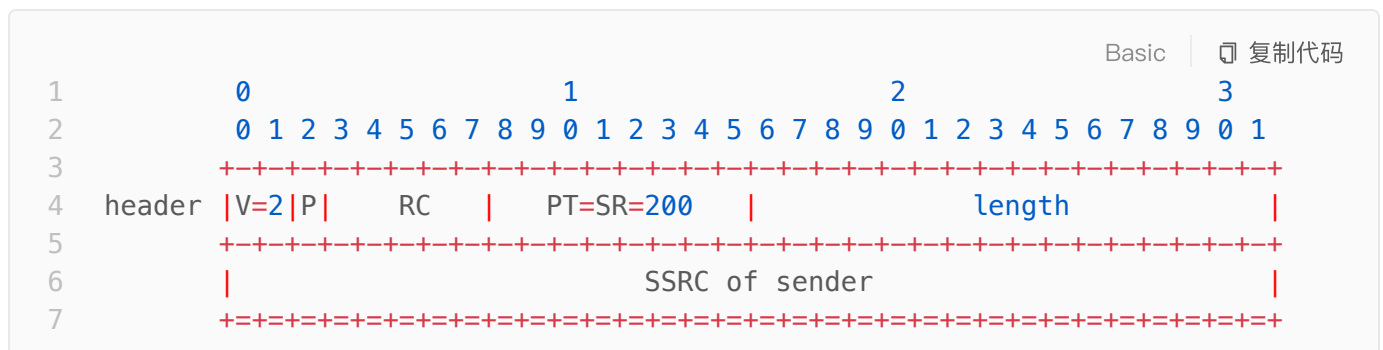
类型	缩写表示	用途
200	SR	发送者报告 rfc3550
201	RR	接收者报告 rfc3550
202	SDES	源点描述 rfc3550
203	BYE	结束 rfc3550
204	APP	实验性的功能和开发 rfc3550
205	RTPFB	传输层反馈RTP Feedback rfc4585/5104
206	PSFB	特定负荷类型反馈PayloadSpecificFeedback rfc4585/rfc5104
207	XR	XR, RTCP扩展 rfc3611

All FB messages **MUST** use a common packet format that is depicted in Figure 3:



RTCP报文格式

在介绍报文类型之前，我们先看一下报文格式：



- version (V): 2bits, 目前统一为2
- padding (P): 2bit, 在复合包中, 只有最后一个RTCP包需要添加填充
- reception report count (RC): 5bits, 多少个接收报告, 0是有效的
- packet type (PT): 8bits
- length : 16bits, 报文长度(32-bit 进制), 包含头部和填充字节
- SSRC: 32bits

每个 RTCP 包都有一个和 RTP 类似的固定格式的头，长度为8，后面跟着长度不定的结构化数据，在不同 RTCP 类型时，这些结构化数据各不一样，但是它们必须都要 **32-bit 对齐**。RTCP 的头部是定长的，而且在头部有一个字段来描述这个 RTCP 数据的长度，因此 RTCP 可以被复合成一组一同发送。

200 SR: Sender Report RTCP Packet

	0										1										2										3									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
1																																								
2																																								
3	+++++																																							
4	header	V=2 P		RC								PT=SR=200								length																				
5	+++++																																							
6			SSRC of sender																																					
7	+++++																																							
8	sender	NTP timestamp, most significant word																																						
9	info	+++++																																						
10			NTP timestamp, least significant word																																					
11	+++++																																							
12			RTP timestamp																																					
13	+++++																																							
14			sender's packet count																																					
15	+++++																																							
16			sender's octet count																																					
17	+++++																																							
18	report	SSRC_1 (SSRC of first source)																																						
19	block	+++++																																						
20	1	fraction lost										cumulative number of packets lost																												
21	+++++																																							
22			extended highest sequence number received																																					
23	+++++																																							
24			interarrival jitter																																					
25	+++++																																							
26			last SR (LSR)																																					
27	+++++																																							
28			delay since last SR (DLSR)																																					
29	+++++																																							
30	report	SSRC_2 (SSRC of second source)																																						
31	block	+++++																																						
32	2	:	...														:																							
33	+++++																																							
34			profile-specific extensions																																					
35	+++++																																							

- SR 也叫发送者报告，发送端会周期性的发送 SR，携带的是会话开始到现在为止累计的发包数/字节数。
- SR 中携带的 NTP/RTP Timestamp 主要用来在接收端进行多个媒体流间的同步。
- SR 和后面的 RR 配合使用可以进行 RTT 的计算。
- **ssrc**: 与 sdp 中的 ssrc 一致
- **ntpSec**: most significant word, 高位时间戳, **ntpFrac** 每到最大值就清零, **ntpSec** 加 1,

- **ntpFrac**: least significant word, 低位时间戳, 单位是微秒。网络时间戳, 用于同步不同源
- **rtpTs**: 相对时间戳, 与rtp一致
- **packetCount**: 一共发送了多少个包
- **octetCount**: 一共发送了多少字节
- **report.ssrc**: 信息所属的ssrc
- **report.fractionLost**: 丢包率
- **report.totalLost**: 一共丢包的个数。迟到包不算丢包, 重传有可能导致负数
- **report.lastSeq**: rtp头的**sequenceNumber**是16bit, 会循环从0开始。此处低位16bit与rtp一致, 高位表示循环次数
- **report.jitter**: RTP包到达时间间隔的统计方差
- **report.lsr**: 上一次收到SR的时间戳, $lsr = (ntpSec \ll 16) + (ntpFrac \gg 16)$
- **report.dlsr**: 上次从收到SR包到发送本报告的时间差

201 RR: Receiver Report RTCP Packet

	0										1										2										3									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
header	V=2 P										RC										PT=RR=201										length									
											SSRC of packet sender																													
report block											SSRC_1 (SSRC of first source)																													
1	fraction lost										cumulative number of packets lost																													
											extended highest sequence number received																													
											interarrival jitter																													
											last SR (LSR)																													
											delay since last SR (DLSR)																													
report block											SSRC_2 (SSRC of second source)																													
2	:											...										:																		
											profile-specific extensions																													

- RC: 0 代表report block为空
- SSRC_n (source identifier): 32bits, 源SSRC
- fraction lost: 8bits, 丢包率
- cumulative number of packets lost: 24bits, 累计丢包
- extended highest sequence number received: 32bits, 收到的最大序列号, 高16位代表循环次数, 低16位代表最新收到的RTP序列号
- interarrival jitter: 数据包发送->到达的抖动时间
- last SR (LSR): 32bits, NTP timestamp(64位)的中间32位
- delay since last SR (DLSR): 32bits, 单位为1/65536秒, 收到SR到现在经过的时间。

与 SR 相对应，RR 也叫接收者报告，RR 中定义了更多的指标信息，即反应了收包状态，又反应了网络状态，因此我们有必要了解这些指标都是怎么计算的，来保证反馈的准确性。

Fraction lost

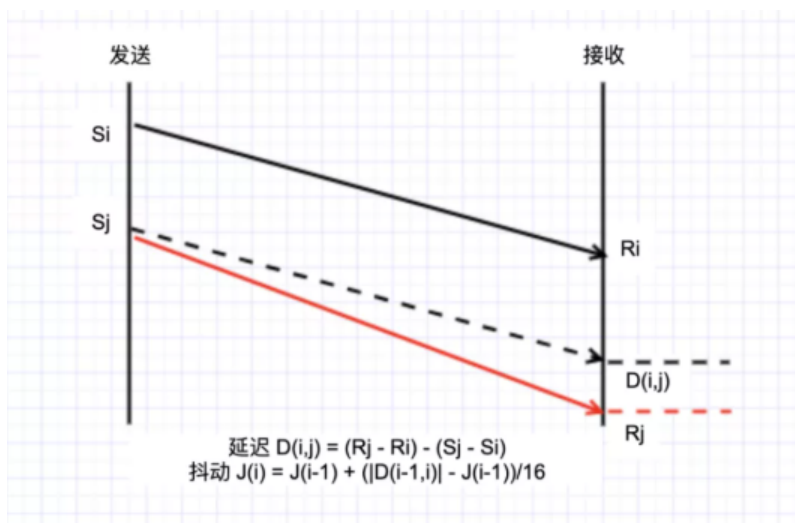
每个统计周期以 SR/RR 为间隔统计， $\text{fraction lost} = (\text{期望收包数} - \text{实际收包数}) / \text{期望收包数} * 255$ 在 RTP 传输中，收包、丢包计算都依赖于 RtpHeader 中的 Sequence number，由于只被分配了16字节，即最大序号为65535，很容易用尽，所以需要我们做循环计数，并在计算中转化为uint32这种取值范围更大的类型来表示。

```
1 // 计算最大序列号
2 uint32_t max_seq = cycles << 16 | seq;
3
4 // 期望累计收包 = 最大序列号 - 起始序列号
5 total_expected_received = max_seqnum - base_seqnum;
6
7 // 当前周期期望收包数 = 期望累计收包 - 上一周期周期累计收包
8 expected_received_interval = total_expected_received -
9   last_total_expected_received;
10 last_total_expected_received = total_expected_received;
11
12 // 当前周期收包数 = 累计收包 - 上一周期累计收包
13 received_interval = total_received - last_total_received;
14 last_total_received = total_received;
15
16 // 丢包数 = 当前周期期望收包数 - 当前周期收包数
17 lost_interval = expected_received_interval - received_interval;
18
19 // 丢包率
20 fraction_lost = lost_interval / expected_received_interval * 255;
```

