PJSIP

Developer's Guide

ABOUT PJSIP

PJSIP is small-footprint and high-performance SIP stack written in C.

PJSIP is distributed under GNU General Public License (GPL). Alternative licensing is available.

Please visit http://www.pjproject.net for more details.

ABOUT THIS DOCUMENT

DOCUMENT REVISION HISTORY

Ver	Date	By	Changes
0.5.1	06 Feb 2006	bennylp	- Endpoint: • added generic capabilities management. - Module interface: • changed (removed supported methods). - Dialog: • no more stateless operations. • introducing dialog set. - Invite session is implemented.
0.5	10 Jan 2006	bennylp	Updated according to changes in module and transaction API.
0.5	19 Dec 2005	bennylp	Initial revision

Table of Contents

Table of Figures

Table of Codes

Chapter 1:General Design

1.1 Architecture

1.1.1 Communication Diagram

The following diagram shows how (SIP) messages are passed back and forth among PJSIP components.

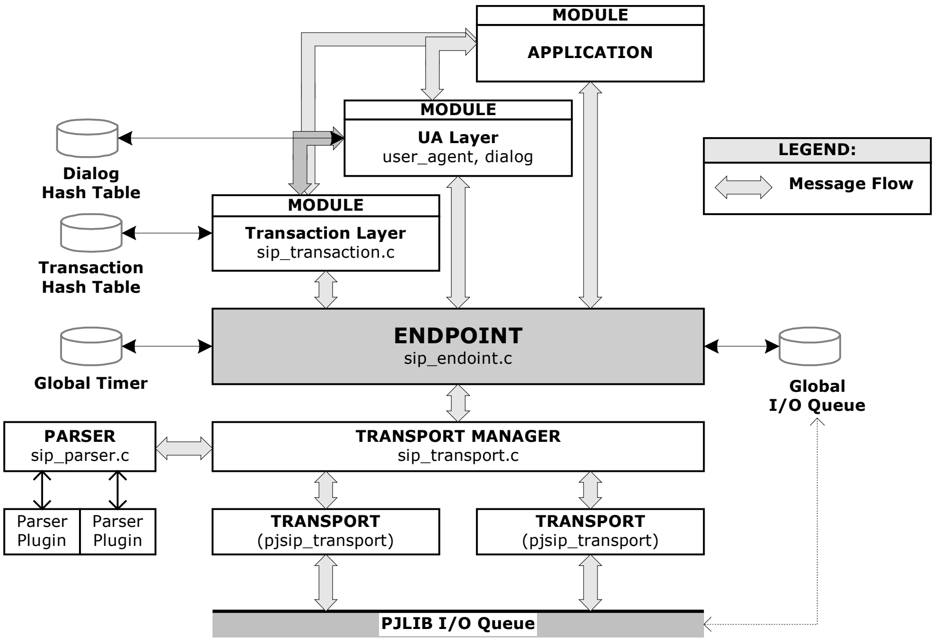

Figure 1 Collaboration Diagram

1.1.2 Class Diagram

The following diagram shows the "class diagram".

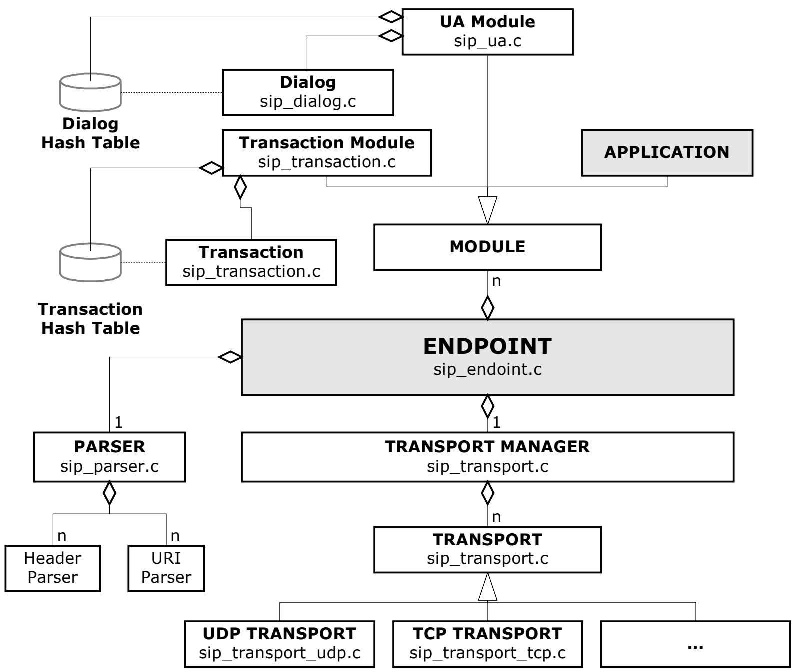

Figure 2 Class Diagram

1.2 Module

Module framework is the main means for distributing SIP messages among software components in PJSIP application. All software components in PJSIP, including the transaction layer and dialog layer, are implemented as module. Without modules, the core stack (pjsip_endpoint and transport) simply wouldn't know how to handle SIP messages.

The module framework is based on a simple but yet powerfull interface abstraction. For incoming messages, the endpoint (pjsip_endpoint) distributes the message to all modules starting from module with highest priority, until one of them says that it has processed the message. For outgoing messages, the endpoint distributes the outgoing messages before they are transmitted to the wire, to allow modules to put last modification on the message if they wish.

1.2.1 Module Declaration

Module interface is declared in <pjsip/sip_module.h> as follows.

```
struct pjsip_module
{
   pj_str_t             name;                                   // Module name.
   int                  id;                                     // Module ID, set by endpt
   int                  priority;                               // Priority
   void                *user_data;                              // User data.

   int                  method_cnt;                             // Nb.of supported methods
   const pjsip_method  *methods[8];                             // Array supported methods

   pj_status_t (*load)          (pjsip_endpoint *endpt);        // Called to load the mod.
   pj_status_t (*start)         (void);                         // Called to start.
   pj_status_t (*stop)          (void);                         // Called top stop.
   pj_status_t (*unload)        (void);                         // Called before unload
   pj_bool_t   (*on_rx_request) (pjsip_rx_data *rdata);         // Called on rx request
   pj_bool_t   (*on_rx_response)(pjsip_rx_data *rdata);         // Called on rx response
   pj_status_t (*on_tx_request) (pjsip_tx_data *tdata);         // Called on tx request
   pj_status_t (*on_tx_response)(pjsip_tx_data *tdata);         // Called on tx request
   void        (*on_tsx_state)  (pjsip_transaction *tsx,        // Called on transaction
                                  pjsip_event *event);          // state changed
};
```

Code 1 Module Declaration

All function pointers are optional; if they're not specified, they'll be treated as if they have returned successfully.

The four function pointers **load**, **start**, **stop**, and **unload** are called by endpoint to control the module state. The following diagram shows the module's state lifetime.

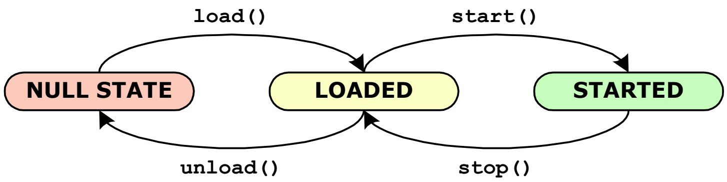

Figure 3 Module State Diagram

The **on_rx_request()** and **on_rx_response()** function pointers are the primary means for the module to receive SIP messages from endpoint (*pjsip_endpt*) or from other modules. The return value of these callbacks is important. If a callback has returned non-zero (i.e. true condition), it semantically means that the module has taken care the message; in this case, the endpoint will stop distributing the message to other modules.

The **`on_tx_request()`** and **`on_tx_response()`** function pointers are called by transport manager before a message is transmitted. This gives an opportunity for some types of modules (e.g. sigcomp, message signing) chance to make last modification to the message. All modules MUST return PJ_SUCCESS (i.e. zero status), or otherwise the transmission will be cancelled.

The **`on_tsx_state()`** function pointer is used to receive notification every time a transaction state has changed. It is different from **`on_rx_request()`** and **`on_rx_response()`** callback because it's only called when transaction state has actually changed, which means it's not called for example when transaction receives 180/Ringing response after 100/Trying response. It also means that this callback may be called for other non-message-arrival related events (e.g. message transmissions, timer timeout event, or transport error event). More information about this callback will be described in next section 1.2.5 "Transaction User and State Callback".

1.2.2 Module Priorities

Module priority specifies the order of which modules are called first to process the callback. Module with higher priority (i.e. lower priority *number*) will have their on_rx_request() and on_rx_response() called **first**, and on_tx_request() and on_tx_response() called **last**.

The values below are the standard to set module priority.

```
enum pjsip_module_priority
{
   PJSIP_MOD_PRIORITY_TRANSPORT_LAYER  = 8,  // Transport
   PJSIP_MOD_PRIORITY_TSX_LAYER        = 16, // Transaction layer.
   PJSIP_MOD_PRIORITY_UA_PROXY_LAYER   = 32, // UA or proxy layer
   PJSIP_MOD_PRIORITY_APPLICATION      = 64, // Application has lowest priority.
};
```

Code 2 Module Priorities

Note: remember that lower priority *number* means higher priority!

The priority PJSIP_MOD_PRIORITY_TRANSPORT_LAYER is the priority used by transport manager. This priority currently is only used to control message transmission, please see 1.2.4Outgoing Message Processing by Modules for more information.

PJSIP_MOD_PRIORITY_TSX_LAYER is the priority used by transaction layer module. PJSIP_MOD_PRIORITY_UA_PROXY_LAYER is the priority used by UA layer (i.e. dialog framework) or proxy layer. PJSIP_MOD_PRIORITY_APPLICATION is the suggested value for typical application modules, when they want to utilize transactions and dialogs.

1.2.3 Incoming Message Processing by Modules

When incoming message arrives, it is represented as receive message buffer (**`struct pjsip_rx_data`**, see section 4.1 "Receive Data Buffer"). Transport manager parses the message, put the parsed data structures in the receive message buffer, and passes the message to the endpoint.

The endpoint distributes the receive message buffer to each registered module by calling **`on_rx_request()`** or **`on_rx_response()`** callback, starting from module with highest priority (i.e. lowest priority *number*) until one of them returns non-zero. When one of the module has returned non-zero, endpoint stops distributing the message to the remaining of the modules, because it assumes that the module has taken care about the processing of the message.

The module which returns non-zero on the callback itself may further distribute the message to other modules. For example, the transaction module, upon receiving matching message, will process the message then distributes the message to its transaction user, which in itself must be a module too. The transaction passes the message to the transaction user (i.e. a module) by calling **`on_rx_request()`** or **`on_rx_response()`** callback of that module, after setting the transaction field in the receive message buffer so that the transaction user module can distinguish between messages that are outside transactions and messages that are inside a transaction.

The following diagram shows an example of how modules may cascadely call other modules.

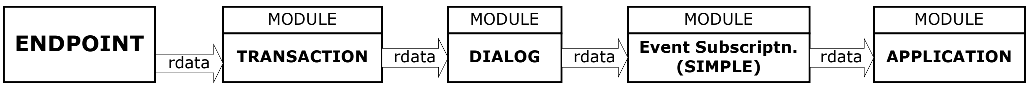

Figure 4 Cascade Module Callback

1.2.4 Outgoing Message Processing by Modules

When **`pjsip_transport_send()`** is called to send a message, transport manager calls **`on_tx_request()`** or **`on_tx_response()`** for all modules, starting with modules with lowest priority (i.e. highest priority number). When these callbacks are called, the message may have or have not been printed to contigous buffer. Modules with priority higher than PJSIP_MOD_PRIORITY_TRANSPORT_LAYER (i.e. has lower priority *number*) will receive the message **after** it has been printed to contigous buffer, while modules with lower priority receive the message **before** it has been printed to contigous buffer.

If modules want to modify the message *structure* before it is printed to buffer, then it must set its priority *number* higher than transport layer priority. If modules want to see the actual packet bytes as they are transmitted to the wire (e.g. for logging purpose), then it should set its priority *number* to lower than transport layer.

In all cases, modules MUST return PJ_SUCCESS for the return value of these callbacks. If modules return other error codes, the transmission will be cancelled and the error code is returned back to **`pjsip_transport_send()`** caller.

1.2.5 Transaction User and State Callback

A special callback in the module definition (**`on_tsx_state`**) is used to receive notification from a particular transaction when transaction state has changed. This callback is unique because transaction state may change because of non-message related events (e.g. timer timeout and transport error), and this callback is called for any changes in the transaction's state (e.g. because of transmission or receipt of messages, timeout, transport error, etc).

This callback will only be called after the module has been registered as transaction user for a particular transaction. Only one transaction user is allowed per transaction. Transaction user can be set to transaction on per transaction basis.

Normally when a transaction is created within a dialog, then the transaction user will be the UA layer on behalf of a particular dialog. But when applications work on top of the transaction layer directly, they may set themselves as the transaction user.

1.2.6 Module Specific Data

Some PJSIP components have a container where modules can put module specific data in that component. This container is named as **mod_data** by convention, and is an array of pointer to void, which is indexed by the module ID.

For example, an incoming packet buffer (**pjsip_rx_data**) has the following declaration for module specific data container:

```
struct pjsip_rx_data
{
   ...
   struct {
      void *mod_data[PJSIP_MAX_MODULE];
   } endpt_info;
};
```

Code 3 Module Specific Data

When an incoming packet buffer (**pjsip_rx_data**) is passed around to modules, a module can put module specific data in the appropriate index in **mod_data**, so that the value can be picked up later by the module or by application. For example, the transaction layer will put the matching transaction instance in the **mod_data**, and user agent layer will put the matching dialog instance in the **mod_data** too. Application can retrieve the value calling **pjsip_rdata_get_tsx()** or **pjsip_rdata_get_dlg()**, which is a simple array lookup function as follows:

```
// This code can be found in sip_transaction.c

static pjsip_module mod_tsx_layer;

pjsip_transaction *pjsip_rdata_get_tsx(pjsip_rx_data *rdata)
{
   return rdata->endpt_info.mod_data[mod_tsx_layer.id];
}
```

Code 4 Accessing Module Specific Data

1.2.7 Callback Summary

The following table summarizes the occurrence of an event and the triggering of particular callbacks. The on_tsx_state() callback will of course only be called when application has chosen to process a request statefully.

Event	**on_rx_request() or**	**on_tsx_state()**

	`on_rx_response()`	
Receipt of new requests or responses	Called	Called
Receipt retransmissions of requests or responses.	Called ONLY when priority number is lower than transaction layer[1]	Not Called
Transmission of new requests or responses.	Not Called	Called
Retransmissions of requests or responses.	Not Called	Not Called
Transaction timeout	Not Called	Called
Other transaction failure events (e.g. DNS query failure, transport failure)	Not Called	Called

Figure 5 Callback Summary

1.2.8 Sample Callback Requirements for Applications

The following table summarizes the requirements for the callbacks for each logical type of applications. Note that any of these logical applications may co-exist in a single physical/executable program, and practically it's application's decision to invoke the appropriate logical functionalities or whether to work statefull or statelessly. This decision is made on per request basis.

Application	Requirements
Stateless Proxies	Stateless proxies need to receive: 1) all incoming **requests**. 2) al incoming **responses**.
Statefull Proxies	Statefull proxies need to receive: 1) new incoming requests (i.e. that are not attached to any transactions). 2) all responses received by a transaction, and 3) any other transaction events (e.g. DNS failure, timeout, transport error).
Statefull/less Registrar Server	Registrar servers need to receive: 1) all incoming REGISTER requests.
UA	Typical UA applications need to receive: 1) incoming requests that are not attached to any transactions. 2) all requests that belong to the dialog. 3) all incoming responses associated with the dialog and any other transaction events (e.g. timeout, transport error).

Figure 6 Sample Callback Requirements

[1] This is because the matching transaction prevents the message from being distributed further (by returning PJ_TRUE) and it also does NOT call TU callback upon receiving retransmissions.

