doubango 框架协议栈分析

author : cyclecentury 李德平

1.tinysip 介绍：

兼容性： SIP(RFC 3261) 以及 3GPPIMS/LTE (TS 24.229) implementation

依赖 tinySAK,tinyNET, tinySDP, tinyMEDIA, tinyHTTP and tinyIPSec.

2.SIP 协议 - tinysip 的实现机制

SIP 是一个分层结构的协议，这意味着它的行为是根据一组平等独立的处理阶段来描述，每一阶段之间只是松耦合。协议分层描述是为了表达，从而允许功能的描述可在一个部分跨越几个元素。它不指定任何方式的实现。当我们说某元素包含某层，我们是指它顺从该层定义的规则集。

不是协议规定的每个元素都包含各层。而且，由 SIP 规定的元素是逻辑元素，不是物理元素。一个物理实现可以选择作为不同的逻辑元素，甚至可能在一个个事务的基础上。

SIP 的最底层是语法和编码。它的编码使用增强 Backus-Nayr 形式语法（BNF）来规定。

第二层是传输层。它定义了网络上一个客户机如何发送请求和接收响应以及一个服务器如何接收请求和发送响应。所有的 SIP 元素包含传输层。

第三层是事务层。事务是 SIP 的基本元素。一个事务是由客户机事务发送给服务器事务的请求（使用传输层），以及对应该请求的从服务器事务发送回客户机的所有响应组成。事务层处理应用层重传，匹配响应到请求，以及应用层超时。任何用户代理客户机（UAC）完成的任务使用一组事务产生。用户代理包含一个事务层，有状态的代理也有。无状态的代理不包含事务层。事务层具有客户机组成部分（称为客户机事务）和服务器组成部分（称为服务器事务），每个代表有限的状态机，它被构造来处理特定的请求。

事务层之上的层称为事务用户（TU）。每个 SIP 实体，除了无状态代理，都是事务用户。当一个 TU 希望发送请求，它生成一个客户机事务实例并且向它传递请求和 IP 地址，端口，和用来发送请求的传输机制。一个 TU 生成客户机事务也能够删除它。当客户机取消一个事务时，它请求服务器停止进一步的处

理，将状态恢复到事务初始化之前，并且生成特定的错误响应到该事务。这由 CANCEL 请求完成，它构成自己的事务，但涉及要取消的事务。

SIP 通过 EMAIL 形式的地址来标明用户地址。每一用户通过一等级化的 URL 来标识，它通过诸如用户电话号码或主机名等元素来构造（例如：SIP:vision-com.com）。因为它与 EMAIL 地址的相似性，SIPURLs 容易与用户的 EMAIL 地址关联。

SIP 提供它自己的可靠性机制从而独立于分组层，并且只需不可靠的数据包服务即可。SIP 可典型地用于 UDP 或 TCP 之上。

Register,Invite, Options …
Nat Tranversal
Dialog Layer
Transaction Layer
Parsing Layer
Transport Layer

sip 协议栈分层结构图

根据 sip 消息流向可以分为 incomingmessage 和 outgoing message,incoming 消息从 下到上，即 TransportLayer → Register,Invite, Options; outgoing message 消息流向与此相反。

3.根据以上定义,tinysip 分如下模块：

1).api 外部接口，对 sip 协议支持的方法的接口封装，协议栈提供的发起请求及接受请求对应的接口,包括 registar layer, presencelayer 等上层应用，当前版本支持如下请求：

REGISTER，SUBSCRIBE(订阅),MESSAGE(即时通信),PUBLISH（状态展示）,OPTIONS（查询服务器能力），INVITE（发起请求），Cancel（取消一个请求），
BYE（结束通话）。

2).Nat traversal ： Nat 穿越层,tinysip 目前支持 stun，turn 穿透。

3).Dialog，会话模块，一路呼叫的唯一标识,处于 sip 事务层之上。
4).parsers，sip 消息解析，处于 sip 语法层，解析从传输层传递的数据包为协议栈理解的结构。

5).transactions， 事务层，事务是一个请求以及与此请求相关的所有响应组成，用 transaction id 唯一标识，由于 sip 信令一般由 udp 承载，所以不能保证信息的可靠到达，所以事务层必须提供一种机制处理 udp 所不能提供的功能，这里一般通过定时器及一个有限状态机来实现。

6).transports ,传输层，即 udp, TCP, TLS, SCTP socket 系统调用系列，此层隐藏了所有传输层的细节，对于 incomingsip message ， 此层为 sip 消息的入口,对于 outgoingsip message, 此层为 sip 消息的出口。

