Doubango voip 框架分析之多媒体部分

author： cyclecnetury 李德平

序言

RTP 提供带有实时特性的端对端数据传输服务，传输的数据如：交互式的音频和视频。那些服务包括有效载荷类型定义，序列号，时间戳和传输监测控制。应用程序在 UDP 上运行 RTP 来使用它的多路技术和 checksum 服务。2 种协议都提供传输协议的部分功能。

RTP 本身没有提供任何的机制来确保实时的传输或其他的服务质量保证，而是由低层的服务来完成。它不保证传输或防止乱序传输，它不假定下层网络是否可靠，是否按 顺序传送数据包。RTP 包含的序列号允许接受方重构发送方的数据包顺序，但序列号也用来确定一个数据包的正确位置，例如，在视频解码的时候不用按顺序的对 数据包进行解码。

1. 介绍

doubango 框架中 tinyRTP 文件夹实现 RTP/RTCP/RTSP 协议栈，目前只实现了 RTP ，RTCP;RTSP 还没实现。Rtp 用来在网络上传输音频视频，协议栈实现时主要在音视频包的封装，拆包。

2. rtp 包由消息头及消息体组成，消息头的结构封装

typedef struct trtp_rtp_header_s　　文件 trtp_rtp_header.h

```
{
    TSK_DECLARE_OBJECT;
    /* RFC 3550 section 5.1 - RTP Fixed Header Fie1ds
     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
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |V=2|P|X|  CC   |M|     PT      |       sequence number         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                           timestamp                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |           synchronization source (SSRC) identifier            |
    +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
    |            contributing source (CSRC) identifiers             |
    |                               ....                            |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    */
    unsigned version:2;
```

//版本(V)：2 比特 此域定义了 RTP 的版本。此协议定义的版本是 2。(值 1 被 RTP 草案版本使用，值 0 用在最初"vat"语音工具使用的协议中。)

```
    unsigned padding:1;
```

//填充(P)：1 比特 若填料比特被设置，则此包包含一到多个附加在末端的填充比特，填充比特不算作负载的一部分。填充的最后一个字节指明可以忽略多少个填充比特。填充可能用于某些具有固定长度的加密算法，或者用于在底层数据单元中传输多个 RTP 包。

```
    unsigned extension:1;
//扩展
```

//扩展(X)：1 比特 若设置扩展比特，固定头(仅)后面跟随一个头扩展。

```
    unsigned csrc_count:4;


    unsigned marker:1;
  //标志位
    unsigned payload_type:7;
 //负载类型，即承载的语音编码类型
```

//负载类型(PT)：7 比特 此域定义了负载的格式，由具体应用决定其解释。协议可以规定负载类型码和负载格式之间一个默认的匹配。其他的负载类型码可以通过非 RTP 方法动态定义。RTP 发送端在任意给定时间发出一个单独的 RTP 负载类型；此域不用来复用不同的媒体流。

```
    uint16_t seq_num;
```

//序列号，重新组包

//序列号（sequence number）：16 比特 每发送一个 RTP 数据包，序列号加 1，接收端可以据此检测丢包和重建包序列。序列号的初始值是随机的(不可预测)，以使即便在 源本身不加密时(有时包要通过翻译器，它会这样做)，对加密算法泛知的普通文本攻击也会更加困难。

```
    uint32_t timestamp;
```

//时间戳，负责流同步

```
    uint32_t ssrc;
```

//同步源标识，32 比特 用以识别同步源。标识符被随机生成，以使在同一个 RTP 会话期中没有任何两个同步源有相同的 SSRC 识别符。尽管多个源选择同一个 SSRC 识别符的概率很低，所有 RTP 实现工具都必须准备检测和解决冲突。若一个源改变本身的源 传输地址，必须选择新的 SSRC 识别符，以避免被当作一个环路源。

```
    uint32_t csrc[15];
```