1.2.9 Sample Callback Diagrams

Incoming Message Outside Transaction and Outside Dialog

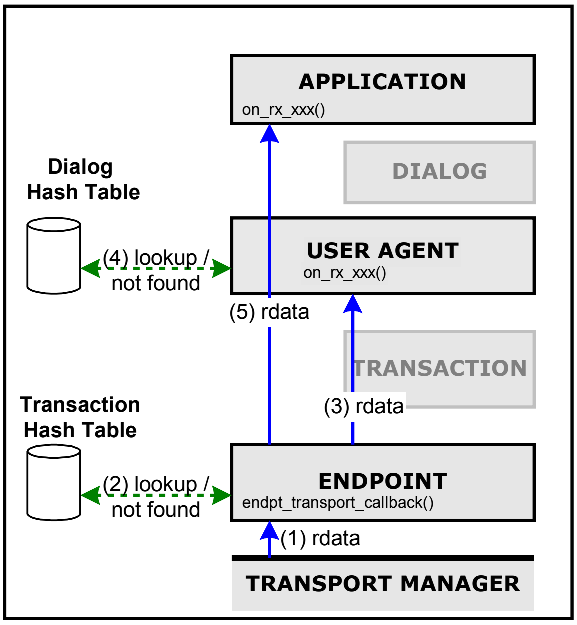

Figure 7 Processing of Incoming Message Outside Transaction/Dialog

The processing is as follows:

1) Transport manager (pjsip_tpmgr) passes all received messages to endpoint (after parsing the message).
2) Endpoint (pjsip_endpt) distributes the message to all registered callbacks. First in the callback list is transaction layer. Endpoint lookup the message in transaction table, and couldn't find a matching transaction.
3) Endpoint distributes the message to next callback in the list, which is user agent.
4) User agent lookup the message in dialog's hash table and couldn't find matching dialog.
5) Endpoint continues distributing the message to next registered callbacks until it reaches application. Application processes the message (e.g. create UAS transaction, or proxy the request, or create dialog, etc.)

Incoming Message Inside Transaction

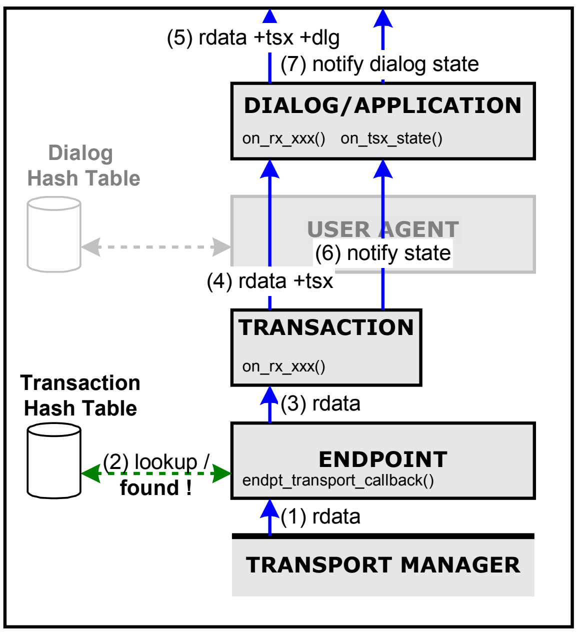

Figure 8 Processing of Incoming Message Inside Transaction

The processing is as follows:

1) Transport manager (pjsip_tpmgr) passes all received messages to endpoint (after parsing the message).
2) Endpoint (pjsip_endpt) distributes the message to all registered callbacks. First in the callback list is transaction layer. Endpoint lookup the message in transaction table, and **found** a matching transaction.
3) Endpoint distributes the message to the transaction. Because transaction's callback returns non-zero, endpoint does not distribute the message to the rest of the registered callbacks.
4) The transaction processes the response (e.g. updates the FSM). If the message is a retransmission, the processing stops here. Otherwise transaction then passes the message to it's transaction user (TU), which can be a dialog or application.
5) If the TU is a dialog, the dialog processes the response then pass the response to it's dialog user (DU, e.g. application).
6) If the arrival of the message has changed transaction's state, transaction will notify it's TU about the new state.
7) If TU is a dialog, it may further notify application about dialog's state changed.

Incoming Message Inside Dialog but Outside Transaction

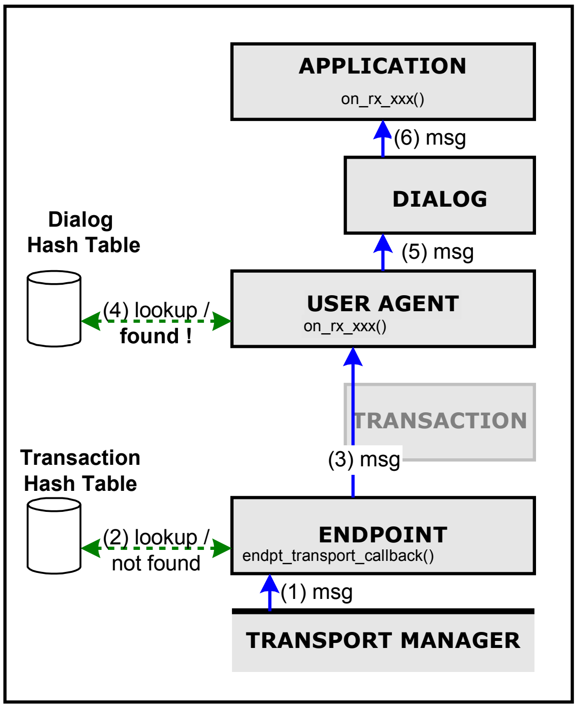

Figure 9 Processing of Incoming Message Inside Dialog but Outside Transaction

The processing is as follows:

1) Transport manager (pjsip_tpmgr) passes all received messages to endpoint (after parsing the message).
2) Endpoint (pjsip_endpt) distributes the message to all registered callbacks. First in the callback list is transaction layer. Endpoint lookup the message in transaction table, and couldn't find a matching transaction.
3) Endpoint distributes the message to next modules in the list, until it reaches user agent module.
4) The user agent module looks-up the owning of the message in dialog's hash table and found a matching dialog.
5) The user agent module passes the message to the appropriate dialog.
6) The dialog processes the message, updates it's state etc, and notify the application.

1.3 Module Management

Modules are managed by PJSIP's endpoint (`pjsip_endpoint`). Application MUST register each module manually to endpoint so that it can be recognized by the stack. Application can register or unregister module at any time during application's life-time, although it is recommended to register module only during startup and to unregister module only during application exit.

> All PJSIP modules can be registered/unregistered dynamically at anytime during application's lifetime. However by doing so, it may severely change the handling of messages. For example, when transaction module is unregistered, application may receive strayed responses that are no longer associated with any transactions.

1.3.1 Module Management API

The module management API are declared in <pjsip/sip_endpt.h>.

```
pj_status_t pjsip_endpt_register_module(    pjsip_endpoint *endpt,
                                            pjsip_module *module );
```

Register a module to the endpoint. The endpoint will then call the load and start function in the module to properly initialize the module, and assign a unique module ID for the module.

```
pj_status_t pjsip_endpt_unregister_module( pjsip_endpoint *endpt,
                                            pjsip_module *module );
```

Unregister a module from the endpoint. The endpoint will then call the stop and unload function in the module to properly shutdown the module.

Chapter 2:Message Elements

2.1 Uniform Resource Indicator (URI)

The Uniform Resource Indicator (URI) in PJSIP is modeled pretty much in object oriented manner (or some may argue it's object based, not object oriented). Because of this, URI can be treated uniformly by the stack, and new types URI can be introduced quite easily.

2.1.1 URI "Class Diagram"

The following diagram shows show the URI objects are designed.

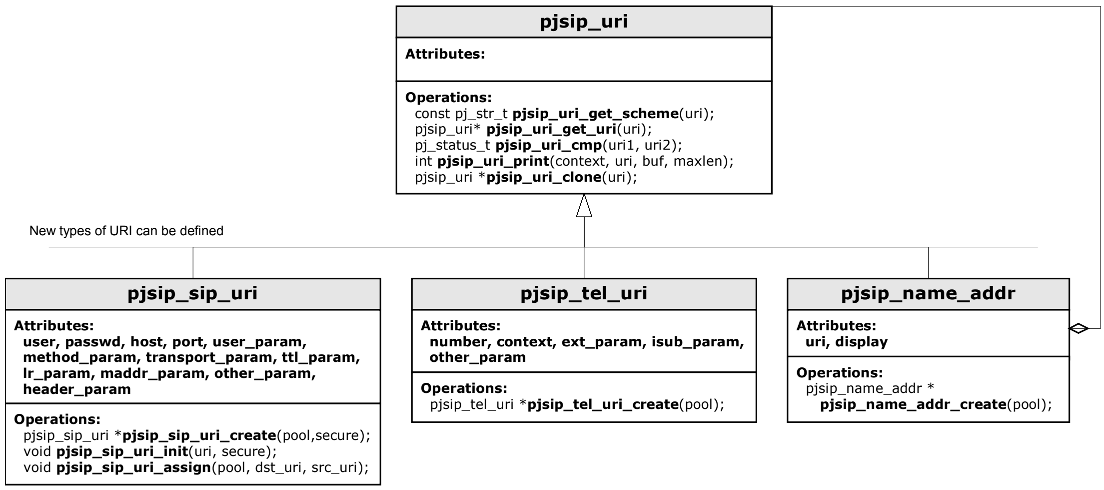

Figure 10 URI "Class Diagram"

More information on each objects will be described in next sections.

2.1.2 URI Context

URI context specifies where the URI is being used (e.g. in request line, in From/To header, etc.). The context specifies what URI elements are allowed to appear in that context. For example, transport parameter is not allowed to appear in From/To header, etc.

In PJSIP, the context must be specified when printing the URI to a buffer and when comparing two URIs. In this case, the parts of URI that is not allowed to appear in the specified context will be ignored during printing and comparison process.

```
enum pjsip_uri_context_e
{
   PJSIP_URI_IN_REQ_URI,       // The URI is in Request URI.
   PJSIP_URI_IN_FROMTO_HDR,    // The URI is in From/To header.
   PJSIP_URI_IN_CONTACT_HDR,   // The URI is in Contact header.
   PJSIP_URI_IN_ROUTING_HDR,   // The URI is in Route/Record-Route header.
   PJSIP_URI_IN_OTHER,         // Other context (web page, business card, etc.)
};
```

Code 5 URI Context

2.1.3 Base URI

The **pjsip_uri** structure contains property that is shared by all types of URI. Because of this, all types of URI can be type-casted to **pjsip_uri** and manipulated uniformly.

```
struct pjsip_uri
{
   pjsip_uri_vptr *vptr;
};
```

Code 6 Generic URI Declaration

The **pjsip_uri_vptr** specifies "virtual" function table, which members will be defined by each type of URI. Application is discouraged from calling these function pointers directly; instead it is recommended to use the URI API because they are more readable (and it saves some typings too).

```
struct pjsip_uri_vptr
{
   const pj_str_t* (*p_get_scheme)  ( const pjsip_uri *uri);
   pjsip_uri*      (*p_get_uri)     ( pjsip_uri *uri);
   int             (*p_print)       ( pjsip_uri_context_e context,
                                      const pjsip_uri *uri,
                                      char *buf, pj_size_t size);
   pj_status_t     (*p_compare)     ( pjsip_uri_context_e context,
                                      const pjsip_uri *uri1, const pjsip_uri *uri2);
   pjsip_uri *     (*p_clone)       ( pj_pool_t *pool, const pjsip_uri *uri);
};
```

Code 7 URI Virtual Function Table

The URI functions below can be applied for all types of URI objects. These functions normally are implemented as inline functions which call the corresponding function pointer in virtual function table of the URI.

```
const pj_str_t* pjsip_uri_get_scheme( const pjsip_uri *uri );
```

Get the URI scheme string (e.g. "sip", "sips", "tel", etc.).

```
pjsip_uri* pjsip_uri_get_uri( pjsip_uri *uri );
```

Get the URI object. Normally all URI objects will return itself except name address which will return the URI inside the name address object.

```
pj_status_t pjsip_uri_cmp( pjsip_uri_context_e context,
                           const pjsip_uri *uri1,
                           const pjsip_uri *uri2);
```

Compare *uri1* and *uri2* according to the specified *context*. Parameters which are not allowed to appear in the specified context will be ignored in the comparison. It will return PJ_SUCCESS is both URIs are equal.

```
int pjsip_uri_print(    pjsip_uri_context_e context,
                        const pjsip_uri *uri,
                        char *buffer,
                        pj_size_t max_size);
```

Print *uri* to the specified *buffer* according to the specified *context*. Parameters which are not allowed to appear in the specified context will not be included in the printing.

pjsip_uri* pjsip_uri_clone(pj_pool_t *pool, const pjsip_uri *uri);

Create a deep clone of *uri* using the specified pool.

2.1.4 SIP and SIPS URI

The structure **pjsip_sip_uri** represents SIP and SIPS URI scheme. It is declared in <pjsip/sip_uri.h>.

```
struct pjsip_sip_uri
{
   pjsip_uri_vptr *vptr;            // Pointer to virtual function table.
   pj_str_t        user;            // Optional user part.
   pj_str_t        passwd;          // Optional password part.
   pj_str_t        host;            // Host part, always exists.
   int             port;            // Optional port number, or zero.
   pj_str_t        user_param;      // Optional user parameter
   pj_str_t        method_param;    // Optional method parameter.
   pj_str_t        transport_param; // Optional transport parameter.
   int             ttl_param;       // Optional TTL param, or -1.
   int             lr_param;        // Optional loose routing param, or 0
   pj_str_t        maddr_param;     // Optional maddr param
   pjsip_param     other_param;     // Other parameters as list.
   pjsip_param     header_param;    // Optional header parameters as list.
};
```

Code 8 SIP URI Declaration

The following functions are specific to SIP/SIPS URI objects. In addition to these functions, application can also use the base URI functions described in previous section to manipulate SIP and SIPS URI too.

pjsip_sip_uri* pjsip_sip_uri_create(pj_pool_t *pool, pj_bool_t secure);

Create a new SIP URL using the specified *pool*. If the *secure* flag is set to non-zero, then SIPS URL will be created. This function will set **vptr** member of the URL to SIP or SIPS vptr and set all other members to blank value.

void pjsip_sip_uri_init(pjsip_sip_uri *url, pj_bool_t secure);

Initialize a SIP URL structure.

```
void pjsip_sip_uri_assign( pj_pool_t *pool,
                           pjsip_sip_uri *url,
                           const pjsip_sip_uri *rhs );
```

Perform deep copy of *rhs* to *url*.

2.1.5 Tel URI

The structure **pjsip_tel_uri** represents **tel:** URL. It is declared in <pjsip/sip_tel_uri.h>.

```
struct pjsip_tel_uri
{
   pjsip_uri_vptr *vptr;            // Pointer to virtual function table.
   pj_str_t        number;          // Global or local phone number
```

```
   pj_str_t        context;          // Phone context (for local number).
   pj_str_t        ext_param;        // Extension param.
   pj_str_t        isub_param;       // ISDN sub-address param.
   pjsip_param     other_param;      // Other parameters.
};
```

Code 9 TEL URI Declaration

The functions below are specific to TEL URI. In addition to these functions, application can also use the base URI functions described in previous section for TEL URI too.

pjsip_tel_uri* pjsip_tel_uri_create(pj_pool_t *pool);
Create a new tel: URI.

int pjsip_tel_nb_cmp(const pj_str_t *nb1, const pj_str_t *nb2);
This utility function compares two telephone numbers for equality, according to rules specified in RFC 3966 (about tel: URI). It recognizes global and local numbers, and it ignores visual separators during the comparison.

2.1.6 Name Address

A name address (**pjsip_name_addr**) does not really define a new type of URI, but rather encapsulates existing URI (e.g. SIP URI) and adds display name.

```
struct pjsip_name_addr
{
   pjsip_uri_vptr  *vptr;            // Pointer to virtual function table.
   pj_str_t         display;         // Display name.
   pjsip_uri       *uri;             // The URI.
};
```

Code 10 Name Address Declaration

The following functions are specific to name address URI object. In addition to these functions, application can also use the base URI functions described before for name address object too.

pjsip_name_addr* pjsip_name_addr_create(pj_pool_t *pool);
Create a new name address. This will set initialize the virtual function table pointer, set blank display name and set the **uri** member to NULL.

```
void pjsip_name_addr_assign(  pj_pool_t *pool,
                              pjsip_name_addr *name_addr,
                              const pjsip_name_addr *rhs );
```
Copy rhs to name_addr.