Jitter

抖动的定义是信号在某特定时刻相对于其理想时间位置上的短期偏离。在网络传输中，数据包可能会经过不同的路由链路，当时的网络或拥塞或空闲，最终到达目的地时，与预期会有所偏差。通过数据包的到达情况，我们可以反过来估测网络的状态变化，用来对发送端进行指导。RFC3550中定义了相关计算公式。

```
延迟  $D(i,j) = (R_j - R_i) - (S_j - S_i) = (R_j - S_j) - (R_i - S_i)$ 
抖动  $J(i) = J(i-1) + (|D(i-1,i)| - J(i-1))/16$ 
```



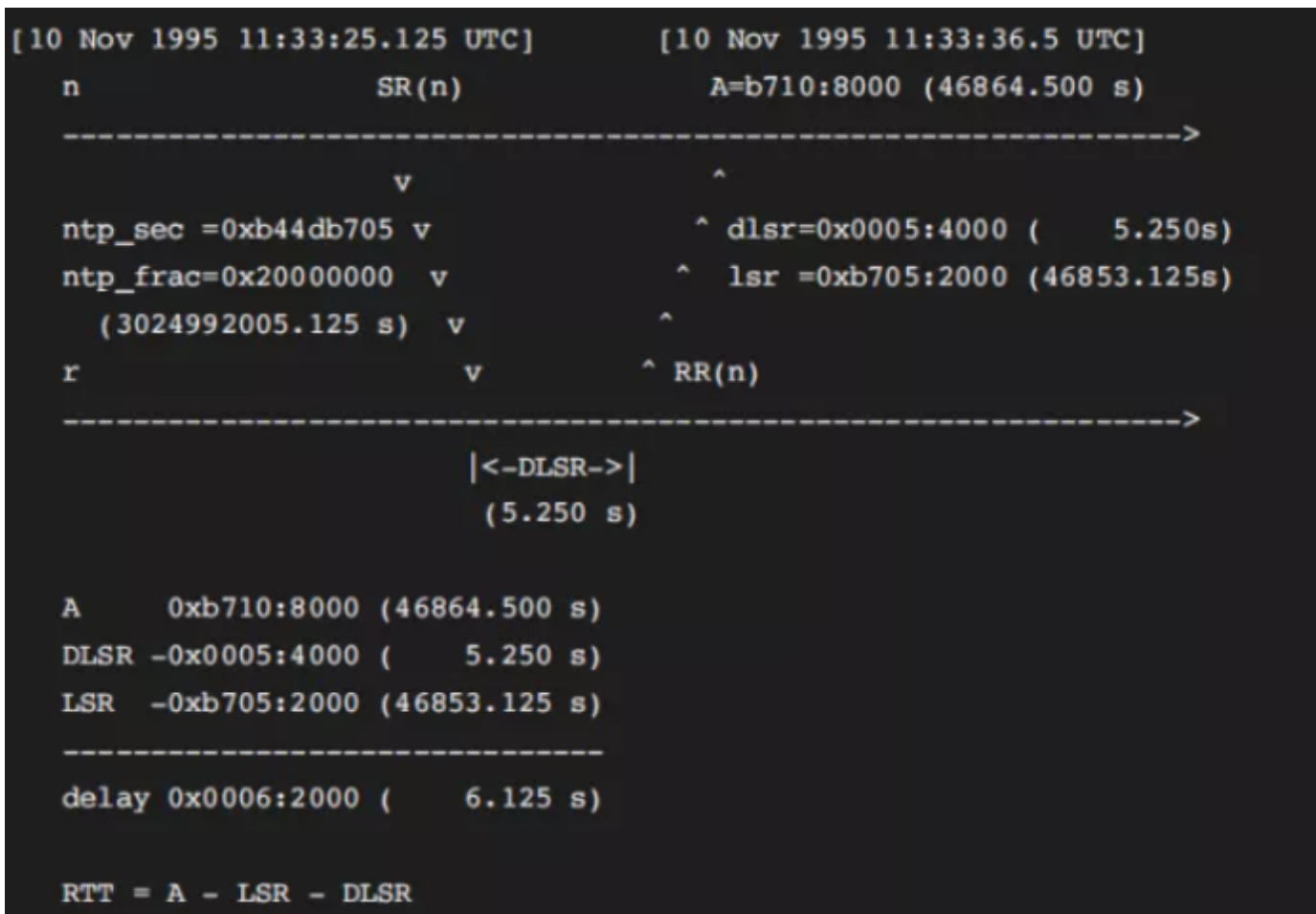
last SR和DelaySinceLastSR

除了丢包、抖动以外，网络中我们最常关注的一个指标就是 RTT，常见的操作是通过ping命令查看网络中的往返延迟。RTCP 中为了计算RTT，在 RR 中会携带上次收到的 SR 中的 NTPTime (last SR)，并计算其收到时**在本机经历的时间**，用 DelaySinceLastSR 表示。发送端收到后，剔除掉 DelaySinceLastSR 便是 RTT。

eg: 假设本次构造 RR 距离上次收到 SR 为 5250毫秒。


$dlsr = 5250/1000 \ll 16 \mid 5250\%1000*65.535 \Rightarrow 0x0005:4000$

RTT计算



音视频同步

在 RTP 传输中，携带的 timestmap 的初始值是随机产生的，另外音频、视频可以从不同的传输通道独立传输，虽然都是线性单调递增的，但是两者 RTP 的 timestmap 是没有相互关系的，需要 RTCP 提供额外信息来进行同步。为了实现音视频同步，发送端会定期发送 Sender Report，携带 rtp timestmap、ntptime，接收端把每路流收到的 rtp timestamp 都转换为 NTP 时间，实现同步。在实现同步之前需要知道，我们常说的采样率对应的时间单位是1秒。音频的timestamp 一般根据每帧的采样点数增加，如opus 48khz 代表每秒采样48000，如果每 20ms采样一次，则每次采样为 $48000 / (1000 / 20) = 960$ 。由于音频采集频率高，不容易出现误差，所以程序实现中增长都是固定的960。

Info			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14420, Time=3650976406			960 线性递增 
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14421, Time=3650977366			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14422, Time=3650978326			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14423, Time=3650979286			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14424, Time=3650980246			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14425, Time=3650981206			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14426, Time=3650982166			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14427, Time=3650983126			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14428, Time=3650984086			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14429, Time=3650985046			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14430, Time=3650986006			
PT=DynamicRTP-Type-111, SSRC=0xF9670969, Seq=14431, Time=3650986966			

视频的采样率一般为90khz， 由于视频采样频率低，容易出现误差，实际计算时间戳时，会根据系统时间来计算，每帧之间增长不一定是固定的值，如图帧率为15fps， 每次增长不是固定值，约等于 90000/15。

PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25504, Time=2777787940, Mark	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25505, Time=2777793880	5940
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25506, Time=2777793880	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25507, Time=2777793880, Mark	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25508, Time=2777799910	6030
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25509, Time=2777799910	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25510, Time=2777799910, Mark	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25511, Time=2777805940	6030
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25512, Time=2777805940	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25513, Time=2777805940, Mark	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25514, Time=2777811880	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25515, Time=2777811880	5940
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25516, Time=2777811880	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25517, Time=2777811880, Mark	
PT=DynamicRTP-Type-102, SSRC=0x86D42D42, Seq=25518, Time=2777817910	