//贡献源标识

```
}
trtp_rtp_header_t;
```

3. rtp 包结构，文件 trtp_rtp_packet.h

```
typedef struct trtp_rtp_packet_s
4.  {
5.    TSK_DECLARE_OBJECT;
6.
7.    trtp_rtp_header_t* header;  //包头
8.
9.    struct{
10.       void* data;
11.       const void* data_const;
12.       tsk_size_t size;
13.       } payload;          //负载，即承载内容
14.
15.   /* extension header as per RFC 3550 section 5.3.1 */
16.   struct{
17.       void* data;
```

```
18.       tsk_size_t size; /* contains the first two 16-bit fields */
19.   } extension;
20. }
21. trtp_rtp_packet_t;
```

4. rtp 包的控制

上面两个结构用来标示一个 rtp 包，同时提供了包的解析，创建等函数。
结构 trtp_manager_s 负责 rtp.rtcp 包的管理，是更高层的抽象，上层应用直接通过 trtp_manager_s 提供的 api 控制 rtp 包，比如在网络上发送音频数据，在音频 session 结构中包含 trtp_manager_s 用来管理经过封装的 rtp 包。

```
/** RTP/RTCP manager */
typedef struct trtp_manager_s
{
    TSK_DECLARE_OBJECT;

    struct{
        uint16_t seq_num;
        uint32_t timestamp;
        uint32_t ssrc;
        uint8_t payload_type;

        char* remote_ip;
        tnet_port_t remote_port;
        struct sockaddr_storage remote_addr;

        char* public_ip;
        tnet_port_t public_port;

        const void* callback_data;
        trtp_manager_rtp_cb_f callback;
    } rtp;

    struct{
        char* remote_ip;
        tnet_port_t remote_port;
        struct sockaddr_storage remote_addr;
        tnet_socket_t* local_socket;

        char* public_ip;
        tnet_port_t public_port;

        const void* callback_data;
        trtp_manager_rtcp_cb_f callback;
    } rtcp;

    char* local_ip;
    tsk_bool_t ipv6;
    tsk_bool_t started;
    tsk_bool_t enable_rtcp;
    tsk_bool_t socket_disabled;
    tnet_transport_t* transport;
}
trtp_manager_t;
```

5. tdav 是音视频会话的抽象层，负责传输层的启动，音频会话，视频会话，各种编码的注册。

对于音频/视频会话(session)被 tmedia_session_mgr_t 管理，而 tmedia_session_mgr_t 则具体由 sip 信令控制会话的状态。比如 sip 客户端请求时通过 tmedia_session_mgr_t 构造自己的 sdp 信息要借助此结构，当客户端对 invite 作 ACK 应答时同样要指定自己的媒体信息。整个 rtp 流的启动入口都由 tmedia_session_mgr_t 控制。
各种媒体会话以插件的形式注册，如音频会话在启动时注册到 tmedia_session_mgr_t 的插件链表，并绑定 start,stop,prepare 回调。tmedia_session_mgr_t 为 sip 信令控制媒体流的接口。

tmedia_session_plugin_def_t 为音频视频抽象接口，指定回调。如音频会话，内部会实现相应的回调函数。

```
/** Virtual table used to define a session plugin */
typedef struct tmedia_session_plugin_def_s
{
    //! object definition used to create an instance of the session
    const tsk_object_def_t* objdef;

    //! the type of the session
    tmedia_type_t type;
    //! the media name. e.g. "audio", "video", "message", "image" etc.
    const char* media;

    int (*set) (tmedia_session_t* , const tmedia_param_t*);
    int (* prepare) (tmedia_session_t* );
    int (* start) (tmedia_session_t* );
    int (* pause) (tmedia_session_t* );
    int (* stop) (tmedia_session_t* );

    struct{ /* Special case */
        int (* send_dtmf) (tmedia_session_t*, uint8_t );
    } audio;

    const tsdp_header_M_t* (* get_local_offer) (tmedia_session_t* );
    /* return zero if can handle the ro and non-zero otherwise */
    int (* set_remote_offer) (tmedia_session_t* , const tsdp_header_M_t* );
}
tmedia_session_plugin_def_t;
```