2.1.7 Sample URI Manipulation Program

```
#include <pjlib.h>
#include <pjsip_core.h>
#include <stdlib.h>           // exit()

static pj_caching_pool cp;
```

```
static void my_error_exit(const char *title, pj_status_t errcode)
{
   char errbuf[80];
   pjsip_strerror(errcode, errbuf, sizeof(errbuf));
   PJ_LOG(3,("main", "%s: %s", title, errbuf));
   exit(1);
}

static void my_init_pjlib(void)
{
   pj_status_t status;
   // Init PJLIB
   status = pj_init();
   if (status != PJ_SUCCESS) my_error_exit("pj_init() error", status);
   // Init caching pool factory.
   pj_caching_pool_init( &cp, &pj_pool_factory_default_policy, 0);
}

static void my_print_uri( const char *title, pjsip_uri *uri )
{
   char buf[80];
   int len;

   len = pjsip_uri_print( PJSIP_URI_IN_OTHER, uri, buf, sizeof(buf)-1);
   if (len < 0)
      my_error_exit("Not enough buffer to print URI", -1);

   buf[len] = '\0';
   PJ_LOG(3, ("main", "%s: %s", title, buf));
}

int main()
{
   pj_pool_t *pool;
   pjsip_name_addr *name_addr;
   pjsip_sip_uri *sip_uri;

   // Init PJLIB
   my_init_pjlib();

   // Create pool to allocate memory
   pool = pj_pool_create(&cp.factory, "mypool", 4000, 4000, NULL);
   if (!pool) my_error_exit("Unable to create pool", PJ_ENOMEM);

   // Create and initialize a SIP URI instance
   sip_uri = pjsip_sip_uri_create(pool, PJ_FALSE);
   sip_uri->user = pj_str("alice");
   sip_uri->host = pj_str("sip.example.com");

   my_print_uri("The SIP URI is", (pjsip_uri*)sip_uri);

   // Create a name address to put the SIP URI
   name_addr = pjsip_name_addr_create(pool);
   name_addr->uri = (pjsip_uri*) sip_uri;
   name_addr->display = "Alice Cooper";

   my_print_uri("The name address is", (pjsip_uri*)name_addr);
   // Done
}
```

Code 11 Sample URI Manipulation Program

2.2 SIP Methods

2.2.1 SIP Method Representation (pjsip_method)

The SIP method representation in PJSIP is also extensible; it can support new methods without needing to recompile the library.

```
struct pjsip_method
{
   pjsip_method_e id;     // Method ID, from pjsip_method_e.
   pj_str_t       name;   // Method name, which will always contain the method string.
};
```

Code 12 SIP Method Declaration

PJSIP core library declares only methods that are specified in core SIP standard (RFC 3261). For these core methods, the `id` field of `pjsip_method` will contain the appropriate value from the following enumeration:

```
enum pjsip_method_e
{
   PJSIP_INVITE_METHOD,
   PJSIP_CANCEL_METHOD,
   PJSIP_ACK_METHOD,
   PJSIP_BYE_METHOD,
   PJSIP_REGISTER_METHOD,
   PJSIP_OPTIONS_METHOD,

   PJSIP_OTHER_METHOD,
};
```

Code 13 SIP Method ID

For methods not specified in the enumeration, the `id` field of `pjsip_method` will contain `PJSIP_OTHER_METHOD` value. In this case, application must inspect the `name` field of `pjsip_method` to know the actual method.

2.2.2 SIP Method API

The following functions can be used to manipulate PJSIP's SIP method objects.

```
void pjsip_method_init(    pjsip_method *method, pj_pool_t *pool,
                           const pj_str_t *method_name );
```

Initialize method from string. This will initialize the `id` of the method field to the correct value.

```
void pjsip_method_init_np( pjsip_method *method,
                           pj_str_t *method_name );
```

Initialize method from *method_name* string without duplicating the string (np stands for no pool). The `id` field will be initialize accordingly.

```
void pjsip_method_set(  pjsip_method *method,
                        pjsip_method_id_e method_id );
```

Initialize method from the method ID enumeration. The `name` field will be initialized accordingly.

```
void pjsip_method_copy(    pj_pool_t *pool,
                           pjsip_method *method,
                           const pjsip_method *rhs );
```

Copy *rhs* to *method*.

```
int pjsip_method_cmp(   const pjsip_method *method1,
                        const pjsip_method *method2 );
```

Compare *method1* to *method2* for equality. This function returns zero if both methods are equal, and (-1) or (+1) if *method1* is less or greater than *method2* respectively.

2.3 Header Fields

All header fields in PJSIP share common header properties such as header type, name, short name, and virtual function table. Because of this, all header fields can be treated uniformly by the stack.

2.3.1 Header "Class Diagram"

The following diagram shows the snippet of PJSIP header "class diagram". There are more headers than the ones shown in the diagram; PJSIP library implements ALL headers that are specified in the core SIP specification (RFC 3261). Other headers will be implemented in the corresponding PJSIP extension module.

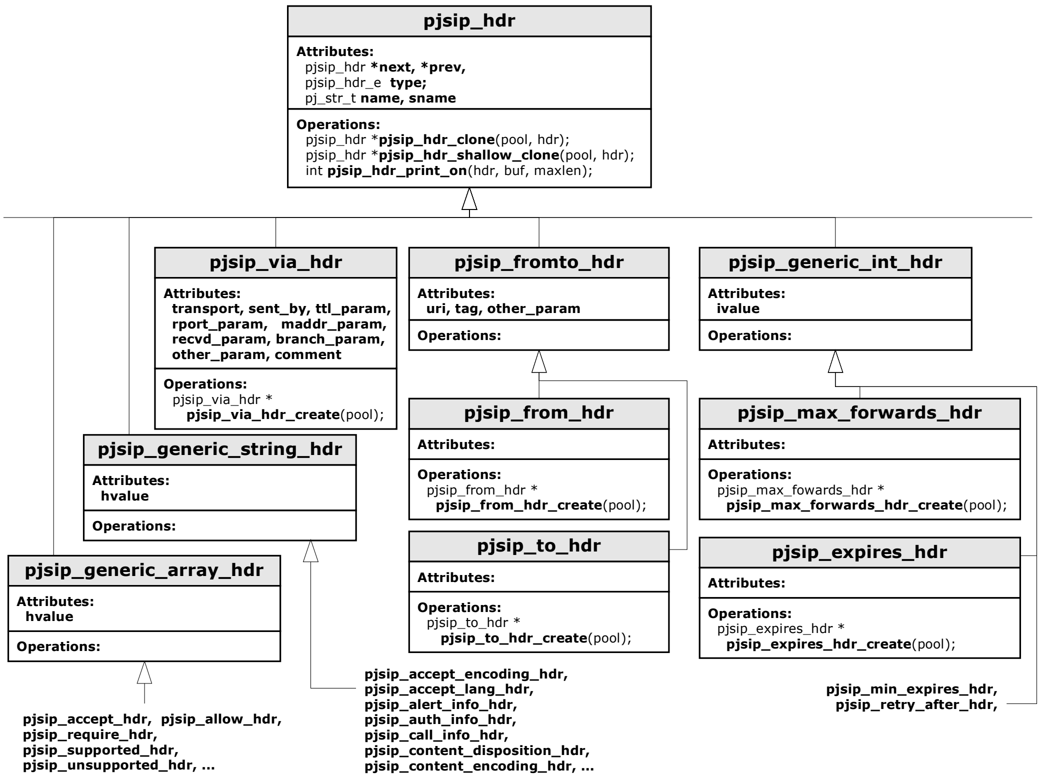

Figure 11 Header "Class Diagram"

As seen in the "class diagram", each of the specific header normaly only provide one function that is specific for that particular header, i.e. function to create the instance of the header.

2.3.2 Header Structure

To make sure that header fields contain common header properties and those properties are in the correct and same memory layout, the header declaration must call **PJSIP_DECL_HDR_MEMBER** macro as the first member field of the header, specifying the header name as argument to the macro.

```
#define PJSIP_DECL_HDR_MEMBER(hdr)              \
            /** List members. */                \
            PJ_DECL_LIST_MEMBER(hdr);           \
```

```
            /** Header type */                 \
            pjsip_hdr_e type;                  \
            /** Header name. */                \
            pj_str_t name;                     \
            /** Header short name version. */  \
            pj_str_t sname;                    \
            /** Virtual function table. */     \
            pjsip_hdr_vptr *vptr
```

Code 14 Generic Header Declaration

PJSIP defines `pjsip_hdr` structure, which contains common properties shared by all header fields. Because of this, all header fields can be typecasted to `pjsip_hdr` so that they can be manipulated uniformly.

```
struct pjsip_hdr
{
   PJSIP_DECL_HDR_MEMBER(struct pjsip_hdr);
};
```

Code 15 Generic Header Declaration

2.3.3 Common Header Functions

The `pjsip_hdr_vptr` specifies "virtual" function table, which implementation is provided by each header types. The table contains pointer to functions as follows:

```
struct pjsip_hdr_vptr
{
    pjsip_hdr *(*clone)        ( pj_pool_t *pool, const pjsip_hdr *hdr );
    pjsip_hdr *(*shallow_clone)( pj_pool_t *pool, const pjsip_hdr *hdr );
    int        (*print_on)     ( pjsip_hdr *hdr, char *buf, pj_size_t len );
};
```

Code 16 Header Virtual Function Table

Although application can freely call the function pointers in the `pjsip_hdr_vptr` directly, it is recommended that it uses the following header APIs instead, because they will make the program more readable.

```
pjsip_hdr *pjsip_hdr_clone(   pj_pool_t *pool,
                              const pjsip_hdr *hdr );
```

Perform deep clone of *hdr* header.

```
pjsip_hdr *pjsip_hdr_shallow_clone(  pj_pool_t *pool,
                                     const pjsip_hdr *hdr );
```

Perform shallow clone of *hdr* header. A shallow cloning creates a new exact copy of the specified header field, however most of its value will still point to the values in the original header. Normally shallow clone is just a simple `memcpy()` from the original header to a new header, therefore it's expected that this operation is faster than deep cloning.

However, care must be taken when shallow cloning headers. It must be understood that the new header still shares common pointers to the values in the old header. Therefore, when the pool containing the original header is destroyed, the new header will be rendered invalid too although the new header was shallow-cloned using different memory pool. Or if some values in the original header was modified, then the corresponding values in the shallow-cloned header will be modified too.

Despite of this, shallow cloning is used widely in the library. For example, a dialog has some headers which values are more or less persistent during the session (e.g. From, To, Call-Id, Route, and Contact). When creating a request, the dialog can just shallow-clone these headers (instead of performing full cloning) and put them in the request message.

```
int pjsip_hdr_print_on( pjsip_hdr *hdr,
                        char *buf,
                        pj_size_t max_size);
```

Print the specified header to a buffer (e.g. before transmission). This function returns the number of bytes printed to the buffer, or –1 when the buffer is overflow.

2.3.4 Supported Header Fields

The "standard" PJSIP header fields are declared in <pjsip/sip_msg.h>. Other header fields may be declared in header files that implement specific functionalities or SIP extensions (e.g. headers used by SIMPLE extension, etc.).

Each header field normally only defines one specific API for manipulating them, i.e. the function to create that specific header field. Other APIs are exported through the virtual function table.

The APIs to create individual header fields are by convention named after the header field name and followed by `_create()` suffix. For example, call function `pjsip_via_hdr_create()` to create an instance of `pjsip_via_hdr` header.

Please refer to <pjsip/sip_msg.h> for complete list of header fields defined by PJSIP core.

2.3.5 Header Array Elements

A lot of SIP headers (e.g. Require, Contact, Via, etc.) can be grouped together as a single header field and separated by comma. Example:

```
Contact: <sip:alice@sip.example.com>;q=1.0, <tel:+442081234567>;q=0.5
Via: SIP/2.0/UDP proxy1.example.com;branch=z9hG4bK87asdks7, SIP/2.0/UDP
     proxy2.example.com;branch=z9hG4bK77asjd
```

NOTE: PJSIP does not support representing array elements in a header for complex header types (e.g. Contact, Via, Route, Record-Route). Simple string array however is supported (e.g. Require, Supported, etc.).

When the parser encounters such arrays in headers, it will split the array into individual headers while maintaining their order of appearance. So for the example above, the parser will modify the message to:

```
Contact: <sip:alice@sip.example.com>;q=1.0
Contact: <tel:+442081234567>;q=0.5
Via: SIP/2.0/UDP proxy1.example.com;branch=z9hG4bK87asdks7
Via: SIP/2.0/UDP proxy2.example.com;branch=z9hG4bK77asjd
```

The SIP standard specifies that there should NOT be any difference in the processing of message containing either kind of header representations. So we believe that the removal of header array support will not limit the functionality of PJSIP at all.

The reason why we impose this limitation is because based on our experience, the removal of header array support greatly simplifies processing of headers. If header array were supported, then application not only must inspect all headers, it also has to inspect some headers to see if they contain arrays. With the removal of array support, application only has to inspect the main header list in the message.

2.4 Message Body (pjsip_msg_body)

SIP message body is represented with **pjsip_msg_body** structure in PJSIP. This structure is declared in <pjsip/sip_msg.h>.

```
struct pjsip_msg_body
{
    /** MIME content type.
     *  For incoming messages, the parser will fill in this member with the
     *  content type found in Content-Type header.
     *
     *  For outgoing messages, application must fill in this member with
     *  appropriate value, because the stack will generate Content-Type header
     *  based on the value specified here.
     */
    pjsip_media_type content_type;

    /** Pointer to buffer which holds the message body data.
     *  For incoming messages, the parser will fill in this member with the
     *  pointer to the body string.
     *
     *  When sending outgoing message, this member doesn't need to point to the
     *  actual message body string. It can be assigned with arbitrary pointer,
     *  because the value will only need to be understood by the print_body()
     *  function. The stack itself will not try to interpret this value, but
     *  instead will always call the print_body() whenever it needs to get the
     *  actual body string.
     */
    void *data;

    /** The length of the data.
     *  For incoming messages, the parser will fill in this member with the
     *  actual length of message body.
     *
     *  When sending outgoing message, again just like the "data" member, the
     *  "len" member doesn't need to point to the actual length of the body
     *  string.
     */
    unsigned len;

    /** Pointer to function to print this message body.
     *  Application must set a proper function here when sending outgoing
     *  message.
     *
     *  @param msg_body       This structure itself.
     *  @param buf            The buffer.
     *  @param size           The buffer size.
     *
     *  @return               The length of the string printed, or -1 if there is
     *                        not enough space in the buffer to print the whole
     *                        message body.
     */
    int (*print_body)         ( struct pjsip_msg_body *msg_body,
                                char *buf, pj_size_t size );

    /** Pointer to function to clone the data in this message body.
     */
    void* (*clone_data)       ( pj_pool_t *pool, const void *data, unsigned len );
};
```

Code 17 Message Body Declaration

The following are APIs that are provided for manipulating SIP message objects.

```
pj_status_t pjsip_msg_body_clone(pj_pool_t *pool,
                                 pjsip_msg_body *dst_body,
                                 const pjsip_msg_body *src_body);
```

Clone the message body in *src_body* to the *dst_body*. This will duplicate the contents of the message body using the *clone_data* member of the source message body.

2.5 Message (pjsip_msg)

Both request and response message in PJSIP are represented with **pjsip_msg** structure in <pjsip/sip_msg.h>. The following code snippet shows the declaration of **pjsip_mg** along with other supporting structures.

```
enum pjsip_msg_type_e
{
   PJSIP_REQUEST_MSG,          // Indicates request message.
   PJSIP_RESPONSE_MSG,         // Indicates response message.
};

struct pjsip_request_line
{
   pjsip_method  method;       // Method for this request line.
   pjsip_uri    *uri;          // URI for this request line.
};

struct pjsip_status_line
{
   int           code;         // Status code.
   pj_str_t      reason;       // Reason string.
};

struct pjsip_msg
{
   /** Message type (ie request or response). */
   pjsip_msg_type_e  type;

   /** The first line of the message can be either request line for request
    *  messages, or status line for response messages. It is represented here
    *  as a union.
    */
   union
   {
      /** Request Line. */
      struct pjsip_request_line   req;

      /** Status Line. */
      struct pjsip_status_line    status;

   } line;

   /** List of message headers. */
   pjsip_hdr hdr;

   /** Pointer to message body, or NULL if no message body is attached to
    *  this message.
    */
   pjsip_msg_body *body;
};
```

Code 18 SIP Message Declaration

The following are APIs that are provided for manipulating SIP message objects.

```
pjsip_msg* pjsip_msg_create(  pj_pool_t *pool,
```

```
                                pjsip_msg_type_e type);
```

Create a request or response message according to the *type*.

```
pjsip_hdr* pjsip_msg_find_hdr(   pjsip_msg *msg,
                                 pjsip_hdr_e hdr_type,
                                 pjsip_hdr *start);
```

Find header in the *msg* which has the specified *type*, searching from (and including) the specified *start* position in the header list. If *start* is NULL, then the function searches from the first header in the message. Returns NULL when no more header at and after the specified position can be found.

```
pjsip_hdr* pjsip_msg_find_hdr_by_name(  pjsip_msg *msg,
                                        const pj_str_t *name,
                                        pjsip_hdr *start);
```

Find header in the *msg* which has the specified *name*, searching both long and short name version of the header from the specified *start* position in the header list. If *start* is NULL, then the function searches from the first header in the message. Returns NULL when no more headers at and after the specified position can be found.

```
void pjsip_msg_add_hdr(    pjsip_msg *msg,
                           pjsip_hdr *hdr);
```

Add *hdr* as the last header in the *msg*.

```
void pjsip_msg_insert_first_hdr(  pjsip_msg *msg,
                                  pjsip_hdr *hdr);
```

Add *hdr* as the first header in the *msg*.

```
pj_ssize_t pjsip_msg_print(   pjsip_msg *msg,
                              char *buf,
                              pj_size_t size );
```

Print the whole contents of *msg* to the specified buffer. The function returns the number of bytes written, or –1 if buffer is overflow.