4.doubango sip 协议栈使用流程：

1）.初始化

doubango sip 协议栈依赖于 tinyNET 模块，所以必须先调用 tnet_startup 函数初始化，退出时调用 tnet_cleanup 清除资源，初始化 sip

协议栈之前必须设置用户的域(realm 参见（1）)及用户的私有（IMPI（2））及共有标别（IMPU（3））,这些为 ims 引入的概念。

(1)realm 解释：reaml 为域名，用来作客户端认证用（authenticate）.必须是一个有效的 sipuri 如：sip:vision-com.com.cn，realm 为 sip 协议栈启动之前必须设置的选项，一旦协议栈启动 realm 就不可以更改，如果不填写 sip 代理服务器地址，则系统会用 realm 通过 dns NAPTR + SRV 或者 DHCP(还没实现)动态查找机制确定 sip 服务器地址。

(2)用户私有标识，为用户所属网络赋予的唯一值，用来做验证，为 IMS 中的概念，如果用 doubangosip 协议栈作为普通 sip 功能，即非 IMS 网络中的 sip 功能，此处的 impi 意义与 sip 协议栈中的验证域名相同，私有 ID 用在身份认证，授权等，主要是安全方面的作用。

（3）用户公有标识，ims 网络中一个 impi 可以对应多个 impu.，公有 ID 用在业务配置、计费等，主要是业务方面的作用。

更进一步解释：
IMPU，它更靠近业务层，用于标识业务签约关系，计费等，还表示用户身份以及用于路由，但它不能表示用户实际的位置信息，当然这个是对于非传统固话来说，传统固定方式下用户号码与位置存在绑定

关系，因为用户线接入是固定的。
然后是 IMPI，它是网络层的东东，用于表示用户和网络的签约关系，一般也可以唯一的表示一个终端。使用 IMPI，网络可以通过鉴权来识别用户是否可以使用网络。
最后是用户的联系地址，这个是注册的时候要带的，它是用户真实的寻址地址，IMPU 和联系地址分离才能支持移动性，对于传统 GSM 网络，漫游号起到了类似联系地址的作用，你也可以认为 GSM 中的位置更新也是一种注册流程。
注册后 S-CSCF 知道 IMPU 以及联系地址，当有人呼叫此 IMPU，根据联系地址即可找到实际的用户，因此注册维护了一个用户的寻址通路。

2）.创建以及启动

通过调用 tsip_stack_create 创建协议栈， 调用 tsip_stack_start 启动，

完整例子：

```
tsip_stack_handle_t*stack = tsk_null;

intret;

constchar* realm_uri = "sip:vision-com.com.cn";

constchar* impi_uri = "lideping@vision-com.com.cn";

constchar* impu_uri = "sip:bob@vision-com.com.cn";


//...必须先初始化 tnet 工具

tnet_startup();

// ...

//创建协议栈，指定回调函数，参数为域名，公有及私有标识。

stack= tsip_stack_create(sip_callback, realm_uri, impi_uri, impu_uri,

TSIP_STACK_SET_PASSWORD("yourpassword"),
```

```
//...other macros...
//此处初始化其他信息，比如代理服务器地址，编码信息等。
tsip_stack_start(stack) //启动协议栈
TSIP_STACK_SET_NULL());//用来终止传给 app_callback 的参数。
TSK_OBJECT_SAFE_FREE(stack);

tnet_cleanup();
//释放资源
//事件回调
intsip_callback(const tsip_event_t *sipevent)
{
//事件类型
switch(sipevent->type){
casetsip_event_register:
{ /*REGISTER */
break;
}
casetsip_event_invite:
{ /*INVITE */
```

```
break

}

casetsip_event_message:

{ /*MESSAGE */

break

}

casetsip_event_publish:

{/* PUBLISH */

break

}

casetsip_event_subscribe:

{ /*SUBSCRIBE */

break

}

casetsip_event_options:

{ /*OPTIONS */

break

}
```

casetsip_event_dialog:

{ /*Common to all dialogs */

break

}

/*case …*/

}

会话事件，协议栈事件放入事件队列，协议栈启动时创建一个线程不断从队列里取事件，然后通过调用事件回调通知上层用户。

5.代码分析

首先调用 tsip_stack_create 创建协议栈，tsip_stack_create 内部首先检查参数是否合法，然后创建协议栈结构 tsip_stack_t，设置 realm,IMPI and IMPU，初始化一些协议栈默认值。创建 SigComp 信令压缩模块（可选）创建 dns 处理模宽，DHCPcontext ，接下来创建上面提到的 sip 协议栈各层，分别调用 tsip_dialog_layer_create，tsip_transac_layer_create,tsip_transport_layer_create 创建会话层，事务层及传输层。至此，创建协议栈毕。