tmedia_session_t 为会话的抽象层，包含 tmedia_session_plugin_def_t，

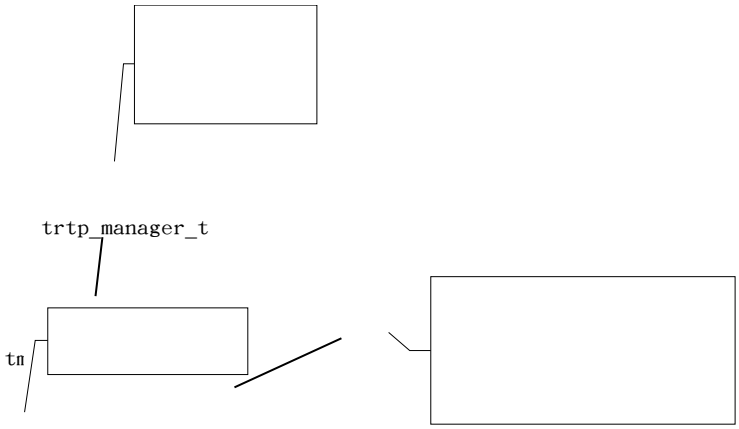
```
/** Base object used for all media sessions */
typedef struct tmedia_session_s
{
    TSK_DECLARE_OBJECT;

    //! unique id. If you want to modifiy this field then you must use @ref tmedia_session_get_unique_id()
    uint64_t id;
    //! session type
    tmedia_type_t type;
    //! list of codecs managed by this session
    tmedia_codecs_L_t* codecs;
    //! negociated codec
    tmedia_codecs_L_t* neg_codecs;
    //! whether the ro have been prepared (up to the manager to update the value)
    tsk_bool_t ro_changed;
    //! whether the session have been initialized (up to the manager to update the value)
```

```
    tsk_bool_t initialized;
    //! whether the session have been prepared (up to the manager to update the value)
    tsk_bool_t prepared;
    //! QoS
    tmedia_qos_tline_t* qos;
    //! bandwidth level
    tmedia_bandwidth_level_t bl;

    struct{
        tsdp_header_M_t* lo;
        tsdp_header_M_t* ro;
    } M;

    //! plugin used to create the session
    const struct tmedia_session_plugin_def_s* plugin;
}
tmedia_session_t;
```



使用过程:

tdav_init 注册音频，视频，多媒体 session；注册支持的编码类型，注册支持的媒体信息承载类型（文本，流等）。

tdav_init -> register sessions, codecs.

⇓

tmedia_session_mgr_create -> tmedia_session_mgr_ctor ，sessions,qos,sdp.

⇓

_tmedia_session_mgr_load_sessions, 创建音视频会话。

⇓

tmedia_session_create ,创建具体会话插件类型,tdav_session_video/audio_ctor

⇓

tmedia_session_init ，初始化

⇓

tmedia_session_load_codecs， 此会话支持的编码类型

⇓

tmedia_codec_create ，穿件具体编码类型。

创建过程

准备阶段

trtp_manager_prepare ，指定传输层接收数据回调 trtp_transport_layer_cb

tdav_session_audio_prepare，trtp_manager_create，trtp_manager_set_rtp_callback

tnet_transport_create

tnet_transport_set_callback

启动

tmedia_session_mgr_start（），启动所有上面创建的会话类型，启动之前一定要设置 sdp 信息

（Starts the session manager by starting all underlying sessions.You should set both remote and local offers before calling this

session->plugin->start（），如视频会话启动 ，tdav_session_video/audio.c

trtp_manager_set_rtp_remote ，设置对端 ip,port,后续发送 rtp 包时构造包头用

trtp_manager_set_payload_type ，设置此次会话用什么编码类型，编码类型通过协商后选择最佳

trtp_manager_start， 启动 rtp,rtcp 包管理，

tnet_transport_start ，启动传输层线程，绑定 socket 地址，开始接收 udp 数据，

tnet_transport_mainthread

请求或响应中 sdp 与 codec 匹配过程

tmedia_session_match_codec ->tmedia_codec_match_fmtp->tdav_codec_h264_fmtp_match->
tdav_codec_h264_get_profile(根据 fmt 获取对方的 profile 版本)，