2.6 SIP Status Codes

SIP status codes that are defined by the core SIP specification (RFC 3261) is represented by **pjsip_status_code** enumeration in <pjsip/sip_msg.h>. In addition, the default reason text can be obtained by calling **pjsip_get_status_text()** function.

The following snippet shows the declaration of the status code in PJSIP.

```
enum pjsip_status_code
{
    PJSIP_SC_TRYING = 100,
    PJSIP_SC_RINGING = 180,
    PJSIP_SC_CALL_BEING_FORWARDED = 181,
    PJSIP_SC_QUEUED = 182,
    PJSIP_SC_PROGRESS = 183,

    PJSIP_SC_OK = 200,

    PJSIP_SC_MULTIPLE_CHOICES = 300,
    PJSIP_SC_MOVED_PERMANENTLY = 301,
    PJSIP_SC_MOVED_TEMPORARILY = 302,
    PJSIP_SC_USE_PROXY = 305,
    PJSIP_SC_ALTERNATIVE_SERVICE = 380,

    PJSIP_SC_BAD_REQUEST = 400,
    PJSIP_SC_UNAUTHORIZED = 401,
    PJSIP_SC_PAYMENT_REQUIRED = 402,
    PJSIP_SC_FORBIDDEN = 403,
    PJSIP_SC_NOT_FOUND = 404,
    PJSIP_SC_METHOD_NOT_ALLOWED = 405,
```

```
    PJSIP_SC_NOT_ACCEPTABLE = 406,
    PJSIP_SC_PROXY_AUTHENTICATION_REQUIRED = 407,
    PJSIP_SC_REQUEST_TIMEOUT = 408,
    PJSIP_SC_GONE = 410,
    PJSIP_SC_REQUEST_ENTITY_TOO_LARGE = 413,
    PJSIP_SC_REQUEST_URI_TOO_LONG = 414,
    PJSIP_SC_UNSUPPORTED_MEDIA_TYPE = 415,
    PJSIP_SC_UNSUPPORTED_URI_SCHEME = 416,
    PJSIP_SC_BAD_EXTENSION = 420,
    PJSIP_SC_EXTENSION_REQUIRED = 421,
    PJSIP_SC_INTERVAL_TOO_BRIEF = 423,
    PJSIP_SC_TEMPORARILY_UNAVAILABLE = 480,
    PJSIP_SC_CALL_TSX_DOES_NOT_EXIST = 481,
    PJSIP_SC_LOOP_DETECTED = 482,
    PJSIP_SC_TOO_MANY_HOPS = 483,
    PJSIP_SC_ADDRESS_INCOMPLETE = 484,
    PJSIP_AC_AMBIGUOUS = 485,
    PJSIP_SC_BUSY_HERE = 486,
    PJSIP_SC_REQUEST_TERMINATED = 487,
    PJSIP_SC_NOT_ACCEPTABLE_HERE = 488,
    PJSIP_SC_REQUEST_PENDING = 491,
    PJSIP_SC_UNDECIPHERABLE = 493,

    PJSIP_SC_INTERNAL_SERVER_ERROR = 500,
    PJSIP_SC_NOT_IMPLEMENTED = 501,
    PJSIP_SC_BAD_GATEWAY = 502,
    PJSIP_SC_SERVICE_UNAVAILABLE = 503,
    PJSIP_SC_SERVER_TIMEOUT = 504,
    PJSIP_SC_VERSION_NOT_SUPPORTED = 505,
    PJSIP_SC_MESSAGE_TOO_LARGE = 513,

    PJSIP_SC_BUSY_EVERYWHERE = 600,
    PJSIP_SC_DECLINE = 603,
    PJSIP_SC_DOES_NOT_EXIST_ANYWHERE = 604,
    PJSIP_SC_NOT_ACCEPTABLE_ANYWHERE = 606,

    PJSIP_SC_TSX_TIMEOUT = 701,
    PJSIP_SC_TSX_RESOLVE_ERROR = 702,
    PJSIP_SC_TSX_TRANSPORT_ERROR = 703,
};

/**
 * Get the default status text for the status code.
 * @param status_code         SIP Status Code
 * @return                    textual message for the status code.
 */
PJ_DECL(const pj_str_t*) pjsip_get_status_text(int status_code);
```

Code 19 SIP Status Code Constants

PJSIP also defines new status class (i.e. 7xx) for additional error status during message processing (e.g. transport error, DNS error, etc). This class however is only used internally; it will not go out on the wire.

2.7 Non-Standard Parameter Elements

In PJSIP, known or "standard" parameters (e.g. URI parameters, header field parameters) will normally be represented as individual attributes/fields of the corresponding structure. Parameters that are not "standard" will be put in a list of parameters, with each parameter is represented as **`pjsip_param`** structure. Non-standard parameter normally is declared as **`other_param`** field in the owning structure.

2.7.1 Data Structure Representation (pjsip_param)

This structure describes each individual parameter in a list.

```
struct pjsip_param
{
   PJ_DECL_LIST_MEMBER(struct pjsip_param);  // Generic list member.
   pj_str_t     name;                        // Param/header name.
   pj_str_t     value;                       // Param/header value.
};
```

Code 20 Non-Standard Parameter Declaration

For example of its usage, please see **`other_param`** and **`header_param`** fields in the declaration of **`pjsip_sip_uri`** (see previous section 2.1.4 "SIP and SIPS URI") or **`other_param`** field in the declaration of **`pjsip_tel_uri`** (see previous section 2.1.5 "Tel URI").

2.7.2 Non-Standard Parameter Manipulation

Some functions are provided to assist manipulation of non-standard parameters in parameter list.

```
pjsip_param* pjsip_param_find(   const pjsip_param *param_list,
                                 const pj_str_t *name );
```

This function will perform case-insensitive search for the specified parameter name.

```
void pjsip_param_clone(pj_pool_t *pool,
                       pjsip_param *dst_list,
                       const pjsip_param *src_list);
```

Perform full/deep clone of parameter list.

```
void pjsip_param_shallow_clone(  pj_pool_t *pool,
                                 pjsip_param *dst_list,
                                 const pjsip_param *src_list);
```

Perform shallow clone of parameter list.

```
pj_ssize_t pjsip_param_print_on( const pjsip_param *param_list,
                                 char *buf,
                                 pj_size_t max_size,
                                 const pj_cis_t *pname_unres,
                                 const pj_cis_t *pvalue_unres,
                                 int sep);
```

Print the parameter list to the specified buffer. The *pname_unres* and *pvalue_unres* is the specification of which characters are allowed to appear unescaped in **`pname`** and **`pvalue`** respectively; any characters outside these specifications will be escaped by the function. The argument *sep* specifies separator character to be used between parameters (normally it is semicolon (';') character for normal parameter or comma (',') when the parameter list is a header parameter).

2.8 Escapement Rules

PJSIP provides automatic un-escapement during parsing and escapement during printing ONLY for the following message elements:

- all types of URI and their elements are automatically escaped and un-escaped according to their individual escapement rule.
- parameters appearing in all message elements (e.g. in URL, in header fields, etc.) are automatically escaped and un-escaped.

Other message elements will be passed un-interpreted by the stack.

Chapter 3:Parser

3.1 Features

Some features of the PJSIP parser:

- It's a top-down, handwritten parser. It uses PJLIB's scanner, which is pretty fast and reduces the complexity of the parser, which make the parser readable.
- As said above, it's pretty fast. On a single P4/2.6GHz machine, it's able to parse more than 68K of typical 800 bytes SIP message or 860K of 80 bytes URLs in one second. Note that your mileage may vary, and different PJSIP versions may have different performance.
- It's reentrant, which will make it scalable on machine with multi-processors.
- It's extensible. Modules can plug-in new types of header or URI to the parser.

The parser features almost a lot of tricks thinkable to achieve the highest performance, such as:

- it uses zero-copy for all message elements; i.e., when an element, e.g. a *pvalue*, is parsed, the parser does not copy the *pvalue* contents to the appropriate field in the message; instead it will just put the pointer and length to the appropriate field in the message. This is only possible because PJSIP uses *pj_str_t* all the way throughout the library, which does not require strings to be NULL terminated.
- it uses PJLIB's memory pool (*pj_pool_t*) for memory allocation for the message structures, which provides multiple times speed-up over traditional *malloc()* function.
- it uses zero synchronization. The parser is completely reentrant so that no synchronization function is required.
- it uses PJLIB's try/catch exception framework, which not only greatly simplifies the parser and make it readable, but also saves tedious error checking in the parsers. With an exception framework, only one exception handler needs to be installed at the top-most function of the parser.

One feature that PJSIP parser doesn't implement is *lazy parsing*, which a lot of people probably brag about its usability. In early stage of the design, we decided **not** to implement lazy parsing, because of the following reasons:

- it complicates things, especially error handling. With lazy parsing, basically all parts of the program must be prepared to handle error condition when parsing failed at later stage when application needs to access a particular message element.
- at the end of the day, we believe that PJSIP parser is very fast anyway that it doesn't need lazy parsing. Although having said that, there will be some switches that can be turned-on in PJSIP parser to ignore parsing of some headers for some type of applications (e.g. proxies, which only needs to inspect few header types).

3.2 Functions

The main PJSIP parser is declared in <pjsip/sip_parser.h> and defined in <pjsip/sip_parser.c>. Other parts of the library may provide other parsing functionalities and extend the parser (e.g. <pjsip/sip_tel_uri.c> provides function to parse TEL URI and registers this function to the main parser).

3.2.1 Message Parsing

```
pj_status_t pjsip_find_msg(   const char *buf,
                              pj_size_t size,
                              pj_bool_t is_datagram,
                              pj_size_t *msg_size);
```

Checks that an incoming packet in *buf* contains a valid SIP message. When a valid SIP message is detected, the size of the message will be indicated in *msg_size*. If *is_datagram* is specified, this function will always return PJ_SUCCESS.

Note that the function expects the buffer in *buf* to be NULL terminated.

```
pjsip_msg* pjsip_parse_msg(   pj_pool_t *pool,
                              char *buf, pj_size_t size,
                              pjsip_parser_err_report *err_list);
```

Parse a buffer in *buf* into SIP message. The parser will return the message if at least SIP request/status line has been successfully parsed. Any error encountered during parsing will be reported in *err_list* if this parameter is not NULL.

Note that the function expects the buffer in *buf* to be NULL terminated.

```
pjsip_msg* pjsip_parse_rdata( char *buf, pj_size_t size,
                              pjsip_rx_data *rdata );
```

Parse a buffer in *buf* into SIP message. The parser will return the message if at least SIP request/status line has been successfully parsed. In addition, this function updates various pointer to headers in *msg_info* portion of the *rdata*.

Note that the function expects the buffer in *buf* to be NULL terminated.

3.2.2 URI Parsing

```
pjsip_uri* pjsip_parse_uri(   pj_pool_t *pool,
                              char *buf, pj_size_t size,
                              unsigned option);
```

Parse a buffer in *buf* into SIP URI. If PJSIP_PARSE_URI_AS_NAMEADDR is specified in the *option*, the function will always "wrap" the URI as name address. If PJSIP_PARSE_URI_IN_FROM_TO_HDR is specified in the *option*, the function will not parse the parameters after the URI if the URI is not enclosed in brackets (because they will be treated as header parameters, not URI parameters).

This function is able to parse any types of URI that are recognized by the library, and return the correct instance of the URI depending on the scheme.

Note that the function expects the buffer in *buf* to be NULL terminated.

3.2.3 Header Parsing

```
void* pjsip_parse_hdr( pj_pool_t *pool, const pj_str_t *hname,
                       char *line, pj_size_t size,
                       int *parsed_len);
```

Parse the content of a header in *line* (i.e. part of header after the colon character) according to the header type *hname*. It returns the appropriate instance of the header.

Note that the function expects the buffer in *buf* to be NULL terminated.

```
pj_status_t pjsip_parse_headers( pj_pool_t *pool,
                                 char *input, pj_size_t size,
                                 pj_list *hdr_list );
```

Parse multiple headers found in *input* buffer and put the results in *hdr_list*. The function expects the header to be separated either by a newline (as in SIP message) or ampersand character (as in URI). The separator is optional for the last header.

Note that the function expects the buffer in *buf* to be NULL terminated.

3.3 Extending Parser

The parser can be extended by registering function pointers to parse new types of headers or new types of URI.

```
typedef pjsip_hdr* (pjsip_parse_hdr_func)(pjsip_parse_ctx *context);
pj_status_t pjsip_register_hdr_parser(  const char *hname,
                                        const char *hshortname,
                                        pjsip_parse_hdr_func *fptr);
```

Register new function to parse new type of SIP message header.

```
typedef void* (pjsip_parse_uri_func)(pj_scanner *scanner, pj_pool_t *pool,
                                     pj_bool_t parse_params);
pj_status_t pjsip_register_uri_parser(  char *scheme,
                                        pjsip_parse_uri_func *func);
```

Register new function to parse new type of SIP URI scheme.

Chapter 4:Message Buffers

4.1 Receive Data Buffer

A SIP message received by PJSIP will be passed around to different PJSIP software components as **pjsip_rx_data** instead of a plain message. This structure contains all information describing the received message.

Receive and transmit data buffers are declared in <pjsip/sip_transport.h>.

4.1.1 Receive Data Buffer Structure

```
struct pjsip_rx_data
{
   // This part contains static info about the buffer.
   struct
   {
      pj_pool_t               *pool;         // Pool owned by this buffer
      pjsip_transport         *transport;    // The transport that received the msg.
      pjsip_rx_data_op_key     op_key;       // Ioqueue's operation key
   } tp_info;

   // This part contains information about the packet
   struct
   {
      pj_time_val              timestamp;                 // Packet arrival time
      char                     packet[PJSIP_MAX_PKT_LEN]; // The packet buffer
      pj_uint32_t              zero;                      // Zero padding.
      int                      len;                       // Packet length
      pj_sockaddr              addr;                      // Source address
      int                      addr_len;                  // Address length.
   } pkt_info;

   // This part describes the message and message elements after parsing.
   struct
   {
      char                    *msg_buf;      // Pointer to start of msg in the buf.
      int                      len;          // Message length.
      pjsip_msg               *msg;          // The parsed message.

      // Shortcut to important headers:

      pj_str_t                 call_id;      // Call-ID string.
      pjsip_from_hdr          *from;         // From header.
      pjsip_to_hdr            *to;           // To header.
      pjsip_via_hdr           *via;          // First Via header.
      pjsip_cseq_hdr          *cseq;         // CSeq header.
      pjsip_max_forwards_hdr  *max_fwd;      // Max-Forwards header.
      pjsip_route_hdr         *route;        // First Route header.
      pjsip_rr_hdr            *record_route; // First Record-Route header.
      pjsip_ctype_hdr         *ctype;        // Content-Type header.
      pjsip_clen_hdr          *clen;         // Content-Length header.
      pjsip_require_hdr       *require;      // The first Require header.

      pjsip_parser_err_report  parse_err;    // List of parser errors.
   } msg_info;

   // This part is updated after the rx_data reaches endpoint.
   struct
   {
      pj_str_t                 key;                           // Transaction key.
      void                    *mod_data[PJSIP_MAX_MODULE];    // Module specific data.
   } endpt_info;
};
```

Code 21 Receive Data Buffer Declaration

4.2 Transmit Data Buffer (pjsip_tx_data)

When PJSIP application wants to send outgoing message, it must create a transmit data buffer. The transmit data buffer provides memory pool from which all message fields pertaining for the message must be allocated from, a reference counter, lock protection, and other information that are needed by the transport layer to process the message.

```
struct pjsip_tx_data
{
    /** This is for transmission queue; it's managed by transports. */
    PJ_DECL_LIST_MEMBER(struct pjsip_tx_data);

    /** Memory pool for this buffer. */
    pj_pool_t                *pool;

    /** A name to identify this buffer. */
    char                      obj_name[PJ_MAX_OBJ_NAME];

    /** Time of the rx request; set by pjsip_endpt_create_response(). */
    pj_time_val               rx_timestamp;

    /** The transport manager for this buffer. */
    pjsip_tpmgr              *mgr;

    /** Ioqueue asynchronous operation key. */
    pjsip_tx_data_op_key      op_key;

    /** Lock object. */
    pj_lock_t                *lock;

    /** The message in this buffer. */
    pjsip_msg                *msg;

    /** Contigous buffer containing the packet. */
    pjsip_buffer              buf;

    /** Reference counter. */
    pj_atomic_t              *ref_cnt;

    /** Being processed by transport? */
    int                       is_pending;

    /** Transport manager internal. */
    void                     *token;
    void                     (*cb)(void*, pjsip_tx_data*, pj_ssize_t);

    /** Transport info, only valid during on_tx_request() and on_tx_response() */
    struct {
       pjsip_transport       *transport;     /**< Transport being used.   */
       pj_sockaddr            dst_addr;      /**< Destination address.    */
       int                    dst_addr_len;  /**< Length of address.      */
       char                   dst_name[16];  /**< Destination address.    */
       int                    dst_port;      /**< Destination port.       */
    } tp_info;
};
```

Code 22 Transmit Data Buffer Declaration

Chapter 5:Transport Layer

Transports are used to send/receive messages across the network. PJSIP transport framework is extensible, which means application can register its own means to transport messages.