在真正启动协议栈之前，即 tsip_stack_create 与 tsip_stack_start 之间可调用协议栈提供的 api 初始化其他参数，然后调用 tsip_stack_start 启动协议栈，首先启动定时器线程，这里定时器主要在事务层提供状态机功调度功能。然后设置传输层类型，设置是否用 ipsec 把 sip 信令加密，然后如果协议栈是处于客户端模式并且代理服务器地址没有设置则用认证的域名查找代理服务器的地址（用 SNAPTR+SRV),然后设置 Runnable 回调，启动 run 线程，run 内部不断从消息队列里取消息，这里的消息是从传输层从下到上传送过来，最终串联到消息队列，然后调用协议栈创建时指定的回调 sip_callback，所有 incomingsip 消息以及媒体信息的改变最终都会走到此回调函数，此函数内部根据消息类型的不同调用相应的 handler，接下来启动 nat 穿越模块，设置 stun 地址。然后调用 tsip_transport_layer_start 启动传输层线程，在 sip 端口 5060 接收数据，最后，设置 stack->started= tsk_true; 至此协议栈启动完毕，各层在相应端口或状态机上监听，不断轮询到来的事件并处理。

驱动过程：

协议栈启动完毕后，对于每一个 incoming 及 outgoing 消息的入口不一样，下面分别分析对于呼入请求(incoming)及外乎请求(outgoing)的代码流程。

a.呼入请求

1）客户端传输层在 5060 端口上接收到 udp 包，语法层解析成识别的 sip 消息后传给事务层。
2）事务层锁住本地事务链表，根据 sip 消息的事务 id 在事务链查找是否存在匹配的事务，没有则创建。
3）一旦找到事务或创建新事务完毕，释放锁并把消息传递到会话层。
4）会话层收到 sip 消息后查找会话链，找不到则创建会话，同时根据消息类型（invite,ack 等）设置此消息的状态机，状态机内指定具体事件的回调。

b.呼出请求
1)构造外乎请求，包括消息头和消息体.
2)创建会话层，事务层，事务层调用传输层接口发出请求。

6. 外部编程接口

为了在 android 上层通过 java 访问 doubango 核心，imsdroid 对 doubango voip 框架做了面向对象封装，根据具体模块功能抽象成具体 Java 类供应用层使用，应用层通过 jni 访问 doubango 核心，同时，在 imsdroid2.0 版本中，根据 android 上应用层的架构抽象出一个类库，doubango-ngn-stack,利用此类库我们可以在 android 上自己开发一些客户端应用程序，包括语音，视频，即时通信，多媒体共享，会议等应用。Imsdroid2.0 即是构建在 doubango-ngn-stack 上的一个具体应用。

Doubango-ngn-stack 原理
doubango-ngn-stack 是对 doubangovoip 框架的一个 java 层封装，内部通过 java 本地调用技术实现(jni),这与 android 上的框架设计是相符的(如 java 类库提供的摄像头功能即依赖于底层驱动，上层通过 jni 访问底层驱动)，

doubango/bindings/java

用 SWIG 工具把 c/C++函数封装成 JAVA 中的类，目前 SWIG 已经可以支持 Python,Java, C#,Ruby，PHP,等语言。

7. 代码实例分析

注册过程

1）. 注册流程(java-->C++-->C)

register(NgnSipService.java)
|
register(NgnRegistrationSession.java)
|
register_(sipsession.cxx)
|
tsip_action_REGISTER(stip_api_register.c)

tsip_action_REGISTER
分三步：
_tsip_action_create 创建注册请求, tsip_dialog_layer_new 创建注册 session,为后续进入状态机作准备， tsip_dialog_fsm_act 进入状态机模式。

1._tsip_action_create

创建注册请求，这里只是创建一个抽象的请求，请求对应 sip 的方法，sip 协议定义了 register,invite,publish,subscribe, bye,message 等方法。
然后调用_tsip_action_set 初始化上层（java）传过来的参数。