C 复制代码

```

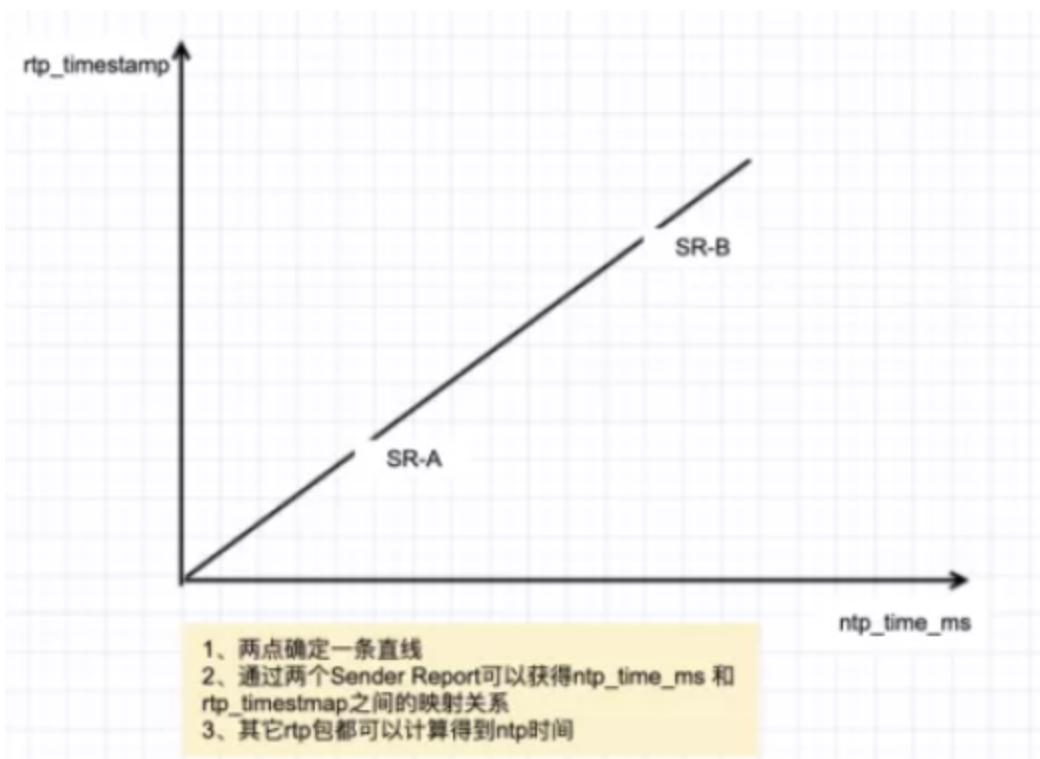
1 //通过两个sender report计算采样率，新版本webrtc中实现会更复杂一些
2 frequency_khz = static_cast<double>(rtp_timestamp1-
  rtp_timestamp2)/static_cast<double>(ntp_ms1-ntp_ms2)
3
4 // 估算rtp_timestamp对应的ntp_ms
5 *ntp_ms = rtcp_ntp_ms - (static_cast<double>(rtcp_timestamp)-
  rtp_timestamp)/frequency_khz + 0.5f;
6 1. 多个sender report计算可以得到采样率
7 2.rtp_timestamp和rtcp_timestamp 的delta基于采样率换算为毫秒
8 3.基于rtcp_ntp_ms计算得到当前rtp包的ntp_time_ms

```

```
// 通过两个Sender Report计算采样率, 新版webrtc 中实现会更复杂一些
frequency_khz = static_cast<double>(rtp_timestamp1 - rtp_timestamp2) / static_cast<double>(ntp_ms1 - ntp_ms2);

// 估算rtp_timestamp 对应的ntp_ms
*ntp_ms = rtcp_ntp_ms - (static_cast<double>(rtcp_timestamp) - rtp_timestamp) / frequency_khz_ + 0.5f;
```

- 1、多个SenderReport计算可以得到采样率
- 2、rtp_timestamp 和 rtcp_timestamp 的delta基于采样率换算为毫秒
- 3、基于rtcp_ntp_ms计算得到当前rtp包的ntp_time_ms



SR和RR的区别

从报文内容上看SR比RR多出来了sender info这部分数据段，这部分包含了发送报告方作为发送者相关的信息。

SR表面译为作为发送者报告，RR表面译为接收者报告

实际上按字面上理解容易造成误解，以为SR就是发送端的报告，而RR就是接收端的报告。其实不是这样的。

SR并不是发送者发送多少包告诉对方，而是sender这端接收了多少包，接收过程中丢了多少包。将这些信息传送给对端。

所以SR包含两个信息，将sender自己发送的信息告诉对端，同时将自己接收的信息也告诉对端。而RR是当报文发送方只作为接收者，而不发送媒体数据时，发给对端自己作为接收方接收到的数据的统计信息。

RTP 报文的接收者可以利用两种类型的 RTCP 报告报文(SR 或 RR)来提供有关数据接收质量的统计信息，具体选用 SR 报文还是 RR 报文要看该接收者是否同时是一个 RTP 报文的发送者：

- 如果一个会话参加者自最后一次 发送 RTCP报文后，发送了新的 RTP 数据报文，那么该参加者需要传送 SR 报文,, 否则传送 RR 报文。
- SR 报文和RR 报文的主要区别在于前者包含了20字节有关发送者的信息。

205 RTP-FEED BACK

RTP-FEEDBACK 主要用来在传输层进行反馈，实现数据包的丢包重传，码率控制，主要有以下几种类型：

FMT	用途
1	NACK请求发送方重传报文
3	TMMBR临时最大码率请求
4	TMMBN临时最大码率通知
5	SR_REQ
6	RAMS
7	TLLEI
8	ECN
9	PS
15	Transport Wide CC

NACK

在 RTP-FEEDBACK 中，最重要的当属NACK，区别于 TCP 中的 ACK，在 RTCP 中 NACK 代表否定应答，当接收方监测到数据包丢失时，发送一个 NACK 到发送方，表明自己

没有收到某个报文。目前 NACK 可以认为是对抗弱网最主要的手段，报文格式如下：

- 4个字节(2字节 PID 代表第一个开始丢的包、2字节BLP代表紧随其后的16个包的丢包信息、bit1代表丢包)；
- 可以同时携带多个nack block.

```
// Generic NACK RTCP_RTP_FB + (FMT 1)rfc4585
//      0              1              2              3
//      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
//      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
//      |              PID              |              BLP              |
//      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Transport-CC

- Transport-cc 是目前 Webrtc 中最新的拥塞控制算法，替代旧的 GCC 算法；
- Transport-CC 需要在 RTP 中增加扩展，接收端记录 RTP 包的到达时间、间隔并反馈给发送端，这里不做详细介绍。