5.1 Transport Layer Design

5.1.1 "Class Diagram"

The following diagram shows the relationship between instances in the transport layer.

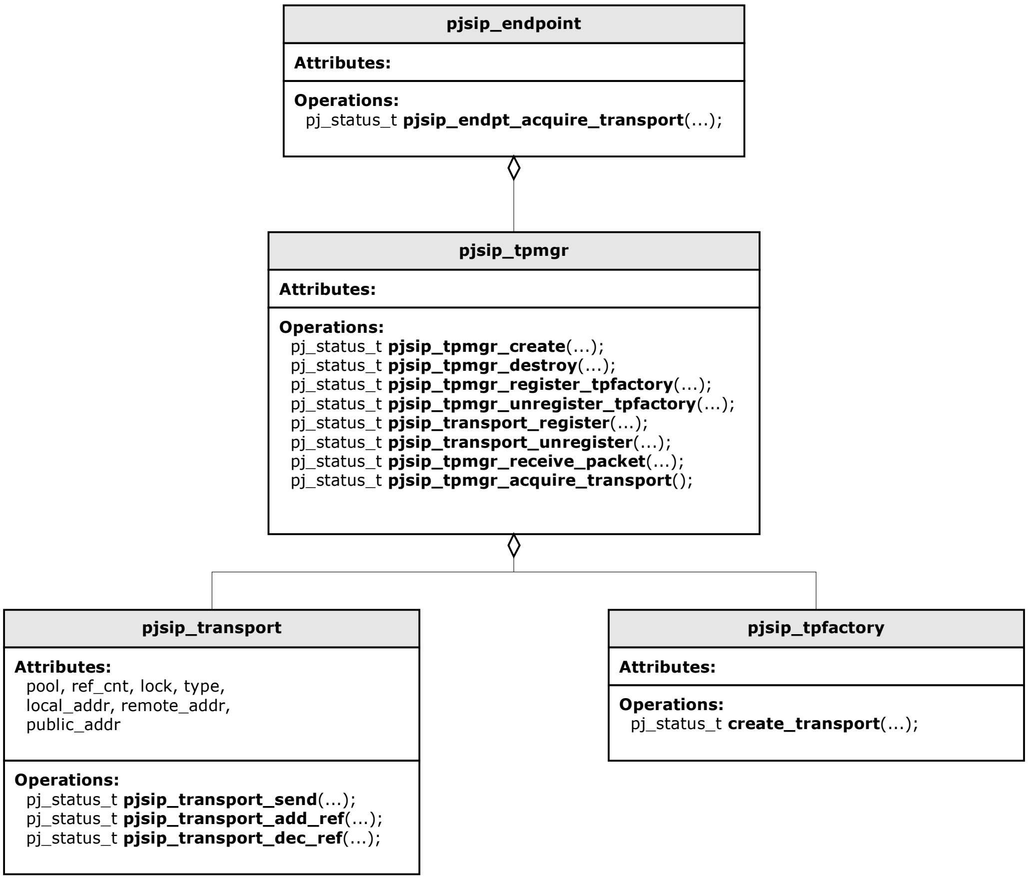

Figure 12 Transport Layer "Class Diagram"

5.1.2 Transport Manager

The transport manager (`pjsip_tpmgr`) manages all transport objects and factories. It provides the following functionalities:

- Manages transports life-time by using transport's reference counter and idle timer.
- Manages transport factories.
- Receives packet from transport, parse the packet, and deliver the SIP message to endpoint.

- Find matching transport to send SIP message to particular destination based on the transport type and remote address.
- Create new transports dynamically when no existing transport is available to send SIP message to a new destination.

There is only one transport manager per endpoint. Transport manager is normally not visible to applications; applications should use the functions provided by endpoint.

5.1.3 Transport Factory

The transport factory (**`pjsip_tpfactory`**) is used to create dynamic connection to remote endpoint. An example of this type of connection is TCP transport, where one TCP transport needs to be created for each destination.

When transport manager detects that it need to create new transport to the new destination, it finds the transport factory with matching specification (i.e. transport type) and ask the factory to create the connection.

A transport factory object is declared as follows.

5.1.4 Transport

Transport object is represented with **`pjsip_transport`** structure. Each instance of this structure normally represents one socket handle (e.g. UDP, TCP), although the transport layer supports non-socket transport as well.

General Transport Operations

From the framework's point of view, transport object is an active object. The framework doesn't have mechanism to poll the transport objects; instead, the transport objects must find their own way to receive packets from network and deliver the packets to *transport manager* for further processing.

The recommended way to achieve this is to register the transport's socket handle to endpoint's I/O queue (**`pj_ioqueue_t`**), so that when the endpoint polls the I/O queue, packets from the network will be received by the transport object.

Once a packet has been received by the transport object, it must deliver the packet to transport manager by calling **`pjsip_tpmgr_receive_packet()`** function, so that it can be parsed and distributed to the rest of the stack. The transport object must initialize both **`tp_info`** and **`pkt_info`** member of receive data buffer (**`pjsip_rx_data`**).

Each transport object has a pointer to function to send messages to the network (i.e. **`send_msg()`** attribute of the transport object). Application (or the stack) sends messages to the network by calling **`pjsip_transport_send()`** function, which eventually will reach the transport object, and **`send_msg()`** will be called. The sending of packet may complete asynchronously; if so, transport must return PJ_EPENDING status in send_msg() and call the callback that is specified in argument when the message has been sent to destination.

Transport Object Declaration

The following code shows the declaration of a transport object.

```
struct pjsip_transport
{
    char                        obj_name[PJ_MAX_OBJ_NAME];   // Name.

    pj_pool_t                  *pool;         // Pool used by transport.
    pj_atomic_t                *ref_cnt;      // Reference counter.
    pj_lock_t                  *lock;         // Lock object.
    int                         tracing;      // Tracing enabled?

    pjsip_transport_type_e      type;         // Transport type.
    char                        type_name[8]; // Type name.
    unsigned                    flag;         // See #pjsip_transport_flags_e

    pj_sockaddr                 local_addr;   // Bound address.
    pjsip_host_port             addr_name;    // Published name (e.g. STUN address).
    pj_sockaddr                 rem_addr;     // Remote addr (zero for UDP)

    pjsip_endpoint             *endpt;        // Endpoint instance.
    pjsip_tpmgr                *tpmgr;        // Transport manager.
    pj_timer_entry              idle_timer;   // Timer when ref cnt is zero.

    /* Function to be called by transport manager to send SIP messages. */
    pj_status_t                (*send_msg)( pjsip_transport *transport,
                                            pjsip_tx_data *tdata,
                                            const pj_sockaddr_in *rem_addr,
                                            void *token,
                                            void (*callback)(  pjsip_transport*,
                                                               void *token,
                                                               pj_ssize_t sent));

    /* Called to destroy this transport. */
    pj_status_t                (*destroy)(  pjsip_transport *transport );

    /* Application may extend this structure. */
};
```

Code 23 Transport Object Declaration

Transport Management

Transports are registered to transport manager by **`pjsip_transport_register()`**. Before this function is called, all members of the transport structure must be initialized.

Transport's life-time is managed automatically by transport manager. Each time reference counter of the transport reaches zero, an idle timer will start. When the idle timer expires and the reference counter is still zero, transport manager will destroy the transport by calling **`pjsip_transport_unregister()`**. This function unregisters the transport from transport manager's hash table and eventually destroy the transport.

Some transports need to exist forever even when nobody is using the transport (for example, UDP transport, which is a singleton instance). To prevent that transport from being deleted, it must set the reference counter to one initially, so that reference counter will never reach zero.

Transport Error Handling

Any errors in the transport (such as failure to send packet or connection reset) are handled by transport user. Transport object doesn't need to handle such errors, other than reporting the error in the function's return value. In particular, it must not try to reconnect a failed/closed connection.

5.2 Using Transports

5.2.1 Function Reference

```
pj_status_t
pjsip_endpt_acquire_transport(   pjsip_endpoint *endpt,
                                 pjsip_transport_type_e t_type,
                                 const pj_sockaddr_t *remote_addr,
                                 int addrlen,
                                 pjsip_transport **p_transport);
```

Acquire transport of type *t_type* to be used to send message to destination *remote_addr*. Note that if transport is successfully acquired, the transport's reference counter will be incremented.

```
pj_status_t pjsip_transport_add_ref( pjsip_transport *transport );
```

Add reference counter of the *transport*. This function will prevent the transport from being destoyed, and it also cancels idle timer if such timer is active.

```
pj_status_t pjsip_transport_dec_ref( pjsip_transport *transport );
```

Decrement reference counter of the *transport*. When transport's reference counter reaches zero, an idle timer will be started and transport will be destroyed by transport manager when the timer has elapsed and reference counter is still zero.

```
pj_status_t pjsip_transport_send(pjsip_transport *transport,
                                 pjsip_tx_data *tdata,
                                 const pj_sockaddr_t *remote_addr,
                                 int addrlen,
                                 void *token,
                                 void (*cb)(void *token,
                                            pjsip_tx_data *tdata,
                                            pj_ssize_t bytes_sent));
```

Send the message in *tdata* to *remote_addr* using transport *transport*. If the function completes immediately and data has been sent, the function returns PJ_SUCCESS. If the function completes immediately with error, a non-zero error code will be returned. In both cases, the callback will not be called.

If the function can not complete immediately (e.g. when the underlying socket buffer is full), the function will return PJ_EPENDING, and caller will be notified about the completion via the callback *cb*. If the pending send operation completes with error, the error code will be indicated as negative value of the error code, in the *bytes_sent* argument of the callback (to get the error code, use "`pj_status_t status = -bytes_sent`").

This function sends the message as is.

5.3 Extending Transports

PJSIP transport can be extended to use custom defined transports. Theoretically any types of transport, not limited to TCP/IP, can be plugged into the transport manager's framework. Please see the header file **<pjsip/sip_transport.h>** and also **sip_transport_udp.[hc]** for more details.

5.4 Initializing Transports

PJSIP doesn't start any transports by default (not even the built-in transports); it is the responsibility of the application to initialize and start any transports that it wishes to use.

Below are the initialization functions for the built-in UDP and TCP transports.

5.4.1 UDP Transport Initialization

PJSIP provides two choices to initialize and start UDP transports. These functions are declared in <pjsip/sip_transport_udp.h>.

```
pj_status_t pjsip_udp_transport_start(  pjsip_endpoint *endpt,
                                         const pj_sockaddr_in *local_addr,
                                         const pj_sockaddr_in *pub_addr,
                                         unsigned async_cnt,
                                         pjsip_transport **p_transport );
```

Create, initialize, register, and start a new UDP transport. The UDP socket will be bound to *local_addr*. If the endpoint is located behind firewall/NAT or other port-forwarding devices, then *pub_addr* can be used as the address that is advertised for this transport; otherwise *pub_addr* should be the same as *local_addr*. The argument *async_cnt* specifies how many simultaneous operations are allowed for this transport, and for maximum performance, the value should be equal to the number of processors in the node.

If transport is successfully started, the function returns PJ_SUCCESS and the transport is returned in *p_transport* argument, should the application want to use the transport immediately. Application doesn't need to register the transport to transport manager; this function has done that when the function returns successfully.

Upon error, the function returns a non-zero error code.

```
pj_status_t pjsip_udp_transport_attach( pjsip_endpoint *endpt,
                                         pj_sock_t sock,
                                         const pj_sockaddr_in *pub_addr,
                                         unsigned async_cnt,
                                         pjsip_transport **p_transport);
```

Use this function to create, initialize, register, and start a new UDP transport when the UDP socket is already available. This is useful for example when application has just resolved the public address of the socket with STUN, and instead of closing the socket and re-create it, the application can just reuse the same socket for the SIP transport.

5.4.2 TCP Transport Initialization

TODO.

5.4.3 TLS Transport Initialization

TODO.

5.4.4 SCTP Transport Initialization

TODO.

Chapter 6:Sending Messages

The core operations in SIP applications are of course sending and receiving message. Receiving incoming message is handled in **`on_rx_request()`** and **`on_rx_response()`** callback of each module, as described in 1 General Design.

This chapter will describe about the basic way to send outgoing messages, i.e. without using transaction or dialog.

The next chapter Transactions describes about how to handle request statefully (both incoming and outgoing requests).

6.1 Sending Messages Overview

6.1.1 Creating Messages

PJSIP provides rich API to create request or response messages. There are various ways to create messages:

- for response messages, the easiest way is to use **`pjsip_endpt_create_response()`** function.
- for request messages, you can use **`pjsip_endpt_create_request()`**, **`pjsip_endpt_create_request_from_hdr()`**, **`pjsip_endpt_create_ack()`**, or **`pjsip_endpt_create_cancel()`**.
- proxies can create request or response messages based on incoming message to be forwarded by calling **`pjsip_endpt_create_request_fwd()`** and **`pjsip_endpt_create_response_fwd()`**.
- alternatively you may create request or response messages manually by creating the transmit buffer with **`pjsip_endpt_create_tdata()`**, creating the message with **`pjsip_msg_create()`**, adding header fields to the message with **`pjsip_msg_add_hdr()`** or **`pjsip_msg_insert_first_hdr()`**, set the message body, etc.
- higher layer module may provide more specific way to create message (e.g. dialog layer). This will be described in the individual module's documentation.

All message creating API (except the low-level **`pjsip_endpt_create_tdata()`**) sets the reference counter of the transmit buffer (*pjsip_tx_data*) to one, which means that at some point application (or stack) must decrement the reference counter to destroy the transmit buffer.

All message sending API will decrement transmit buffer's reference counter. Which means that as long as application doesn't do anything with the transmit buffer's reference counter, the buffer will be destroyed after it is sent.

6.1.2 Sending Messages

The most basic way to send message is to call **`pjsip_endpt_acquire_transport()`** and **`pjsip_transport_send()`** functions. For this to work, however, you must know the destination address (i.e. sockaddr, not just hostname) to send the message. Since there can be several steps from having the message and getting the exact socket address (e.g. determining which address to use, performing RFC 3263 lookup, etc.), practically this function is too low-level to be used directly.

The core API to send messages are **pjsip_endpt_send_request_stateless()** and **pjsip_endpt_send_response()** functions. These two are very powerfull functions in the sense that it handles transport layer automatically, and are the basic building-blocks used by upper layer modules (e.g. transactions).

The **pjsip_endpt_send_request_stateless()** function are for sending request messages, and it performs the following procedures:

- Determine which destination to contact based on the Request-URI and parameters in Route headers,
- Resolve the destination server using procedures in RFC 3263 (Locating SIP Servers),
- Select and establish transport to be used to contact the server,
- Modify sent-by in Via header to reflect current transport being used,
- Send the message using current transport,
- Fail-over to next server/transport if server can not be contacted using current transport

The **pjsip_endpt_send_response()** function are for sending response messages, and it performs the following procedures:

- Follow the procedures in Section 18.2.2 of RFC 3261 to select which transport to use and which address to send response to,
- Additionally conform to RFC 3581 about rport parameter,
- Send the response using the selected transport,
- Fail-over to next address when response failed to be sent using the selected transport, resolving the server according to RFC 3263 when necessary.

Since messages may be sent asynchronously (e.g. after TCP has been connected), both functions provides callback to notify application about the status of the transmission. This callback also inform the application that fail-over will happen (or not), and application has the chance to override the behavior.