2. 请求创建成功后创建会话层 tsip_dialog_layer_new
sip 协议中每个请求方法对应一个会话,即 session,此函数根据会话类型创建相应会话的 session.对于注册会话，会调用 tsip_dialog_register_create 创建会话层，然后把创建后的会话保存到协议栈的会话层链表。
（1）tsip_dialog_register_create（tsip_dialog_register.c）创建注册会话
内部 new 一个注册对象 tsip_dialog_register_def_t，构造函数中执行如下动作。
tsip_dialog_register_ctor 内部分三步：
a.tsip_dialog_init 初始化基本的会话信息，这里分客户端及服务器端。同时会创建此会话的状态机，并初始化状态机的状态。
b.tsk_fsm_set_callback_terminated 设置注册会话结束状态机回调。
c.tsip_dialog_register_init 初始化具体注册会话信息

tsip_dialog_register_client_init 初始化注册请求的客户端状态机回调。

这里实际上是当一个请求过程中，当请求从一个状态到另一个状态时应该调用的回调函数。
tsip_dialog_register_server_init 功能相同。

tsip_dialog_register_event_callback 设置从传输层过来的注册事件的回调，通知上层。

3. tsip_dialog_fsm_act，是所有 sip 会话开始进入状态机的入口。
参数为 会话，会话类型，sip 消息，请求。
此函数内部会调用状态机通用函数 tsk_fsm_act。执行一个具体的请求，同时可能改变相应会话的状态机的状态。
inttsk_fsm_act(tsk_fsm_t* self, tsk_fsm_action_id action, const void*cond_data1, const void* cond_data2, ...)
内部是一个循环，不断检测状态，根据状态调用相应的回调。

每个状态机都有一个初始状态作为运转入口，对于注册请求，入口为，

TSK_FSM_ADD_ALWAYS(_fsm_state_Started,_fsm_action_oREGISTER, _fsm_state_InProgress,tsip_dialog_register_Started_2_InProgress_X_oRegister,"tsip_dialog_register_Started_2_InProgress_X_oRegister"),

tsk_fsm_act 一开始会执行 tsip_dialog_register_Started_2_InProgress_X_oRegister 回调，
此回调是由状态_fsm_state_Started 到_fsm_state_InProgress 时执行的操作，
内部先更改自己的当前状态为_fsm_state_InProgress，这样给 tsk_fsm_act 提供了调用下一个状态的入口。
函数内部调用 tsip_dialog_register_send_REGISTER 创建事务层，初始化事务层状态机，最后调用传输层 socket 接口把请求发送出去。
由于 sip 根据请求的类型把事务层分为几种类型，包括客户端请求(invite)事务，客户端非请求事务(如 bye,register),服务器端请求事务，服务器端非请求事务。

所以对于注册请求，会创建非请求客户端事务层。
tsip_dialog_register_send_REGISTER---> tsip_dialog_request_new
--->tsip_dialog_request_send---->tsip_transac_layer_new--->tsip_transac_start
--->tsip_transac_nict_start,进入事务层状态机模式。
这里，在创建事务层时会设置事务层事件回调，tsip_transac_nict_event_callback。

比如发出 register 请求，服务器端给 200ok 响应。此时传输层会把此响应作为事件给事务层，事务层收到此事件，解析后进入事务层状态机。

所以对于 一次 sip 请求，有两个状态机在运转，一个为会话层状态机，一个为事务层状态机。

至此，一个 register 请求已经发送出去，会话层，事务层状态机都在运转，
此时，如果服务器返回注册成功，则会给客户端传输层发送 200OK 响应。
tsip_transport_layer_dgram_cb，tsip_transport_layer_handle_incoming_msg，
tsip_transac_layer_handle_incoming_msg，tsip_transac_layer_find_client

客户端传输层收到响应后会根据响应的事务 id 查找是否为已经创建的事务。
，找到后会根据此事务创建时指定的回调，调用相应事务层的回调函数，这里为
tsip_transac_nict_event_callback，此函数内部根据消息类型调用 tsip_transac_fsm_act 执行事务层状态机，比如 200ok ，会调用事务层创建时指定的回调。这里为
tsip_transac_nict_Trying_2_Completed_X_200_to_699。
内部调用会话层回调通知上层注册成功。tsip_dialog_register_event_callback
此函数根据状态调用相应会话层状态机，tsip_dialog_register_InProgress_2_Connected_X_2xx，修改状态机当前状态，为下一个状态作准备，最后调用 TSIP_DIALOG_REGISTER_SIGNAL 发射 注册成功事件给上层用户。
这里事件机制：TSIP_DIALOG_REGISTER_SIGNAL 通过 tsip_register_event_signal 创建一个注册事件，然后把此事件放入协议栈启动时的事件队列中，

tsip_stack_start 内部启动 run 线程处理协议栈事件。

```
/* ===Runnable === */
TSK_RUNNABLE(stack)->run= run;
if((ret= tsk_runnable_start(TSK_RUNNABLE(stack), tsip_event_def_t))){
stack_error_desc= "Failed to start timer manager";
TSK_DEBUG_ERROR("%s",stack_error_desc);
gotobail;
}
```