当发起外乎请求时 codec 与 sdp 处理关系，
发起 invite 或对方更改媒体信息时要把 codec 信息加载到 sdp 消息体中，
对于 video,audio 过程是一样的。

(video session from codecs to sdp)
tdav_session_video_get_lo
|
tsdp_header_M_create（创建 sdp 媒体头）

|
tmedia_session_match_codec （此函数最终会返回一个协商成功的编码类型）

对于 h264 编码格式， 此函数内部调用过程， 遍历协议栈初始化时指定的编码链表，用此次请求的 sdp 消息体中的编码与自己的编码链表比较。->tmedia_codec_match_fmtp->tdav_codec_h264_fmtp_match->
tdav_codec_h264_get_profile

tmedia_session_match_codec 返回协商成功的编码列表（即双方都支持的编码类型列表）后复制给协议栈，
self->neg_codecs = tmedia_session_match_codec
然后调用 tmedia_codec_video_set_callback 设置此编码类型对应的回调函数，当想发送 rtp 包时直接触发此回调函数即可完成发送 rtp 包的任务。
tmedia_codec_video_set_callback((tmedia_codec_video_t*)TSK_LIST_FIRST_DATA(self->neg_codecs),
tdav_session_video_raw_cb, self);

tdav_session_video_raw_cb 为具体的毁掉函数，内部为调用 trtp_manager_send_rtp ，发送 rtp 包。

值得注意的是传给函数的 tdav_session_video_raw_cb 数据只是未经过加工成 rtp 包的裸数据，
tdav_session_video_raw_cb 内部调用 trtp_manager_send_rt ，由 trtp_manager_send_rt 来把数据加工成 rtp 包，
然后调用传输层发送到网络上。

```
/* Encapsulate raw data into RTP packet and send it over the network
* Very IMPORTANT: For voice packets, the marker bits indicates the beginning of a talkspurt */
int trtp_manager_send_rtp(trtp_manager_t* self, const void* data, tsk_size_t size, uint32_t duration,
tsk_bool_t marker, tsk_bool_t last_packet)
```

trtp_manager_send_rtp 内部又具体调用 trtp_rtp_packet_create，创建 rtp 格式的数据包，包括 rtp 消息头的创建，初始化默认参数(version， marker, payload_type,seq_num 等)。然后调用 trtp_rtp_packet_serialize 把 rtp 包序列化到一个 buffer 中。
trtp_manager_send_rtp 最后调用 tnet_sockfd_sendto 传输层函数完成实际发送到网络上。

回到设置 tmedia_codec_video_set_callback 完毕后，tdav_session_video_get_lo 调用 tmedia_codec_to_sdp 把协商后的编码类型的信息转换成 sdp 格式的信息。
tmedia_codec_to_sdp(self->neg_codecs, self->M.lo); 保存到 M.lo 属性，即本地的媒体信息。
tmedia_codec_to_sdp 分析：
此函数的功能即把协商后的编码链表放到协议栈的 sdp 属性中，这样以后发送 invite 请求时就可以直接用。

```
/**@ingroup tmedia_codec_group
* Serialize a list of codecs to sdp (m= line) message.<br>
* Will add: fmt, rtpmap and fmtp.
* @param codecs The list of codecs to convert
* @param m The destination
* @retval Zero if succeed and non-zero error code otherwise
*/
int tmedia_codec_to_sdp(const tmedia_codecs_L_t* codecs, tsdp_header_M_t* m)

TSK_DEBUG_INFO("Serialize a list of codecs to sdp (m= line) message\n");
    tsk_list_foreach(item, codecs){
```

遍历每个编码类型，添加 fmt, rtpmap 属性，fmtp 属性（tmedia_codec_get_fmtp ，对于 h264 格式即调用 tmedia_codec_h264_get_fmtp）