```

1  /*
2  @doc: https://tools.ietf.org/html/draft-holmer-rmcat-transport-wide-cc-
    extensions-01#section-3.1
3
4      0      1      2      3
5      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
6      |V=2|P|  FMT=15 |    PT=205    |          length          |
7      +-----+-----+-----+-----+-----+-----+-----+
8      |                                     SSRC of packet sender |
9      +-----+-----+-----+-----+-----+-----+-----+
10     |                                     SSRC of media source  |
11     +-----+-----+-----+-----+-----+-----+-----+
12     |   base sequence number   |   packet status count   |
13     +-----+-----+-----+-----+-----+-----+-----+
14     |               reference time               | fb pkt. count |
15     +-----+-----+-----+-----+-----+-----+-----+
16     |   packet chunk   |   packet chunk   |
17     +-----+-----+-----+-----+-----+-----+-----+
18     .
19     .
20     +-----+-----+-----+-----+-----+-----+-----+
21     |   packet chunk   |   recv delta   |   recv delta   |
22     +-----+-----+-----+-----+-----+-----+-----+
23     .
24     .
25     +-----+-----+-----+-----+-----+-----+-----+
26     |   recv delta   |   recv delta   |   zero padding   |
27     +-----+-----+-----+-----+-----+-----+-----+
28  */

```

206 PAYLOAD-FEEDBACK

RTC 中，主要传输的是音频、视频，由于两种媒体有不同的特点，音频小包，前后帧无参考；视频帧具有前后参考关系，关键帧又是 GOP 中后续帧的主要参考对象，可以说是重中之重，需要 RTCP 针对具体的载荷类型进行更精细化的信息反馈。目前主要有以下类型：

FMT	用途
0	unassigned
1	Picture Loss Indication (PLI)
2	Slice Loss Indication (SLI)
3	Reference Picture Selection Indication (RPSI)
4	Full Intra Request
5	Temporal-Spatial Trade-off Request
6	Temporal-Spatial Trade-off Notification (TSTN)
7	Video Back Channel Message (VBCM)
8-14	unassigned
15	ApplicationLayerFeeedbackMessage
16-30	unassigned
31	reserved for future expansion of the sequence number space

- 当前的实现中，主要的反馈实现是PLI，当网络出现丢包时，接收方反馈帧丢失，请求发送方重新编码关键帧发送；
- FIR 也是请求关键帧，主要用在新用户加入的场景中；
- ApplicationLayerFeeedbackMessage 中主要是 REMB 经常使用。

REMB

REMB 即接收端最大接收码率估测，需要配合rtp扩展abs-send-time一起使用，是接收端估算的本地最大带宽能力，发送端根据 REMB、丢包等进行拥塞控制。

REMB 估算的码率代表的是一个传输通道内所有 SSRC 的码率之和，而不是针对于某一个特定的 SSRC。

FMT	用途
1	Loss RLE Report (LRLE)
2	Duplicate RLE (DRLE)
3	Packet Receipt Times (PRT)
4	Receiver Reference Time (RRT)
5	Delay since the last Receiver (DLRR)
6	Statistics Summary (SS)
7	Voip Metrics (VM)

C++ | 复制代码

```

1  @see: http://www.rfc-editor.org/rfc/rfc3611.html#section-2
2
3      0              1              2              3
4      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
5      +-----+-----+-----+-----+-----+-----+-----+-----+
6      |V=2|P|reserved|   PT=XR=207   |               length               |
7      +-----+-----+-----+-----+-----+-----+-----+-----+
8      |                                     SSRC                                     |
9      +-----+-----+-----+-----+-----+-----+-----+-----+
10     :                                     report blocks                                     :
11     +-----+-----+-----+-----+-----+-----+-----+-----+

```

RRT/DLRR

上面讲到的 RTT 是发送方的计算逻辑，对于纯接收方，同样有必要知道网络 RTT，就需要用到该扩展。接收方发送 RRT 扩展后，发送方在下次的 RTCP 中携带 DLRR 扩展，计算方式和 SR 计算 RTT 类似。

@see: <http://www.rfc-editor.org/rfc/rfc3611.html#section-4.4>



SS

```

1      0              1              2              3
2      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
3      +-----+-----+-----+-----+-----+-----+-----+-----+
4      |      BT=6      |L|D|J|ToH|rsvd.|      block length = 9      |
5      +-----+-----+-----+-----+-----+-----+-----+-----+
6      |                                     SSRC of source          |
7      +-----+-----+-----+-----+-----+-----+-----+-----+
8      |      begin_seq          |      end_seq          |
9      +-----+-----+-----+-----+-----+-----+-----+-----+
10     |                                     lost_packets          |
11     +-----+-----+-----+-----+-----+-----+-----+-----+
12     |                                     dup_packets          |
13     +-----+-----+-----+-----+-----+-----+-----+-----+
14     |                                     min_jitter          |
15     +-----+-----+-----+-----+-----+-----+-----+-----+
16     |                                     max_jitter          |
17     +-----+-----+-----+-----+-----+-----+-----+-----+
18     |                                     mean_jitter          |
19     +-----+-----+-----+-----+-----+-----+-----+-----+
20     |                                     dev_jitter          |
21     +-----+-----+-----+-----+-----+-----+-----+-----+
22     | min_ttl_or_hl | max_ttl_or_hl | mean_ttl_or_hl | dev_ttl_or_hl |
23     +-----+-----+-----+-----+-----+-----+-----+-----+

```

可以看到，报文中携带了更加详细的指标信息，尤其是 Jitter 指标，对于带宽评估有很好的参考作用。

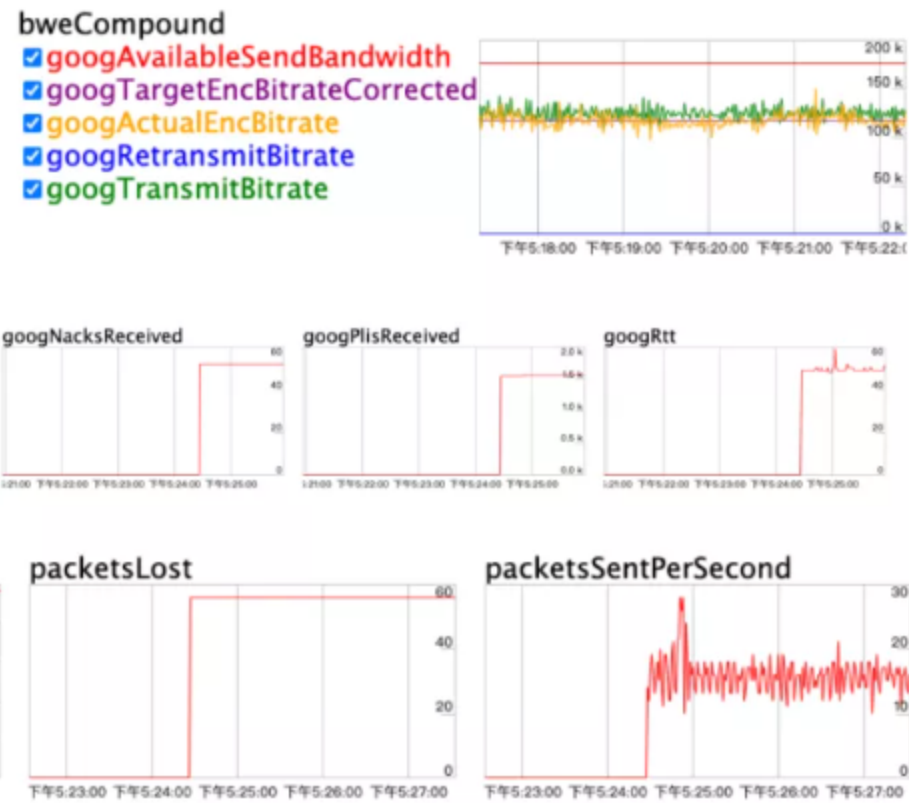
Chrome 指标查看

了解了指标的计算，如何确认指标计算是否正常，反馈是否准确，对于开发者同样重要。以 Chrome 浏览器为例，目前开发者想要查看底层统计指标有两种方式：

- 通过 `RTCPeerConnection.getStats()` 接口调用；
- 通过 `chrome://webrtc-internals/` 查看，目前 Chrome 处于两种 `getStats` 的过渡期，可以切换查看不同指标。

我们下面列举几个通过chrome://webrtc-internals/查看的指标：

▼ Stats graphs for bweforvideo (VideoBwe)



mslabel:s9knsCqchLnk7Qbzk0VDtJAQ1NgPXhS305cr
label:9f0429e0-d1a7-4498-bc1f-de9d7c22d562

timestamp	2021/4/9下午5:28:39
bytesSent	721635684
codecImplementationName	ExternalEncoder
framesEncoded	99532
mediaType	video
packetsLost	56
packetsSent	652926
ssrc	1047913980
transportId	Channel-0-1
googAdaptationChanges	0
googAvgEncodeMs	840
googBandwidthLimitedResolution	false
googCodecName	H264
googContentType	realtime
googCpuLimitedResolution	false
googEncodeUsagePercent	101

以上我们对 RTCP 进行了简单的讲解，实际的开发场景中，对于 RTCP 的使用还有很多需要注意的点，不同类型的报文采用不同的发送策略、发送周期、触发时机等，欢迎大家一起讨论。

SRS的处理

发布码流

处理rtcp包