在 run 线程中不断扫描事件队列，pop 出一个事件，然后送给 sip 协议栈创建时指定的事件回调，从而把协议栈事件(事务层事件，会话层事件，协议栈事件)返回给上层用户。

至此，一次正常(没考虑异常，401 认证过程等)的注册流程分析完毕。

2.）外乎流程：

screenAV.java

publicstatic boolean makeCall(String remoteUri, NgnMediaType mediaType){

makeCall(StringremoteUri, NgnMediaType mediaType)
|
createOutgoingSession
|
makeCall(remoteUri); (NgnAVSession.java)
|

callAudioVideo
|
callAudioVideo(sipsession.cxx)
|
__droid_call(sipsession.cxx)
|
__droid_call_thread(sipsession.cxx)
|
tsip_action_INVITE(tsip_api_invite)

至此，由上层调用到底层协议栈。

同样分三步：
_tsip_action_create
tsip_dialog_layer_new
tsip_dialog_fsm_act

从创建会话层开始分析：

tsip_dialog_layer_new
tsip_dialog_invite_create
tsip_dialog_invite_ctor (tsip_dialog_invite.c)
构造函数内过程：
（1）tsip_dialog_init 创建相关头域，创建 invitesession 状态机并初始化状态。

（2）tsk_fsm_set_callback_terminated 设置 invite 会话状态机结束的回调函数 tsip_dialog_invite_OnTerminated。

（3）tsip_dialog_invite_init 初始化 invite 请求本身，具体如下：

a.tsip_dialog_invite_client_init 初始化服务器端 invite 会话信息，实际上为设置客户端 invite 会话的状态机。

Started-> (send INVITE) -> Outgoing-> Connected .....

状态转换过程中调用相应回调。

b.tsip_dialog_invite_server_init 设置服务器端 invite 会话状态机。

//Started -> (Bad Extendion) -> Terminated

//Started -> (Bad content) -> Terminated

//Started -> (Session Interval Too Small) -> Started

.....

c. tsip_dialog_invite_hold_init 设置 hold 状态机，3GPPTS 24.610: CommunicationHold。

d.tsip_dialog_invite_stimers_init ,设置 sessiontimer 状态机，rfc RFC

4028:SessionTimers

e.tsip_dialog_invite_qos_init,设置 qos 状态机，RFC 3312

f.初始化其他状态机。

TSIP_DIALOG(self)->callback= TSIP_DIALOG_EVENT_CALLBACK_F(tsip_dialog_invite_event_callback);

设置 invite 会话层事件回调，比如服务器响应，则会调用此回调，内部根据响应类型调用状态机，根据状态执行相应回调。

tsip_dialog_fsm_act，所有准备阶段完成后，第三步进入状态机模型，轮转吧。

第一个调用的状态机回调为 c0000_Started_2_Outgoing_X_oINVITE，

Started-> (oINVITE) -> Outgoing。

此函数内部：

首先调用 tmedia_session_mgr_create 初始化自己的媒体信息，后面放到 invite 请求的 sdp 里面。

接下来更新此次请求的状态机阶段，这样下一个状态机回调以此为起点。
然后设置 invite 请求的一些特殊头域。tmedia_session_mgr_set_qos，/*100rel */
self->supported._100rel 等。

最后调用 send_INVITE，TSIP_DIALOG_SIGNAL 产生 invite 事件通知上层用户。

send_INVITE 又调用 send_INVITEorUPDATE

内部又分如下过程。

a.tsip_dialog_request_new，创建一个通用的 sip 请求。

b.tmedia_session_mgr_get_lo 创建 invite 消息的消息体，即 sdp 信息。

c. 初始化其他头域
tsip_dialog_request_send 发送请求。 创建事务层。
tsip_transac_layer_new，tsip_transac_ict_create

tsip_transac_ict_ctor 初始化 客户端 invite 事务层：

tsip_transac_init
tsk_fsm_set_callback_terminated
tsip_transac_ict_init

最终开启客户端请求事务状态机。

tsip_transac_start

当服务器端返回 200ok 后，会调用事务层，事务层又转给会话层，最终转到

c0000_Outgoing_2_Connected_X_i2xxINVITE 回调函数，
内部对 200ok 响应做出里：

tsip_dialog_update 更新会话状态，

tsip_dialog_invite_process_ro 处理对端 sdp，建立 rtp 流。

send_ACK 给 ACK 响应。

TSIP_DIALOG_INVITE_SIGNAL 最后给上层发射 事件，通知用户接通。