6.2 Function Reference

6.2.1 Sending Response

Base Functions

```
pj_status_t pjsip_endpt_create_response(   pjsip_endpoint *endpt,
                                           pjsip_rx_data *rdata,
                                           int st_code,
                                           const char *st_text,
                                           pjsip_tx_data **tdata);
```

Create a standard response message for the request in *rdata* with status code *st_code* and status text *st_text*. If *st_text* is NULL, default status text will be used.

```
pj_status_t pjsip_get_response_addr( pj_pool_t *pool,
                                     pjsip_rx_data *rdata,
                                     pjsip_response_addr *res_addr);
```

Determine which address (and transport) to use to send response message based on the received request in *rdata*. This function follows the specification in section 18.2.2 of RFC 3261 and RFC 3581 for calculating the destination address and transport. The address and transport information about destination to send the response will be returned in *res_addr* argument.

```
pj_status_t pjsip_endpt_send_response(  pjsip_endpoint *endpt,
                                        pjsip_response_addr *res_addr,
                                        pjsip_tx_data *response,
                                        void *token,
                                        void (*cb)(  pjsip_send_state*,
                                                     pj_ssize_t sent,
                                                     pj_bool_t *cont));
```

Send response in *response* statelessly, using the destination address and transport in *res_addr*. The response address information (*res_addr*) is normally initialized by calling `pjsip_get_response_addr()`.

The definite status of the transmission will be reported when callback *cb* is called, along with other information (including the original *token*) which will be stored in *pjsip_send_state*. If message was successfully sent, the *sent* argument of the callback will be a non-zero positive number. If there is failure, the *sent* argument will be negative value, and the error code is the positive part of the value (i.e. status=-sent). If *cont* argument value is non-zero, it means the function will try other addresses to send the message (i.e. fail-over). Application can choose not to try other addresses by setting this argument to zero upon exiting the callback.

If application doesn't specify callback *cb*, then the function will not fail-over to next address in case the selected transport fails to deliver the message.

The function returns PJ_SUCCESS if the message is valid, or a non-zero error code. However, even when it returns PJ_SUCCESS, there is no guarantee that the response has been successfully sent.

Note that callback MAY be called before the function returns.

Composite Functions

```
pj_status_t pjsip_endpt_respond_stateless(  pjsip_endpoint *endpt,
                                            pjsip_rx_data *rdata,
                                            int st_code,
                                            const char *st_text,
                                            const pjsip_hdr *hdr_list,
                                            const pjsip_msg_body *body);
```

This function creates and sends a response to an incoming request. In addition, caller may specify message body and additional headers to be put in the response message in the *hdr_list* and *body* argument. If there is no additional header or body, to be sent, the arguments should be NULL.

The function returns PJ_SUCCESS if response has been successfully created and send to transport layer, or a non-zero error code. However, even when it returns PJ_SUCCESS, there is no guarantee that the response has been successfully sent.

6.2.2 Sending Request

```
pj_status_t pjsip_endpt_create_tdata(   pjsip_endpoint *endpt,
                                        pjsip_tx_data **tdata);
```

Create a new, blank transmit data.

```
pj_status_t pjsip_endpt_create_request( pjsip_endpoint *endpt,
                                        const pjsip_method *method,
                                        const pj_str_t *target,
                                        const pj_str_t *from,
                                        const pj_str_t *to,
                                        const pj_str_t *contact,
                                        const pj_str_t *call_id,
                                        int cseq,
                                        const pj_str_t *text,
```

```
                                    pjsip_tx_data **p_tdata);
```

Create a new request message of the specified *method* for the specified *target* URI, *from*, *to*, and *contact*. The *call_id* and *cseq* are optional. If *text* is specified, then a "text/plain" body is added. The request message has initial reference counter set to 1, and is then returned to sender in *p_tdata*.

```
pj_status_t pjsip_endpt_create_request_from_hdr(pjsip_endpoint *endpt,
                                            const pjsip_method *method,
                                            const pjsip_uri *target,
                                            const pjsip_from_hdr *from,
                                            const pjsip_to_hdr *to,
                                            const pjsip_contact_hdr *ch,
                                            const pjsip_cid_hdr *call_id,
                                            int cseq,
                                            const pj_str_t *text,
                                            pjsip_tx_data **p_tdata);
```

Create a new request header by shallow-cloning the headers from the specified arguments.

```
pj_status_t pjsip_endpt_create_ack(  pjsip_endpoint *endpt,
                                     const pjsip_tx_data *tdata,
                                     const pjsip_rx_data *rdata,
                                     pjsip_tx_data **ack );
```

Create ACK request message from the original request in *tdata* based on the received response in *rdata*. This function is normally used by transaction when it receives non-successful response to INVITE. An ACK request for successful INVITE response is normally generated by dialog's create request function.

```
pj_status_t pjsip_endpt_create_cancel(  pjsip_endpoint *endpt,
                                        const pjsip_tx_data *tdata,
                                        pjsip_tx_data **p_tdata);
```

Create CANCEL request based on the previously sent request in *tdata*. This will create a new transmit data buffer in *p_tdata*.

NEW!

```
pj_status_t pjsip_endpt_send_request_stateless(pjsip_endpoint *endpt,
                                            pjsip_tx_data *tdata,
                                            void *token,
                                            void (*cb)(pjsip_send_state*,
                                                       pj_ssize_t sent,
                                                       pj_bool_t *cont));
```

Send request in *tdata* statelessly. The function will take care of which destination and transport to use based on the information in the message, taking care of URI in the request line and Route header. There are several steps will be performed by this function:

- determine which host to contact based on Request-URI and Route headers (*pjsip_get_request_addr()*),
- resolve the destination host (*pjsip_endpt_resolve()*),
- acquire transport to be used (*pjsip_endpt_acquire_transport()*).
- send the message (*pjsip_transport_send()*).
- fail-over to next address/transport if necessary.

The definite status of the transmission will be reported when callback *cb* is called, along with other information (including the original *token*) which will be stored in *pjsip_send_state*. If message was successfully sent, the *sent* argument of the callback will be a non-zero positive number. If there is failure, the *sent* argument will be negative value, and the error code is the positive part of the value (i.e. status=-sent). If *cont* argument value is non-zero, it means the function will try other addresses to send the message (i.e. fail-over). Application can choose not to try other addresses by setting this argument to zero upon exiting the callback.

If application doesn't specify callback *cb*, then the function will not fail-over to next address in case the selected transport fails to deliver the message.

The function returns PJ_SUCCESS if the message is valid, or a non-zero error code. However, even when it returns PJ_SUCCESS, there is no guarantee that the request has been successfully sent.

Note that callback MAY be called before the function returns.

6.2.3 Stateless Proxy Forwarding

Proxy may choose to forward a request statelessly. When doing so however, it must strictly follow guidelines in section **16.11 Stateless Proxy** of RFC 3261.

```
pj_status_t pjsip_endpt_create_request_fwd(pjsip_endpoint *endpt,
                                           pjsip_rx_data *rdata,
                                           const pjsip_uri *uri,
                                           const pj_str_t *branch,
                                           unsigned options,
                                           pjsip_tx_data **tdata);
```

Create new request message to be forwarded upstream to new destination URI *uri*. The new request is a full/deep clone of the request received in *rdata*, unless if other copy mechanism is specified in the *options*. The *branch* parameter, if not NULL, will be used as the branch-param in the Via header. If it is NULL, then a unique branch parameter will be used.

```
pj_status_t pjsip_endpt_create_response_fwd(   pjsip_endpoint *endpt,
                                               pjsip_rx_data *rdata,
                                               unsigned options,
                                               pjsip_tx_data **tdata);
```

Create new response message to be forwarded downstream by the proxy from the response message found in *rdata*. Note that this function practically will clone the response as is, i.e. without checking the validity of the response or removing top most Via header. This function will perform full/deep clone of the response, unless other copy mechanism is used in the *options*.

```
pj_str_t pjsip_calculate_branch_id(  pjsip_rx_data *rdata );
```

Create a globally unique branch parameter based on the information in the incoming request message. This function guarantees that subsequent retransmissions of the same request will generate the same branch id.

This function can also be used in the loop detection process. If the same request arrives back in the proxy with the same URL, it will calculate into the same branch id.

Note that the returned string was allocated from rdata's pool.

6.3 Examples

6.3.1 Sending Responses

Sending Account Not Found Response Statelessly

```
static pj_bool_t on_rx_request(pjsip_rx_data *rdata )
{
   pjsip_account *acc;
   pj_status_t status;

   // Find account referred to in the request.
   acc = ...

   // Respond statelessly if account can not be found.
   if (!acc) {
      status = pjsip_endpt_respond_stateless( endpt, rdata, 404, NULL /*Not Found*/,
                                              NULL, NULL, NULL);
      return PJ_TRUE;
   }

   // Process the account
   ...

   return PJ_TRUE;
}
```

Code 24 Sample: Stateless Response

Handling Authentication Failures Statelessly

Another (longer) way to send stateless response:

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pjsip_account *acc;

   // Lookup acc.
   acc = ...;

   // Check authorization and handle failure statelessly
   if (!pjsip_auth_authorize( acc, rdata->msg )) {
      pjsip_proxy_authenticate_hdr *auth_hdr;

      status = pjsip_endpt_create_response(  endpt, rdata,
                                             407, NULL /* Proxy Auth Required */,
                                             &tdata);

      // Add Proxy-Authenticate header.
      status = pjsip_auth_create_challenge( tdata->pool, ..., &auth_hdr);
      pjsip_msg_add_hdr( &tdata->msg, auth_hdr );

      // Send response statelessly
      status = pjsip_endpt_send_response( endpt, tdata, NULL);
      return PJ_TRUE;
   }

   // Authorization success. Proceed to next stage..
   ...
   return PJ_TRUE;
}
```

Code 25 Sample: Stateless Response

Stateless Redirection

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pjsip_account *acc;
   pj_status_t status;

   // Find the account referred to in the request.
   acc = ...
   if (!acc) {
      status = pjsip_endpt_respond_stateless( endpt, rdata, 404, NULL /*Not Found*/,
                                              NULL, NULL, NULL );
      return PJ_TRUE;
   }

   //
   // Send 301/Redirect message, specifying the Contact details in the response
   //
   status = pjsip_endpt_respond_stateless(   endpt, rdata,
                                             301, NULL /*Moved Temporarily*/,
                                             &acc->contact_list, NULL, NULL);
   return PJ_TRUE;
}
```

Code 26 Stateless Redirection

6.3.2 Sending Requests

Sending Request Statelessly

```
void my_send_request()
{
   pj_status_t status;
   pjsip_tx_data *tdata;

   // Create the request.
   // Actually the function takes pj_str_t* argument instead of char*.
   status = pjsip_endpt_create_request(  endpt,                      // endpoint
                                         method,                     // method
                                         "sip:bob@example.com",      // target URI
                                         "sip:alice@thishost.com",   // From:
                                         "sip:bob@example.com",      // To:
                                         "sip:alice@thishost.com",   // Contact:
                                         NULL,                       // Call-Id
                                         0,                          // CSeq#
                                         NULL,                       // body
                                         &tdata );                   // output

   // You may modify the message before sending it.
   ...

   // Send the request statelessly (for whatever reason...)
   status = pjsip_endpt_send_request_stateless( endpt, tdata, NULL);
}
```

Code 27 Sending Stateless Request

6.3.3 Stateless Forwarding

Stateless Forwarding

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pjsip_account *acc;
   pjsip_tx_data *tdata;
   pj_str_t branch_id;
   pj_status_t status;

   // Find the account specified in the request.
   acc = ...

   // Generate unique branch ID for the request.
   branch_id = pjsip_calculate_branch_id( rdata );

   // Create new request to be forwarded to new destination.
   status = pjsip_endpt_create_request_fwd(  endpt, rdata, dest, &branch_id, 0,
                                             &tdata );

   // The new request is good to send, but you may modify it if necessary
   // (e.g. adding/replacing/removing headers, etc.)
   ...

   // Send the request downstream
   status = pjsip_endpt_send_request_stateless( endpt, tdata, NULL );

   return PJ_TRUE;
}

//
// Forward response upstream
//
static pj_bool_t on_rx_response(pjsip_rx_data *rdata)
{
   pjsip_tx_data *tdata;
   pj_status_t status;

   // Check that topmost Via is ours, strip top-most Via, etc.
   ...

   // Create new tdata for the response.
   status = pjsip_endpt_create_response_fwd( endpt, rdata, 0, &tdata );

   // Send the response upstream
   status = pjsip_endpt_send_response( endpt, tdata, NULL);

   return PJ_TRUE;
}
```

Code 28 Stateless Forwarding

Chapter 7:Transactions

7.1 Design

7.1.1 Introduction

Transaction in PJSIP is represented with `pjsip_transaction` structure in header file <pjsip/sip_transaction.h>. Transaction's lifetime normally follows these steps:

- Created by `pjsip_tsx_endpt_create_uac()` / `pjsip_tsx_create_uas()`.
- When application wants to send a message using the transaction, it will call `pjsip_tsx_send_msg()`.
- Transaction state automatically changes as messages are passed to it (either by endpoint for incoming message or by transaction user for outgoing message) or timer elapses, and transaction user is notified via `on_tsx_state()` callback.
- Transaction will be automatically destroyed once it the state has reached PJSIP_TSX_STATE_TERMINATED. Application can also forcely terminate the transaction by calling `pjsip_tsx_terminate()`.

7.1.2 Timers and Retransmissions

Transaction only has two types of timers: retransmission timer and timeout timer. The value of both timer types are automatically set by the transaction according to the transaction type (UAS or UAC), transport (reliable or non-reliable), and method (INVITE or non-INVITE).

Application can change the interval value of timers only on a global basis (perhaps even only during compilation).

A transaction handles both incoming and outgoing retransmissions. Incoming retransmissions are silently absorbed and ignored by transaction; there is no notification about incoming retransmissions emitted by transaction. Outgoing messages are automatically retransmitted by transactions where necessary; again there will be no notification emitted by transaction on outgoing retransmissions.

7.1.3 INVITE Final Response and ACK Request

Failed INVITE Request

> The transaction behaves ***exactly*** according to RFC 3261 for failed INVITE request.

Client transaction: when a client INVITE transaction receives 300-699 final response to INVITE, it will automatically emit ACK request to the response. The transaction then wait for timer D interval before it is terminated, during which any incoming 300-699 response retransmissions will be automatically answered with ACK request.

Server transaction: when a server INVITE transaction is asked to transmit 300-699 final response, it will transmit the response and keep retransmitting the response until an ACK request is received or timer H interval has elapsed. During this interval, when ACK request is received, transaction will move to Confirmed state and will be destroyed after timer I interval has elapsed. When timer H elapsed without receving a valid ACK request, transaction will be destroyed.

Successfull INVITE Request

Client transaction: when a client INVITE transaction receives 2xx final response to INVITE, it will destroy itself automatically after it passes the response to its transaction user (can be a dialog or application). Subsequent incoming 2xx response retransmission will be passed directly to dialog or application.

In any case, application MUST send ACK request manually upon receiving 2xx final response to INVITE.

Server transaction: when a server INVITE transaction is asked to transmit 2xx final response, it will transmit the response and **keep retransmitting** the response until ACK is received or transaction is terminated by application with `pjsip_tsx_terminate()`.

For simplicity in the implementation, a typical UAS dialog normally will let the transaction handle the retransmission of the 2xx INVITE response. But proxy application MUST destroy the UAS transaction as soon as it receives and sends the 2xx response, to allow the 2xx retransmission to be handled by end-to-end user agents.

> This behavior of INVITE server transaction is ***different*** than RFC 3261 for successfull INVITE request, which says that INVITE server transaction MUST be destroyed once 2xx response is sent. The PJSIP transaction behavior allows more simplicity in the dialog implementation, while maintaining the flexibility to be compliant with RFC 3261 for proxy applications.

The default behavior of the INVITE server transaction can be overridden by setting `transaction->handle_200resp` to zero (default is non-zero) after transaction is created. In this case, UAS INVITE transaction will be destroyed as soon as 2xx response to INVITE is sent.

7.1.4 Incoming ACK Request

When the server INVITE transaction was completed with non-successful final response, the ACK request will be absorbed by transaction; transaction user **WILL NOT be notified** about the incoming ACK request.

When the server INVITE transaction was completed with 2xx final response, the first ACK request will be notified to transaction user. Subsequent receipt of ACK retransmission WILL NOT be notified to transaction user.

7.1.5 Server Resolution and Transports

Transaction uses the core API `pjsip_endpt_send_request_stateless()` and `pjsip_endpt_send_response()` to send outgoing messages. These functions provide server resolution and transport establishment to send the message, and fail over to alternate transport when a failure is detected. The transaction uses the callbacks provided by these functions to monitor the progress of the transmission and track the transport being used.

The transaction adds reference counter to the transport it currently uses.

TCP Connection Closure

A TCP connection closure will not automatically cause the transaction to fail. In fact, the transaction will not even detect the failure until it tries to send a message. When it does, it follows the normal procedure to send the message using alternative transport.

7.1.6 Via Header

Branch Parameter

UAC transaction automatically generates a unique branch parameter in the Via header when one is not present. If branch parameter is already present, the transaction will used it as its key, complying to rules set by both RFC 3261 and RFC 2543.

Via Sent-By

Via sent-by is always put by `pjsip_endpt_send_request_stateless()` and `pjsip_endpt_send_response()`.

7.2 Reference

7.2.1 Base Functions

```
pj_status_t pjsip_tsx_layer_init(pjsip_endpoint *endpt );
```

Initialize and register the transaction layer module to the specified endpoint.