```
1  srs_error_t SrsRtcPublishStream::on_rtcp(SrsRtcpCommon* rtcp)
2  {
3      if(SrsRtcpType_sr == rtcp->type()) {
4          SrsRtcpSR* sr = dynamic_cast<SrsRtcpSR*>(rtcp);
5          return on_rtcp_sr(sr);
6      } else if(SrsRtcpType_xr == rtcp->type()) {
7          SrsRtcpXr* xr = dynamic_cast<SrsRtcpXr*>(rtcp);
8          return on_rtcp_xr(xr);
9      } else if(SrsRtcpType_sdes == rtcp->type()) {
10         //ignore RTCP SDES
11         return srs_success;
12     } else if(SrsRtcpType_bye == rtcp->type()) {
13         // TODO: FIXME: process rtcp bye.
14         return srs_success;
15     } else {
16         return srs_error_new(ERROR_RTC_RTCP_CHECK, "unknown rtcp type=%u",
17             rtcp->type());
18     }
19 }
```

从这里可以得知，srs主要处理了sr、xr两种类型的包。

处理on_rtcp_sr

我们主要关注ntp值，在回发rr包的时候需要使用。


```

1 void SrsRtcPublishStream::update_send_report_time(uint32_t ssrc, const
  SrsNtp& ntp)
2 { // 每个tracker有自己的ntp值
3     SrsRtcVideoRecvTrack* video_track = get_video_track(ssrc);
4     if (video_track) {
5         return video_track->update_send_report_time(ntp);
6     }
7
8     SrsRtcAudioRecvTrack* audio_track = get_audio_track(ssrc);
9     if (audio_track) {
10        return audio_track->update_send_report_time(ntp);
11    }
12 }

```

堆栈

```

#0 SrsRtcPublishStream::update_send_report_time (this=0x117e810, ssrc=2554582209, ntp=...)
   at src/app/srs_app_rtc_conn.cpp:1621
#1 0x0000000000608cdc in SrsRtcPublishStream::on_rtcp_sr (this=0x117e810,
rtcp=0x10e6b20) at src/app/srs_app_rtc_conn.cpp:1479
#2 0x0000000000608b7c in SrsRtcPublishStream::on_rtcp (this=0x117e810, rtcp=0x10e6b20)
   at src/app/srs_app_rtc_conn.cpp:1456
#3 0x000000000060c426 in SrsRtcConnection::dispatch_rtcp (this=0x10906b0,
rtcp=0x10e6b20) at src/app/srs_app_rtc_conn.cpp:2112
#4 0x000000000060bb7e in SrsRtcConnection::on_rtcp (this=0x10906b0, data=0x1064260
"\200", <incomplete sequence \310>,
   nb_data=106) at src/app/srs_app_rtc_conn.cpp:2026
#5 0x0000000000643dba in SrsRtcServer::on_udp_packet (this=0xf4f500,
skt=0x7ffff7fe4b80) at src/app/srs_app_rtc_server.cpp:430
#6 0x00000000005de8a4 in SrsUdpMuxListener::cycle (this=0x10410b0) at
src/app/srs_app_listener.cpp:636
#7 0x0000000000536712 in SrsFastCoroutine::cycle (this=0x10511f0) at
src/app/srs_app_st.cpp:270
#8 0x00000000005367a8 in SrsFastCoroutine::pfm (arg=0x10511f0) at
src/app/srs_app_st.cpp:285
#9 0x00000000006a6bc9 in _st_thread_main () at sched.c:363

```

处理on_rtcp_xr

主要是用来计算rtt，计算RTT有什么用呢？主要用来控制nack的时间间隔。

```
#0 SrsRtcPublishStream::on_rtcp_xr (this=0x1087070, rtcp=0x12d7850) at
src/app/srs_app_rtc_conn.cpp:1485
#1 0x0000000000608bdf in SrsRtcPublishStream::on_rtcp (this=0x1087070, rtcp=0x12d7850)
at src/app/srs_app_rtc_conn.cpp:1459
#2 0x000000000060c426 in SrsRtcConnection::dispatch_rtcp (this=0x10906b0,
rtcp=0x12d7850) at src/app/srs_app_rtc_conn.cpp:2112
#3 0x000000000060bb7e in SrsRtcConnection::on_rtcp (this=0x10906b0, data=0x1064260
"\200", <incomplete sequence \310>,
nb_data=106) at src/app/srs_app_rtc_conn.cpp:2026
#4 0x0000000000643dba in SrsRtcServer::on_udp_packet (this=0xf4f500,
skt=0x7ffff7fe4b80) at src/app/srs_app_rtc_server.cpp:430
#5 0x00000000005de8a4 in SrsUdpMuxListener::cycle (this=0x10410b0) at
src/app/srs_app_listener.cpp:636
#6 0x0000000000536712 in SrsFastCoroutine::cycle (this=0x10511f0) at
src/app/srs_app_st.cpp:270
#7 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0x10511f0) at
src/app/srs_app_st.cpp:285
#8 0x00000000006a6bc9 in _st_thread_main () at sched.c:363
```

回发rtcp包

SrsRtcRecvTrack::send_rtcp_rr() 回发rr包 定时器控制发包
SrsRtcRecvTrack::send_rtcp_xr_rrtr() 回发xr扩展包 定时器控制发包
以上两者是在同一个定时器发送，先发rr，再发送了xr包。

SrsRtcPublishStream::send_periodic_twcc()

SrsRtcConnection::send_rtcp_fb_pli() 回发fb pli包 定时器触发发包请求

SrsRtcRecvTrack::send_rtcp_rr()

```
#0 SrsRtcConnection::send_rtcp_rr (this=0x108b930, ssrc=1438760234,
rtp_queue=0x1093940,
last_send_systime=@0x1093660: 1634215659391917, last_send_ntp=...) at
src/app/srs_app_rtc_conn.cpp:2381
```

```

#1 0x0000000000651c58 in SrsRtcRecvTrack::send_rtcp_rr (this=0x1093610) at
src/app/srs_app_rtc_source.cpp:2252
#2 0x000000000060744a in SrsRtcPublishStream::send_rtcp_rr (this=0x1092ab0) at
src/app/srs_app_rtc_conn.cpp:1178
#3 0x000000000060552b in SrsRtcPublishRtcpTimer::on_timer (this=0x1092310,
interval=1000000)
    at src/app/srs_app_rtc_conn.cpp:892
#4 0x00000000005ec66f in SrsFastTimer::cycle (this=0xf4a7d0) at
src/app/srs_app_hourglass.cpp:204
#5 0x0000000000536712 in SrsFastCoroutine::cycle (this=0xf4a830) at
src/app/srs_app_st.cpp:270
#6 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0xf4a830) at
src/app/srs_app_st.cpp:285
#7 0x00000000006a6bc9 in _st_thread_main () at sched.c:363
#8 0x00000000006a7446 in st_thread_create (start=0x0, arg=0x7fffffff080, joinable=0,
stk_size=16034912) at sched.c:694
#9 0x00007fffffddee8 in ?? ()
#10 0x0000000000000008 in ?? ()

```