最后，tdav_session_video_get_lo 内部在属性 M.ro(即已经有请求的 sdp 信息)非空时考虑此请求是否为保持还是接回，通过设置 spd 属性，sendrecv，sendonly 来提示类型。最后，设置 Qos 信息。

流程 tdav_session_video_get_lo

```
          |
     tsdp_header_M_create (创建 sdp 媒体头)

          |
tmedia_session_match_codec
          |
 tmedia_codec_video_set_callback
         |
tmedia_codec_to_sdp
```

但是

tdav_session_video_get_lo 又是由谁触发的呢？tdav_session_video_get_lo 为某一具体 session 的回调，
比如视频的 session 回调，音频的回调，视频，音频的 session 以 plugin 的方式挂在到 session 中。

```
/** Virtual table used to define a session plugin */
typedef struct tmedia_session_plugin_def_s
{
    //! object definition used to create an instance of the session
    const tsk_object_def_t* objdef;

    //! the type of the session
    tmedia_type_t type;
    //! the media name. e.g. "audio", "video", "message", "image" etc.
    const char* media;

    int (*set) (tmedia_session_t* , const tmedia_param_t*);
    int (* prepare) (tmedia_session_t* );
    int (* start) (tmedia_session_t* );
    int (* pause) (tmedia_session_t* );
    int (* stop) (tmedia_session_t* );

    struct{ /* Special case */
        int (* send_dtmf) (tmedia_session_t*, uint8_t );
    } audio;

    const tsdp_header_M_t* (* get_local_offer) (tmedia_session_t* );
    /* return zero if can handle the ro and non-zero otherwise */
    int (* set_remote_offer) (tmedia_session_t* , const tsdp_header_M_t* );
}
tmedia_session_plugin_def_t;
```

tdav_session_video_get_lo 即为 get_local_offer 的具体回调。
get_local_offer 被 tmedia_session_get_lo 调用。tmedia_session_get_lo 又被 tmedia_session_mgr_get_lo】
调用，正是上面提到的 tmedia_session_mgr 为管理 session 的抽象接口，用来与 sip 信令交互。
整个流程为：

```
tmedia_session_mgr
          |
tmedia_session_get_lo
          |
tdav_session_video_get_lo
         |
tsdp_header_M_create (创建 sdp 媒体头)
           |
tmedia_session_match_codec
           |
tmedia_codec_video_set_callback
          |
tmedia_codec_to_sdp
```

tmedia_session_mgr_get_lo 又被谁触发呢？

刚才说了，是由 sip 协议栈调用的，具体有这样几个与 sdp 协商有关的 sip 点，我们知道，invite 请求以及 200 ok 应答，
183 响应,100 响应的确认(prack)中有 sdp 信息：

（1）发送或者更新请求(invite)

```
send_INVITEorUPDATE
// send INVITE/UPDATE request
int send_INVITEorUPDATE(tsip_dialog_invite_t *self, tsk_bool_t is_INVITE, tsk_bool_t force_sdp)
```

(2)prack 响应

```
// Send PRACK
int send_PRACK(tsip_dialog_invite_t *self, const tsip_response_t* rlxx)
```

(3) // Send ACK

```
int send_ACK(tsip_dialog_invite_t *self, const tsip_response_t* r2xxINVITE)
```

初始请求中没有 sdp 信息，在 ack 中需要携带 sdp 信息

（4）发送响应时

```
/ Send any response
int send_RESPONSE(tsip_dialog_invite_t *self, const tsip_request_t* request, short code, const char*
phrase, tsk_bool_t force_sdp)
```

2. 处理请求中的 sdp 信息过程

```
tsip_dialog_invite_process_ro
          |
 tmedia_session_mgr_set_ro
```

tsip_dialog_invite_process_ro 为 sip 信令中处理 sdp 信息的入口，在状态机的回调中适时调用。比如在 保持状态转到接回状态。

tsip_dialog_invite_process_ro 会初始化 mgr ，启动，tmedia_session_mgr_create，tmedia_session_mgr_set_ro，tmedia_session_mgr_set_natt_ctx，tmedia_session_mgr_start。