```
pjsip_module *pjsip_tsx_layer_instance(void);
```

Get the instance of transaction layer module.

```
pj_status_t pjsip_tsx_layer_destroy(void);
```

Shutdown the transaction layer module and unregister it from the endpoint where it currently registered.

```
pj_status_t pjsip_tsx_create_uac (   pjsip_module *tsx_user,
                                     pjsip_tx_data *tdata,
                                     pjsip_transaction **p_tsx );
```

Create a new UAC transaction for the outgoing request in *tdata* with the transaction user set to *tsx_user*. The transaction is automatically initialized and registered to the transaction table. Note that after calling this function, applications normally would call `pjsip_tsx_send_msg()` to actually send the request.

```
pj_status_t pjsip_tsx_create_uas (   pjsip_module *tsx_user,
                                     pjsip_rx_data *rdata,
                                     pjsip_transaction **p_tsx );
```

Create a new UAS transaction for the incoming request in *rdata* with the transaction user set to *tsx_user*. The transaction is automatically initialized and registered to endpoint's transaction table.

```
pj_status_t pjsip_tsx_send_msg(   pjsip_transaction *tsx,
                                  pjsip_tx_data *tdata );
```

Send message through the transaction. If *tdata* is NULL, the last message or the message that was specified during creation will be retransmitted. When the function returns PJ_SUCCESS, the tdata reference counter will be decremented.

```
pj_status_t pjsip_tsx_create_key(pj_pool_t *pool,
                                 pj_str_t *out_key,
```

```
                                pjsip_role_e role,
                                const pjsip_method *method,
                                const pjsip_rx_data *rdata);
```

Create a transaction key from an incoming request or response message, taking into consideration whether the message is compliant with RFC 3261 or RFC 2543. The key can be used to find the transaction in endpoint's transaction table.

The function returns the key in *out_key* parameter. The *role* parameter is used to find either UAC or UAS transaction, and the *method* parameter contains the method of the message.

```
pjsip_transaction* pjsip_tsx_layer_find_tsx(   const pj_str_t *key,
                                               pj_bool_t lock );
```

Find transaction with the specified key in transaction table. If *lock* parameter is non-zero, this function will also lock the transaction before returning the transaction, so that other threads are not able to delete the transaction. Caller then is responsible to unlock the transaction when it's finished using the transaction, using pj_mutex_unlock().

```
pj_status_t pjsip_tsx_terminate( pjsip_transaction *tsx,
                                 int st_code );
```

Forcefully terminate the transaction *tsx* with the specified status code *st_code*. Normally application doesn't need to call this function, since transactions will terminate and destroy themselves according to their state machine.

This function is used for example when 200/OK response to INVITE is sent/received and the UA layer wants to handle retransmission of 200/OK response manually.

The transaction will emit transaction state changed event (state changed to PJSIP_TSX_STATE_TERMINATED), then it will be unregistered and destroyed immediately by this function.

```
pjsip_transaction* pjsip_rdata_get_tsx (   pjsip_rx_data *rdata );
```

Get the transaction object in an incoming message.

7.2.2 Composite Functions

```
pj_status_t pjsip_endpt_respond( pjsip_endpoint *endpt,
                                 pjsip_module *tsx_user,
                                 pjsip_rx_data *rdata,
                                 int st_code,
                                 const char *st_text,
                                 const pjsip_hdr *hdr_list,
                                 const pjsip_msg_body *body,
                                 pjsip_transaction **p_tsx)
```

Send respond by creating a new UAS transaction for the incoming request.

```
pj_status_t pjsip_endpt_send_request(pjsip_endpoint *endpt,
                                     pjsip_tx_data *tdata,
                                     int timeout,
                                     void *token,
                                     void (*cb)(void*, pjsip_event*))
```

Send the request by using an UAC transaction, and optionally request callback to be called when the transaction completes.

7.3 Sending Statefull Responses

7.3.1 Usage Examples

Sending Response Statefully (The Hard Way)

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pj_status_t status;
   pjsip_transaction *tsx;
   pjsip_tx_data *tdata;

   // Create and initialize transaction.
   status = pjsip_endpt_create_uas_tsx( endpt, NULL, rdata, &tsx );

   // Create response
   status = pjsip_endpt_create_response( endpt, rdata, 200, NULL /*OK*/, &tdata);

   // The response message is good to send, but you may modify it before
   // sending the response.
   ...

   // Send response with the specified transaction.
   pjsip_tsx_send_msg( tsx, tdata );

   return PJ_TRUE;
}
```

Code 29 Sending Statefull Response

Sending Response Statefully (The Easy Way)

```
static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pj_status_t status;

   // Respond to the request statefully
   status = pjsip_endpt_respond( endpt, NULL, rdata,
                                 200, NULL /* OK */, NULL, NULL, NULL);

   return PJ_TRUE;
}
```

Code 30 Sending Statefull Response

7.4 Sending Statefull Request

Two ways to send statefull request:

- use pjsip_endpt_send_request()
- using transaction manually.

7.4.1 Usage Examples

Sending Request with Transaction

```
extern pjsip_module app_module;

void my_send_request()
{
   pj_status_t status;
   pjsip_tx_data *tdata;
```

```
   pjsip_transaction *tsx;

   // Create the request.
   status = pjsip_endpt_create_request(  endpt, ..., &tdata );

   // You may modify the message before sending it.
   ...

   // Create transaction.
   status = pjsip_endpt_create_uac_tsx( endpt, &app_module, tdata, &tsx );

   // Send the request.
   status = pjsip_tsx_send_msg( tsx, tdata /*or NULL*/);
}

static void on_tsx_state( pjsip_transaction *tsx, pjsip_event *event )
{
   pj_assert(event->type == PJSIP_EVENT_TSX_STATE);
   PJ_LOG(3,("app", "Transaction %s: state changed to %s",
                    tsx->obj_name, pjsip_tsx_state_str(tsx->state)));
}
```

Code 31 Sending Request Statefully

7.5 Statefull Proxy Forwarding

7.5.1 Usage Examples

Statefull Forwarding

The following code shows a sample statefull forwarding proxy. The code creates UAS and UAC transaction (one for each side), forward the request to the UAC side, and forward all responses from the UAC side to UAS side. It also handles transaction timeout or other error in the UAC side and sends response to the UAS side.

One that it doesn't handle is receiving CANCEL request in the UAC side.

```
// This is our proxy module.
extern pjsip_module proxy_module;

static pj_bool_t on_rx_request( pjsip_rx_data *rdata )
{
   pjsip_account *acc;
   pjsip_uri *dest;
   pjsip_transaction *uas_tsx, *uac_tsx;
   pjsip_tx_data *tdata;
   pj_status_t status;

   // Find the account specified in the request.
   acc = ...

   // Respond statelessly with 404/Not Found if account can not be found.
   if (!acc) {
      ...
      return PJ_TRUE;
   }
   // Set destination URI from account's contact list that has highest priority.
   dest = ...

   // Create UAS transaction
   status = pjsip_endpt_create_uas_tsx( endpt, &proxy_module, rdata, &uas_tsx);

   // Copy request to new tdata with new target URI.
   status = pjsip_endpt_create_request_fwd( endpt, rdata, dest, NULL, 0, &tdata);

   // Create new UAC transaction.
```

```
   status = pjsip_endpt_create_uac_tsx( endpt, &proxy_module, tdata, &uac_tsx );

   // "Associate" UAS and UAC transaction
   uac_tsx->mod_data[proxy_module.id] = (void*)uas_tsx;
   uas_tsx->mod_data[proxy_module.id] = (void*)uac_tsx;

   // Forward message to UAC side
   status = pjsip_tsx_send_msg( uac_tsx, tdata );
   return PJ_TRUE;
}

static pj_bool_t on_rx_response( pjsip_rx_data *rdata )
{
   pjsip_transaction *tsx;
   pjsip_tx_data *tdata;
   pj_status_t status;

   // Get transaction object in rdata.
   tsx = pjsip_rdata_get_tsx( rdata );

   // Check that this transaction was created by the proxy
   if (tsx->tsx_user == &proxy_module) {
      // Get the peer UAC transaction.
      pjsip_transaction *uas_tsx;
      uas_tsx = (pjsip_transaction*) tsx->mod_data[proxy_module.id];

      // Check top-most Via is ours
      ...
      // Strip top-most Via
      // Note that after this code, rdata->msg_info.via is invalid.
      pj_list_erase(rdata->msg_info.via);
      // Code above is equal to:
      //  pjsip_hdr *via = pjsip_msg_find_hdr(rdata->msg, PJSIP_H_VIA);
      //  pj_list_erase(via);

      // Copy the response msg.
      status = pjsip_endpt_create_response_fwd( endpt, rdata, 0, &tdata);

      // Forward the response upstream.
      pjsip_tsx_send_msg( uas_tsx, tdata );

      return PJ_TRUE;
   }
   ...
}
```

Code 32 Statefull Forwarding

Chapter 8:Authentication Framework

PJSIP provides framework for performing both client and server authentication. The authentication framework supports HTTP digest authentication by default, but other authentication schemes may be added to the framework.

The following diagram illustrates the framework's "class diagram".

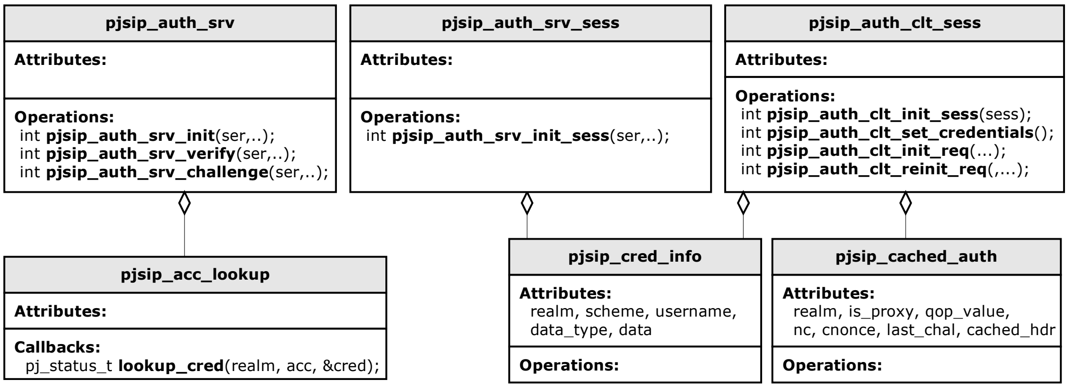

Figure 13 Authentication Framework

8.1 Client Authentication Framework

The client authentication framework manages authentication process by client to all downstream servers. It can respond to server's challenge with the correct credential (when such credential is supplied), cache the authorization info, and initialize subsequent requests with the cached authorization info.

8.1.1 Client Authentication Framework Reference

The authentication APIs are declared in <pjsip/sip_auth.h>. Below are the documentation reference for authentication data structures and functions.

Data Structure Reference

UPDATED!

Structure	Description
`pjsip_cred_info`	This structure describes the credential to be used to authenticate against a specific realm. A client can have multiple credentials to use for the duration of a dialog or registration; each one of the credential contains information needed to authenticate against a particular downstream proxy or server. For example, client needs one credential to authenticate against its outbound proxy, and another credential to authenticate against the end server.
`pjsip_cached_auth`	This structure keeps the last challenge received from a particular server. It is needed so that client can initialize next request with the last challenge.
`pjsip_auth_clt_sess`	This structure describes the client authentication session. Client would normally keep this structure for

	the duration of a dialog or client registration.

Figure 14 Client Authentication Data Structure

Function Reference

pj_status_t pjsip_auth_clt_init(pjsip_auth_clt_sess *sess, pj_pool_t *pool, unsigned options);

Initialize client authentication session data structure, and set the session to use *pool* for its subsequent memory allocation. The argument *options* should be set to zero for this PJSIP version.

pj_status_t pjsip_auth_clt_set_credentials(pjsip_auth_clt_sess *s, int cred_cnt, const pjsip_cred_info cred[]);

Set the credentials to be used during the session. This will duplicate the specified credentials using client authentication's pool.

pj_status_t pjsip_auth_clt_init_req(pjsip_auth_clt_sess *sess, pjsip_tx_data *tdata);

This function add all relevant authorization headers to a new outgoing request *tdata* according to the cached information in the session. The request line in the request message must be valid before calling this function.

pj_status_t pjsip_auth_clt_reinit_req(pjsip_auth_clt_session *sess, pjsip_endpoint *endpt, const pjsip_rx_data *rdata, pjsip_tx_data *old_request, pjsip_tx_data **new_request);

Call this function to re-initialize a request upon receiving failed authentication status (401/407 response). This function will recreate *new_request* according to *old_request*, and add appropriate Authorization and Proxy-Authorization headers according to the challenges found in *rdata* response. In addition, this function also put the relevant information in the session.

This function will return failure if there is a missing credential for the challenge. Note that this function may reuse the old request instead of creating a fresh one.

8.1.2 Examples

Client Transaction Authentication

The following example illustrates how to initialize outgoing request with authorization information and how to handle challenge received from the server. For brevity, error handling is not shown in the example. A real application should be prepared to handle error situation in all stages.

```
pjsip_auth_client_session auth_sess;

// Initialize client authentication session with some credentials.
void init_auth(pj_pool_t *session_pool)
{
   pjsip_cred_info cred;
   pj_status_t status;

   cred.realm = pj_str(“sip.example.com”);
   cred.scheme = pj_str(“digest”);
   cred.username = pj_str(“alice”);
   cred.data_type = PJSIP_CRED_DATA_PLAIN_PASSWD;
   cred.data = pj_str(“secretpassword”);
```

```
   status = pjsip_auth_client_init( &auth_sess, session_pool, 0);
   status = pjsip_auth_set_credentials( &auth_sess, 1, &cred );
}

// Initialize outgoing request with authorization information and
// send the request statefully.
void send_request(pjsip_tx_data *tdata)
{
   pj_status_t status;

   status = pjsip_auth_client_init_req( &auth_sess, tdata );
   status = pjsip_endpt_send_request( endpt, tdata, -1, NULL, &on_complete);
}

// Callback when the transaction completes.
static void on_complete( void *token, pjsip_event *event )
{
   int code;

   pj_assert(event->type == PJSIP_EVENT_TSX_STATE);
   code = event->body.tsx_state.tsx->status_code;
   if (code == 401 || code == 407) {
      pj_status_t status;
      pjsip_tx_data *new_request;

      status = pjsip_auth_client_reinit_req( &auth_sess, endpt,
                                             event->body.tsx_state.src.rdata,
                                             tsx->last_tx,
                                             &new_request);
      if (status == PJ_SUCCESS)
         status = pjsip_endpt_send_request(  endpt, new_request, -1, NULL,
                                             &on_complete);
      else
         PJ_LOG(3,("app","Authentication failed!!!"));
   }
}
```

Code 33 Client Athorization Example

8.2 Server Authorization Framework

The server authorization framework provides two types of server authorization mechanisms:

- session-less server authorization, which provides general API for authenticating clients. This API provides global server authorization mechanism on request-per-request basis, and is normally used for proxy application when it doesn't do call statefull.
- server authorization session, which provides API for authenticating requests inside a particular dialog or registration session. One server authorization session instance needs to be created for each server side dialog or registration session. A server auth session will have exactly one credential which is setup initially, and this credential must be used by client throughout the duration of the dialog/registration session.

The server authorization session currently is not implemented. Only global, session-less server authorization framework is available.

8.2.1 Server Authorization Reference

Data Types Reference

```
typedef pj_status_t pjsip_auth_lookup_cred(pj_pool_t *pool,
                                           const pj_str_t *realm,
```

```
const pj_str_t *acc_name,
pjsip_cred_info *cred_info );
```

Type of function to be registered to authorization server to lookup for credential information for the specified *acc_name* in the specified *realm*. When credential information is successfully retrieved, the function must fill in the *cred_info* with the credentials and return PJ_SUCCESS. Otherwise it should return one of the following error code:

- PJSIP_EAUTHACCNOTFOUND: account not found for the specified realm,
- PJSIP_EAUTHACCDISABLED: account was found but disabled,

Functions Reference

```
pj_status_t pjsip_auth_srv_init( pj_pool_t *pool,
                                 pjsip_auth_srv *auth_srv,
                                 const pj_str_t *realm,
                                 pjsip_auth_lookup_cred *lookup_func,
                                 unsigned options );
```

Initialize server authorization session data structure to serve the specified *realm* and to use *lookup_func* function to look for the credential info. The argument *options* is bitmask combination of the following values:

- PJSIP_AUTH_SRV_IS_PROXY: to specify that the server will authorize clients as a proxy server (instead of as UAS), which means that Proxy-Authenticate will be used instead of WWW-Authenticate.

```
pj_status_t pjsip_auth_srv_verify(   pjsip_auth_srv *auth_srv,
                                     pjsip_rx_data *rdata,
                                     int *status_code );
```

Request the authorization server framework to verify the authorization information in the specified request in *rdata*. If *status_code* is not NULL, it will be filled with suitable status for the response (401/407/etc.).