```

#0 SrsRtcPublishRtcpTimer::SrsRtcPublishRtcpTimer (this=0x1092a50, p=0x1089310) at
src/app/srs_app_rtc_conn.cpp:869
#1 0x0000000000605ca5 in SrsRtcPublishStream::SrsRtcPublishStream (this=0x1089310,
session=0x108e900, cid=...)
    at src/app/srs_app_rtc_conn.cpp:972
#2 0x000000000061570f in SrsRtcConnection::create_publisher (this=0x108e900,
req=0x10a2d40, stream_desc=0x109a4c0)
    at src/app/srs_app_rtc_conn.cpp:3487
#3 0x000000000060adb9 in SrsRtcConnection::add_publisher (this=0x108e900,
ruc=0x1126c00, local_sdp=...)
    at src/app/srs_app_rtc_conn.cpp:1880
#4 0x00000000006444d7 in SrsRtcServer::do_create_session (this=0xf4f500, ruc=0x1126c00,
local_sdp=..., session=0x108e900)
    at src/app/srs_app_rtc_server.cpp:501
#5 0x0000000000644362 in SrsRtcServer::create_session (this=0xf4f500, ruc=0x1126c00,
local_sdp=..., psession=0x1126108)
    at src/app/srs_app_rtc_server.cpp:483
#6 0x000000000065d5ac in SrsGoApiRtcPublish::do_serve_http (this=0x1051450,
w=0x11272d0, r=0x10923f0, res=0x108d390)
    at src/app/srs_app_rtc_api.cpp:413

```

#7 0x000000000065c3c0 in SrsGoApiRtcPublish::serve_http (this=0x1051450, w=0x11272d0, r=0x10923f0)
at src/app/srs_app_rtc_api.cpp:287

SrsRtcRecvTrack::send_rtcp_xr_rrtr()

#0 SrsRtcRecvTrack::send_rtcp_xr_rrtr (this=0x1093d90) at
src/app/srs_app_rtc_source.cpp:2261
#1 0x0000000000607591 in SrsRtcPublishStream::send_rtcp_xr_rrtr (this=0x1092b30) at
src/app/srs_app_rtc_conn.cpp:1192
#2 0x0000000000605618 in SrsRtcPublishRtcpTimer::on_timer (this=0x108cf50,
interval=1000000)
at src/app/srs_app_rtc_conn.cpp:897
#3 0x00000000005ec66f in SrsFastTimer::cycle (this=0xf4a7d0) at
src/app/srs_app_hourglass.cpp:204
#4 0x0000000000536712 in SrsFastCoroutine::cycle (this=0xf4a830) at
src/app/srs_app_st.cpp:270
#5 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0xf4a830) at
src/app/srs_app_st.cpp:285
#6 0x00000000006a6bc9 in _st_thread_main () at sched.c:363

#0 SrsRtcPublishRtcpTimer::SrsRtcPublishRtcpTimer (this=0x1092a50, p=0x1089310) at
src/app/srs_app_rtc_conn.cpp:869
#1 0x0000000000605ca5 in SrsRtcPublishStream::SrsRtcPublishStream (this=0x1089310,
session=0x108e900, cid=...)
at src/app/srs_app_rtc_conn.cpp:972
#2 0x000000000061570f in SrsRtcConnection::create_publisher (this=0x108e900,
req=0x10a2d40, stream_desc=0x109a4c0)
at src/app/srs_app_rtc_conn.cpp:3487
#3 0x000000000060adb9 in SrsRtcConnection::add_publisher (this=0x108e900,
ruc=0x1126c00, local_sdp=...)
at src/app/srs_app_rtc_conn.cpp:1880
#4 0x00000000006444d7 in SrsRtcServer::do_create_session (this=0xf4f500, ruc=0x1126c00,
local_sdp=..., session=0x108e900)

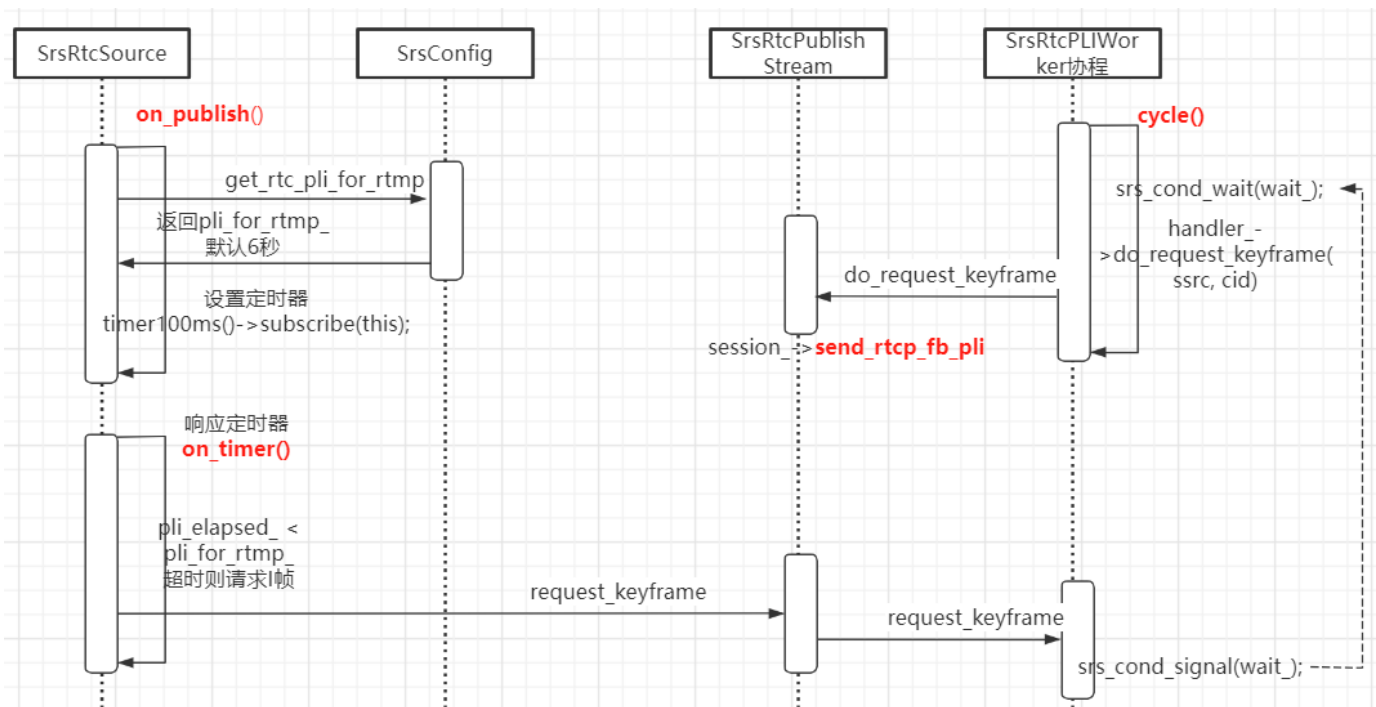
```

at src/app/srs_app_rtc_server.cpp:501
#5 0x0000000000644362 in SrsRtcServer::create_session (this=0xf4f500, ruc=0x1126c00,
local_sdp=..., psession=0x1126108)
at src/app/srs_app_rtc_server.cpp:483
#6 0x000000000065d5ac in SrsGoApiRtcPublish::do_serve_http (this=0x1051450,
w=0x11272d0, r=0x10923f0, res=0x108d390)
at src/app/srs_app_rtc_api.cpp:413
#7 0x000000000065c3c0 in SrsGoApiRtcPublish::serve_http (this=0x1051450, w=0x11272d0,
r=0x10923f0)
at src/app/srs_app_rtc_api.cpp:287

```

SrsRtcConnection::send_rtcp_fb_pli

这里的请求主要是针对rtc转rtmp，缺省6秒向推流者请求发送I帧。



SrsRtcPLIWorker::request_keyframe 触发（通过定时器触发该函数的调用）

```

#0 SrsRtcPLIWorker::request_keyframe (this=0x108dc10, ssrc=1686873537, cid=...) at
src/app/srs_app_rtc_conn.cpp:333
#1 0x000000000060914b in SrsRtcPublishStream::request_keyframe (this=0x1090ff0,
ssrc=1686873537)
at src/app/srs_app_rtc_conn.cpp:1547
#2 0x0000000000649eb4 in SrsRtcSource::on_timer (this=0x108aec0, interval=100000) at
src/app/srs_app_rtc_source.cpp:706

```

```

#3 0x00000000005ec66f in SrsFastTimer::cycle (this=0xf4a6f0) at
src/app/srs_app_hourglass.cpp:204
#4 0x0000000000536712 in SrsFastCoroutine::cycle (this=0xf4a750) at
src/app/srs_app_st.cpp:270
#5 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0xf4a750) at
src/app/srs_app_st.cpp:285
#6 0x00000000006a6bc9 in _st_thread_main () at sched.c:363

#0      SrsRtcConnection::send_rtcp_fb_pli      (this=0x1087070,      ssrc=2471027922,
cid_of_subscriber=...)
    at src/app/srs_app_rtc_conn.cpp:2482
#1 0x000000000060931e in SrsRtcPublishStream::do_request_keyframe (this=0x1090520,
ssrc=2471027922, sub_cid=...)
    at src/app/srs_app_rtc_conn.cpp:1560
#2 0x00000000006020b8 in SrsRtcPLIWorker::cycle (this=0x108df40) at
src/app/srs_app_rtc_conn.cpp:357
#3 0x0000000000536712 in SrsFastCoroutine::cycle (this=0x109a7f0) at
src/app/srs_app_st.cpp:270
#4 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0x109a7f0) at
src/app/srs_app_st.cpp:285
#5 0x00000000006a6bc9 in _st_thread_main () at sched.c:363
#6 0x00000000006a7446 in st_thread_create (start=0x9e7fd2, arg=0x7fff7fe2460,
joinable=11, stk_size=0) at sched.c:694
#7 0x00007ffff6bb3287 in _IO_vsscanf (string=0x1087c60 "\016", format=0x7fff6cd1255
"%hu%n:%hu%n:%hu%n",
    args=0x536fd0 <SrsFileLog::trace(char const*, _SrsContextId, char const*, ...) >) at
iovsscanf.c:41

```

SrsRtcPublishStream::send_periodic_twcc()

实际发送的是205 RTP-FEED BACK

```

#0      SrsRtcPublishStream::send_periodic_twcc      (this=0x1089310)      at
src/app/srs_app_rtc_conn.cpp:1412
#1 0x0000000000605962 in SrsRtcPublishTwccTimer::on_timer (this=0x1087010,
interval=100000)
    at src/app/srs_app_rtc_conn.cpp:940

```

```
#2 0x00000000005ec66f in SrsFastTimer::cycle (this=0xf4a6f0) at
src/app/srs_app_hourglass.cpp:204
#3 0x0000000000536712 in SrsFastCoroutine::cycle (this=0xf4a750) at
src/app/srs_app_st.cpp:270
#4 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0xf4a750) at
src/app/srs_app_st.cpp:285
#5 0x00000000006a6bc9 in _st_thread_main () at sched.c:363
```

SrsRtcpTWCC::do_encode 细节实现。

SrsRtcConnection::check_send_nacks

检测是否有丢包的现象，如果有丢包则请求

```
#0 SrsRtcConnection::check_send_nacks (this=0x108e900, nack=0x10a29a0,
ssrc=3583944678, sent_nacks=@0xf6006c: 0,
    timeout_nacks=@0xf600a4: 0) at src/app/srs_app_rtc_conn.cpp:2351
#1 0x000000000065210a in SrsRtcpRecvTrack::do_check_send_nacks (this=0x1092930,
timeout_nacks=@0xf600a4: 0)
    at src/app/srs_app_rtc_source.cpp:2328
#2 0x0000000000652726 in SrsRtcpVideoRecvTrack::check_send_nacks (this=0x1092930) at
src/app/srs_app_rtc_source.cpp:2429
#3 0x0000000000608438 in SrsRtcpPublishStream::check_send_nacks (this=0x1089260) at
src/app/srs_app_rtc_conn.cpp:1378
#4 0x0000000000609a6e in SrsRtcConnectionNackTimer::on_timer (this=0x10963d0,
interval=20000)
    at src/app/srs_app_rtc_conn.cpp:1670
#5 0x00000000005ec66f in SrsFastTimer::cycle (this=0xf4a610) at
src/app/srs_app_hourglass.cpp:204
#6 0x0000000000536712 in SrsFastCoroutine::cycle (this=0xf4a670) at
src/app/srs_app_st.cpp:270
#7 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0xf4a670) at
src/app/srs_app_st.cpp:285
#8 0x00000000006a6bc9 in _st_thread_main () at sched.c:363

#0 SrsRtcConnectionNackTimer::SrsRtcConnectionNackTimer (this=0x1149e20,
p=0x108e900) at src/app/srs_app_rtc_conn.cpp:1640
#1 0x0000000000609e4e in SrsRtcConnection::SrsRtcConnection (this=0x108e900,
s=0xf4f500, cid=...)
```

```
at src/app/srs_app_rtc_conn.cpp:1705
#2 0x000000000064433d in SrsRtcServer::create_session (this=0xf4f500, ruc=0x10b7500,
local_sdp=..., psession=0x10b6a08)
at src/app/srs_app_rtc_server.cpp:482
#3 0x000000000065d5ac in SrsGoApiRtcPublish::do_serve_http (this=0x1051450,
w=0x10b7bd0, r=0x108c8f0, res=0x108a8f0)
at src/app/srs_app_rtc_api.cpp:413
#4 0x000000000065c3c0 in SrsGoApiRtcPublish::serve_http (this=0x1051450, w=0x10b7bd0,
r=0x108c8f0)
at src/app/srs_app_rtc_api.cpp:287
#5 0x00000000004b8dad in SrsHttpServeMux::serve_http (this=0xf4f260, w=0x10b7bd0,
r=0x108c8f0)
at src/protocol/srs_http_stack.cpp:730
#6 0x00000000004b9bea in SrsHttpCorsMux::serve_http (this=0x10973a0, w=0x10b7bd0,
r=0x108c8f0)
at src/protocol/srs_http_stack.cpp:878
#7 0x00000000005a12c7 in SrsHttpConn::process_request (this=0x109a980, w=0x10b7bd0,
r=0x108c8f0, rid=1)
at src/app/srs_app_http_conn.cpp:250
#8 0x00000000005a0f22 in SrsHttpConn::process_requests (this=0x109a980,
preq=0x10b7ca8) at src/app/srs_app_http_conn.cpp:223
#9 0x00000000005a0ac2 in SrsHttpConn::do_cycle (this=0x109a980) at
src/app/srs_app_http_conn.cpp:177
#10 0x00000000005a04e5 in SrsHttpConn::cycle (this=0x109a980) at
src/app/srs_app_http_conn.cpp:122
#11 0x0000000000536712 in SrsFastCoroutine::cycle (this=0x1087b10) at
src/app/srs_app_st.cpp:270
```

播放码流

处理rtcp包

SrsRtcPlayStream::on_rtcp_rr() 尚没有做处理

SrsRtcPlayStream::on_rtcp_nack() 丢包重传

SrsRtcPlayStream::on_rtcp_ps_feedback() 主要处理了 kPLI, I帧

SrsRtcConnection::on_rtcp_feedback_remb()没有处理

SrsRtcConnection::on_rtcp_feedback_twcc() 没有处理

SrsRtcPlayStream::on_rtcp_nack()

```
srs_error_t SrsRtcPlayStream::on_rtcp(SrsRtcpCommon* rtcp)
{
    if(SrsRtcpType_rr == rtcp->type()) {
        SrsRtcpRR* rr = dynamic_cast<SrsRtcpRR*>(rtcp);
        return on_rtcp_rr(rr);
    } else if(SrsRtcpType_rtpfb == rtcp->type()) {
        //currently rtpfb of nack will be handle by player. TWCC will be handled by
SrsRtcConnection
        SrsRtcpNack* nack = dynamic_cast<SrsRtcpNack*>(rtcp);
        return on_rtcp_nack(nack);
    } else if(SrsRtcpType_psfb == rtcp->type()) {
        SrsRtcpPsfbCommon* psfb = dynamic_cast<SrsRtcpPsfbCommon*>(rtcp);
        return on_rtcp_psf_feedback(psfb);
    } else if(SrsRtcpType_xr == rtcp->type()) {
        SrsRtcpXr* xr = dynamic_cast<SrsRtcpXr*>(rtcp);
        return on_rtcp_xr(xr);
    } else if(SrsRtcpType_bye == rtcp->type()) {
        // TODO: FIXME: process rtcp bye.
        return srs_success;
    } else {
        return srs_error_new(ERROR_RTC_RTCP_CHECK, "unknown rtcp type=%u", rtcp-
>type());
    }
}
```

```
(gdb) n
799      vector<uint16_t> seqs = rtcp->get_lost_sns();
(gdb) n
800      if((err = target->on_recv_nack(seqs)) != srs_success) {
(gdb) print seqs
$1 = std::vector of length 6, capacity 8 = {3326, 3327, 3328, 3329, 3330, 3331}
(gdb) n
```

#0 SrsRtcPlayStream::on_rtcp_nack (this=0x112e440, rtcp=0x13e2f50) at

src/app/srs_app_rtc_conn.cpp:804

#1 0x0000000000604a0d in SrsRtcPlayStream::on_rtcp (this=0x112e440, rtcp=0x13e2f50) at
src/app/srs_app_rtc_conn.cpp:725

#2 0x000000000060c486 in SrsRtcConnection::dispatch_rtcp (this=0x1098690,
rtcp=0x13e2f50) at src/app/srs_app_rtc_conn.cpp:2115

```
#3 0x000000000060bb7e in SrsRtcConnection::on_rtcp (this=0x1098690, data=0x1064250
"\201", <incomplete sequence \315>,
    nb_data=54) at src/app/srs_app_rtc_conn.cpp:2026
#4 0x0000000000643dba in SrsRtcServer::on_udp_packet (this=0xf4f500,
    skt=0x7ffff7fe4b80) at src/app/srs_app_rtc_server.cpp:430
#5 0x000000000005de8a4 in SrsUdpMuxListener::cycle (this=0x10410a0) at
src/app/srs_app_listener.cpp:636
#6 0x00000000000536712 in SrsFastCoroutine::cycle (this=0x10511e0) at
src/app/srs_app_st.cpp:270
#7 0x000000000005367a8 in SrsFastCoroutine::pfn (arg=0x10511e0) at
src/app/srs_app_st.cpp:285
```

SrsRtcSendTrack::on_nack

把要发送的包插入队列中，以备后续客户端请求重传时可以取出来。本质就是缓存已发送的包，是一个 fifo，超过最大队列时，释放掉最早的数据。

```
#0 SrsRtcSendTrack::on_nack (this=0x109daf0, ppkt=0x124a1a8) at
src/app/srs_app_rtc_source.cpp:2516
#1 0x000000000060449a in SrsRtcPlayStream::send_packet (this=0x1138d60,
    pkt=@0x124a1a8: 0x11310a0)
    at src/app/srs_app_rtc_conn.cpp:680
#2 0x0000000000603df8 in SrsRtcPlayStream::cycle (this=0x1138d60) at
src/app/srs_app_rtc_conn.cpp:608
#3 0x00000000000536712 in SrsFastCoroutine::cycle (this=0x1099ab0) at
src/app/srs_app_st.cpp:270
#4 0x000000000005367a8 in SrsFastCoroutine::pfn (arg=0x1099ab0) at
src/app/srs_app_st.cpp:285
#5 0x00000000006a6bc9 in _st_thread_main () at sched.c:363
#6 0x00000000006a7446 in st_thread_create (start=0x0, arg=0x1099ab0, joinable=0,
    stk_size=18059008) at sched.c:694
#7 0x0000000000601d71 in SrsRtcPLIWorker::start (this=0x1096330) at
src/app/srs_app_rtc_conn.cpp:325
#8 0x0000000000603625 in SrsRtcPlayStream::start (this=0x1138d60) at
src/app/srs_app_rtc_conn.cpp:533
#9 0x000000000060cad8 in SrsRtcConnection::on_connection_established (this=0x1098f00)
at src/app/srs_app_rtc_conn.cpp:2205
```

#10 0x00000000006013dc in SrsSecurityTransport::on_dtls_handshake_done (this=0x124bd30) at src/app/srs_app_rtc_conn.cpp:164
#11 0x00000000006355df in SrsDtlsServerImpl::on_handshake_done (this=0x11377b0) at src/app/srs_app_rtc_dtls.cpp:952
#12 0x0000000000633e55 in SrsDtlsImpl::do_handshake (this=0x11377b0) at src/app/srs_app_rtc_dtls.cpp:619
#13 0x000000000063379d in SrsDtlsImpl::do_on_dtls (

SrsRtcPlayStream::on_rtcp_ps_feedback请求I帧

#0 SrsRtcPlayStream::on_rtcp_ps_feedback (this=0x111a350, rtcp=0x1120b70) at src/app/srs_app_rtc_conn.cpp:808
#1 0x0000000000604a70 in SrsRtcPlayStream::on_rtcp (this=0x111a350, rtcp=0x1120b70) at src/app/srs_app_rtc_conn.cpp:728
#2 0x000000000060c486 in SrsRtcConnection::dispatch_rtcp (this=0x1256ac0, rtcp=0x1120b70) at src/app/srs_app_rtc_conn.cpp:2115
#3 0x000000000060bb7e in SrsRtcConnection::on_rtcp (this=0x1256ac0, data=0x1064250 "\201", <incomplete sequence \316>, nb_data=26) at src/app/srs_app_rtc_conn.cpp:2026
#4 0x0000000000643dba in SrsRtcServer::on_udp_packet (this=0xf4f500, skt=0x7ffff7fe4b80) at src/app/srs_app_rtc_server.cpp:430
#5 0x00000000005de8a4 in SrsUdpMuxListener::cycle (this=0x10410a0) at src/app/srs_app_listener.cpp:636
#6 0x0000000000536712 in SrsFastCoroutine::cycle (this=0x10511e0) at src/app/srs_app_st.cpp:270
#7 0x00000000005367a8 in SrsFastCoroutine::pfn (arg=0x10511e0) at src/app/srs_app_st.cpp:285

SrsRtcConnection::on_rtcp_feedback_remb没有处理

#0 SrsRtcConnection::on_rtcp_feedback_remb (this=0x10a7cf0, rtcp=0x13a9da0) at src/app/srs_app_rtc_conn.cpp:2130
#1 0x000000000060bf21 in SrsRtcConnection::dispatch_rtcp (this=0x10a7cf0, rtcp=0x13a9da0) at src/app/srs_app_rtc_conn.cpp:2052
#2 0x000000000060bb7e in SrsRtcConnection::on_rtcp (this=0x10a7cf0, data=0x1064250 "\201", <incomplete sequence \311>, nb_data=70) at src/app/srs_app_rtc_conn.cpp:2026
#3 0x0000000000643dba in SrsRtcServer::on_udp_packet (this=0xf4f500, skt=0x7ffff7fe4b80) at src/app/srs_app_rtc_server.cpp:430

```
#4 0x000000000005de8a4 in SrsUdpMuxListener::cycle (this=0x10410a0) at
src/app/srs_app_listener.cpp:636
#5 0x00000000000536712 in SrsFastCoroutine::cycle (this=0x10511e0) at
src/app/srs_app_st.cpp:270
#6 0x000000000005367a8 in SrsFastCoroutine::pfn (arg=0x10511e0) at
src/app/srs_app_st.cpp:285
#7 0x000000000006a6bc9 in _st_thread_main () at sched.c:363
#8 0x000000000006a7446 in st_thread_create (start=0x7fffffffdeb0, arg=0x12000,
joinable=32767, stk_size=-152012000)
```

回发rtcp包

SrsRtcPlayStream::do_request_keyframe

名词解释补充

第一类：关键帧请求

主要包括SLI / PLI / FIR，作用是在关键帧丢失无法解码时，请求发送方重新生成并发送一个关键帧。

这本质是一种重传，但是跟传输层的重传的区别是，它重传是最新生成的帧。

PLI 是Picture Loss Indication，SLI 是Slice Loss Indication。

发送方接收到接收方反馈的PLI或SLI需要重新让编码器生成关键帧并发送给接收端。

FIR 是Full Intra Request，这里面Intra的含义可能很多人不知道。

Intra的含义是图像内编码，不需要其他图像信息即可解码；Inter指图像间编码，解码需要参考帧。

故Intra Frame其实就是指I帧，Inter Frame指P帧或B帧。

那么为什么在PLI和SLI之外还需要一个FIR呢？

原因是使用场景不同，FIR更多是在一个中心化的Video Conference中，新的参与者加入，就需要发送一个FIR，其他的参与者给他发送一个关键帧这样才能解码，

而PLI和SLI的含义更多是在发生丢包或解码错误时使用。

第二类：重传请求

主要包括RTX / NACK / RPSI

这个重传跟关键帧请求的区别是它可以要求任意帧进行重传

第三类：码率控制

主要包括REMB / TMMBR / TMMBN

TMMBR是Temporal Max Media Bitrate Request，表示临时最大码率请求。表明接收端当前带宽受限，告诉发送端控制码率。

REMB是ReceiverEstimated Max Bitrate，接收端估计的最大码率。

TMMBN是Temporal Max Media Bitrate Notification

参考

WebRTC(7) 实时视频传输中的RTCP协议 <https://blog.csdn.net/wangruihit/article/details/47041515>

RTCP中SR和RR的简介与区别 <https://www.freession.com/article/2916527752/>

技术解码 | WebRTC中RTCP使用及相关指标计算 <https://mp.weixin.qq.com/s/c9AYDyJ9-oeZvmCgP1cAUQ>