This function will return PJ_SUCCESS if the authorization information found in the request can be accepted, or the following error when authorization failed:

- PJSIP_EAUTHNOAUTH: no authorization header is specified in the request.
- PJSIP_EINVALIDAUTHSCHEME: invalid/unsupported authorization scheme (only digest is supported at present).
- PJSIP_EAUTHACCNOTFOUND or PJSIP_EAUTHACCDISABLED are the error codes returned by the lookup function.
- PJSIP_EAUTHINVALIDDIGEST: invalid digest,
- other non-zero values may be returned to indicate system error.

```
pj_status_t pjsip_auth_srv_challenge(   pjsip_auth_srv *auth_srv,
                                        const pj_str_t *qop,
                                        const pj_str_t *nonce,
                                        const pj_str_t *opaque,
                                        pj_bool_t stale,
                                        pjsip_tx_data *tdata);
```

Add authentication challenge headers to the outgoing response in *tdata*. If *qop* is specified, then it will be put in the challenge. Application may also specify its customized *nonce* and *opaque* for the challenge, or can leave the value to NULL to make the function fills them in with random characters.

8.3 Extending Authentication Framework

The authentication framework can be extended to support authentication framework other than HTTP digest (e.g. PGP, etc.).

TODO.

Chapter 9:Basic User Agent Layer (UA)

9.1 Basic Dialog Concept

The basic UA dialog provides basic facilities for managing SIP dialogs and dialog usages, such as basic dialog state, session counter, Call-ID, From, To, and Contact headers, sequencing of CSeq in transactions, and route-set.

The basic UA dialog is agnostic/skeptical of what kind of sessions it is being used to (e.g. INVITE session, SUBSCRIBE/NOTIFY sessions, REFER/NOTIFY sessions, etc.), and it can be used to establish multiple and different types of sessions simultaneously in a single dialog.

A PJSIP dialog can be considered just as a passive data structure to hold common dialog attributes. You must not confuse dialog with an INVITE session. An INVITE session is a session (also commonly known as *dialog usage*) "inside" a dialog. There can be other sessions/usages in the same dialog; all of them share common dialog properties (although there can only be one INVITE session per dialog).

For more information about dialog-usage concept, please refer to **draft-sparks-sipping-dialogusage-01.txt**. The document identifies two dialog-usages, i.e. invite usage and subscribe usage.

PJSIP dialog does not know the state of its sessions. It doesn't know whether the INVITE session has been established or disconnected. In fact, PJSIP dialog does not even know what kind of sessions are there in the dialog. All it cares is how many active sessions are there in the dialog. The dialog is started with one active session, and when the session counter reaches zero and the last transaction is terminated, the dialog will be destroyed.

It will be the responsibility of each dialog usages to increment and decrement the dialog's session counter.

9.1.1 Dialog Sessions

Dialog sessions in PJSIP dialog framework is just represented with a reference counter. This reference counter is incremented and decremented by dialog usage module everytime it creates/destroys a session in that particular dialog.

Dialog's sessions are created by dialog usages. In one particular dialog, one dialog usage can create more than one sessions (except invite usage, which can only create one invite session in a single dialog).

The exact representation of "session" will be defined by the dialog usage module. As stated previously, the basic dialog only cares about the number of sessions currently active in the dialog.

9.1.2 Dialog Usages

Dialog usages are PJSIP modules that are registered to the dialog to receive dialog's events. Multiple modules can be registered to one dialog, hence the dialog can have multiple usages. Each dialog usage module is responsible to handle a specific session. For example, the subscribe usage module will create a new subscribe session each time it receives new SUBSCRIBE request (and increment dialog's session counter), and decrement the session counter when the subscribe session has terminated.

The processing of dialog usages by a dialog is similar to the processing of modules by endpoint; on each `on_rx_request()` and `on_rx_response()` event, the dialog passes the event to each dialog usages starting from the higher priority module (i.e. the one with lower priority number) until one of the module returns true (i.e. non-zero), which in this case the dialog will stop the distribution of the event further. The `on_tsx_state()` notification will be distributed to all dialog usages. Each dialog usage should filter out the transaction events that don't belong to it.

In its most basic (i.e. low-level) use, the application manages the dialog directly, and it is the only "usage" (or user) of the dialog. In this case, the application is responsible for managing the sessions inside the dialog, which means handling ALL requests and responses and establishing/tearing down sessions manually.

In later chapters, we will learn about high-level APIs that can be used to manage sessions. These high-level APIs are PJSIP modules that are registered to the dialog as dialog usages, and they will handle/react to different types of SIP messages that are specific to each type of sessions (e.g. an invite usage module will handle INVITE, PRACK, CANCEL, ACK, BYE, UPDATE and INFO, a subscribe usage module will handle REFER, SUBSCRIBE, and NOTIFY, etc.). These high level APIs provide high-level callbacks according to the session's specification.

In this chapter however, we'll only lean about basic, low-level dialog usage.

9.1.3 Dialog Set

Each dialog is a member of a dialog set. A dialog set is identified by common local tag (i.e. From tag). Normally a dialog set only has one dialog as a member. The only time when a dialog set contains multiple dialog is when outgoing INVITE forks, which in this case each response message received with different To tag will create a new dialog in the same dialog set.

A dialog set is defined in PJSIP as an opaque type (i.e. `void*`). A dialog structure (pjsip_dialog) has a member called `dlg_set` to identify the dialog set that it belongs. Application can use linked list API to retrieve the siblings of a dialog (in the same dialog set).

For more information about dialog set, please refer to subsequent section 9.1.6Forking.

9.1.4 Client Authentication

A dialog maintains a client authentication session (`pjsip_auth_clt_sess`), to be used to authenticate requests within the dialog against all downstream servers. The basic dialog initializes each outgoing request with appropriate authentication headers, if they are available. However, authentication challenges MUST be handled by dialog usages; e.g. the basic dialog does not automatically retry a request when it is failed with 401/407 response.

9.1.5 Class Diagram

The following diagram illustrates user agent layer and basic dialog framework.

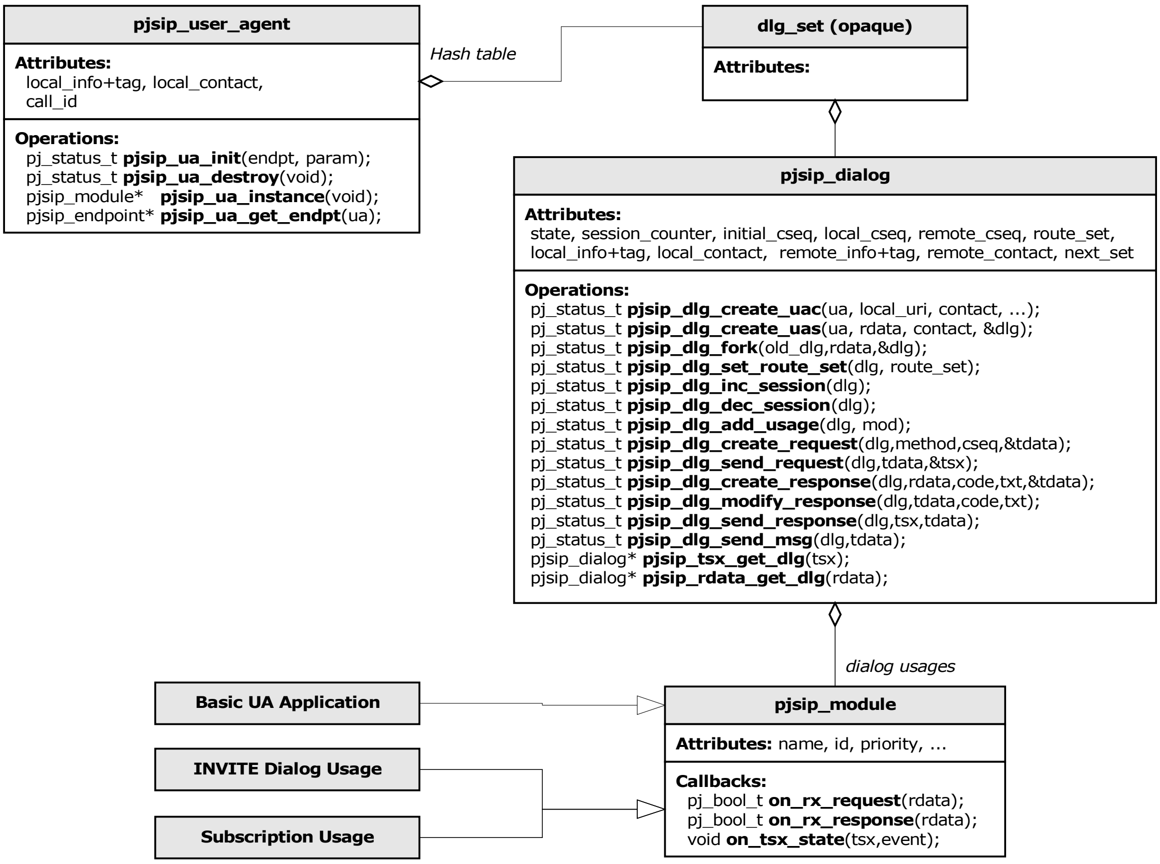

Figure 15 Basic User Agent Class Diagram

The diagram shows the relationship between dialog and its usages. In the most basic/low-level scenario, the application module is the only usage of the dialog. In more high-level scenario, some high-level modules (e.g. `pjsip_invite_usage` and `pjsip_subscribe_usage`) can be registered to a dialog as dialog's usages, and the application will receive events from these usages instead instead of directly from the dialog.

The diagram also shows PJSIP user agent module (`pjsip_user_agent`). The user agent module is the "owner" of all dialogs; the user agent module maintains a hash table of all dialog sets currently active.

9.1.6 Forking

Handling Forking Condition

The user agent module provides a callback that can be registered by application when the user agent detects forked response from the downstream proxy. A forked response in defined as a response (can be provisional or 2xx response) within a dialog that has To tag that is different from any of existing dialogs. When such responses are received, the user agent will call `on_dlg_forked()` callback, passing the received response and the original dialog (the dialog that application created originally) as the arguments.

It is the complete responsibility of the application to handle forking condition!

Upon receiving a forked provisional response, application can:

- ignore the provisional response (perhaps waiting until a final, forked 2xx response is received); or
- create a new dialog (by calling `pjsip_dlg_fork()`). In this case, subsequent responses received from this particular call leg will go to this new dialog.

Upon receiving a forked 2xx response, application can:

- decide to terminate this particular call leg. In this case, the application would construct ACK request from the response, send the ACK, then construct a BYE transaction and send it to the call-leg. Application MUST construct Route headers manually for both ACK and BYE requests according to the Record-Route headers found in the response before sending them to the transaction/transport layer.
- create a dialog for this particular call leg (by calling `pjsip_dlg_fork()`). Application then constructs and sends ACK request to the call leg to establish the dialog. After dialog is established, application may terminate the dialog by sending BYE request.

Application MUST NOT ignore a forked 2xx responses.

Creating Forked Dialog

Application creates a forked dialog by calling `pjsip_dlg_fork()` function. This function creates a dialog and performs the following:

- Copy all attributes of the original dialog (including authorization client session) to the new dialog.
- Assign different remote tag value, according to the tag found in the To header in the response.
- Register the new dialog to user agent's dialog set.
- If the original dialog has an application timer, it will copy the timer and update the timer of the new dialog.

Note that the function WILL NOT copy the dialog usages (i.e. modules) from the original dialog.

> The reason why the function **pjsip_dlg_fork()** doesn't copy the dialog usages from the original dialog is because each usage will normally have dialog specific data that can not be copied without knowing the semantic of the data.

After the new dialog has been created, the application then MUST re-register each dialog usages with the new dialog, by calling `pjsip_dlg_add_usage()`.

The new dialog then MUST be returned as return value of the callback function. This will cause the user agent to dispatch the message to the new dialog, causing dialog usages (e.g. application) to receive `on_rx_response()` notification on the behalf of the new dialog.

Using Timer to Handle Failed Forked Dialog

Application can schedule application specific timer with the dialog by calling `pjsip_dlg_start_app_timer()` function. For timer associated with a dialog, this timer is preferable than general purpose timer because this timer will be automatically deleted when the dialog is destroyed.

Timer is important to handle failed forked dialog. A forked early dialog may not complete with a final response at all, because forking proxy will not forward 300-699 if it receives 2xx response. So the only way to terminate these dangling early dialogs is by setting a timer on these dialogs.

The best way to use dialog's application timer to handle failed forked early dialog, is to start the timer on the other forked dialogs the first time when it receives 2xx response on one of the dialog in the dialog set. When the timer expires and no 2xx response is received, the dialog should be terminated.

9.1.7 CSeq Sequencing

The local cseq of the dialog is updated when the request is sent (as opposed to when the request is created). When CSeq header is present in the request, the value may be updated as the request is sent within the dialog.

The remote cseq of the dialog is updated when a request is received. When dialog's remote cseq is empty, the first request received will set the dialog's remote cseq. For subsequent requests, when dialog receives request with cseq lower than dialog's recorded cseq, this request would be **automatically** answered statelessly by the dialog with a 500 response (Internal Server Error). When the request's cseq is greater than dialog's recorded cseq, the dialog would update the remote's cseq automatically (including when the request's cseq is greater by more than one).

This behavior is compliant with SIP specification RFC 3261 Section 12.2.2.

9.1.8 Transactions

Dialog always acts statefully. It automatically creates UAS transaction when incoming request arrives, and it creates UAC transaction when it is asked to send outgoing request.

The only time when dialog acts statelessly is when it receives incoming request with CSeq lower then current CSeq, which in this case it would answer the request with 500 (Internal Server Error) response.

When a transaction is created on behalf of a dialog (via dialog API, for both UAS and UAC transactions), the transaction user (TU) of the transaction is set to user agent instance, and the dialog instance will be put in the transaction's `mod_data` in the appropriate index. The index is the user agent's module ID. When events or message arrives, the transaction reports the events to user agent module, which will lookup the dialog and pass the event to the dialog.

9.2 Basic UA API Reference

9.2.1 User Agent Module API

typedef pjsip_module pjsip_user_agent;

User agent type.

pj_status_t pjsip_ua_init(pjsip_endpoint *endpt,
const pjsip_ua_init_param *prm);
pjsip_user_agent* pjsip_ua_instance(void);

Get the instance of the user agent.

pj_status_t pjsip_ua_destroy(void);

Destroy the user agent module.

9.2.2 Dialog Structure

The dialog structure and its API are declared in <pjsip/sip_dialog.h>. The following code shows the declaration of **pjsip_dialog**.

```
// This structure is used to describe dialog's participants, local and remote party.
struct pjsip_dlg_party
{
    pjsip_fromto_hdr          *info;         // From/To header, inc tag
    pj_uint32_t                tag_hval;     // Hashed value of the tag
    pjsip_contact_hdr         *contact;      // Contact header.
    pj_int32_t                 first_cseq;   // First CSeq seen.
    pj_int32_t                 cseq;         // Next sequence number.
};

// This structure describes basic dialog.
struct pjsip_dialog
{
    PJ_DECL_LIST_MEMBER(pjsip_dialog);       // List node in dialog set.

    // Static properties:
    char                       obj_name[PJ_MAX_OBJ_NAME];   // Log identification
    pj_pool_t                 *pool;                        // Dialog's memory pool.
    pj_mutex_t                *mutex;                       // Dialog's mutex.
    pjsip_user_agent          *ua;                          // User agent instance.
    void                      *dlg_set;                     // The dialog set.

    // Dialog session properties.
    pjsip_uri                 *target;       // Current target.
    pjsip_dlg_party            local;        // Local party info.
    pjsip_dlg_party            remote;       // Remote party info.
    pjsip_role_e               role;         // Initial role.
    pj_bool_t                  secure;       // Use secure transport?
    pjsip_cid_hdr             *call_id;      // Call-ID header.
    pjsip_route_hdr            route_set;    // Route set list.
    pjsip_auth_clt_sess        auth_sess;    // Client authentication session.

    // Session Management
    int                        sess_count;   // Session counter.
    int                        tsx_count;    // Active transaction counter.

    // Dialog usages
    unsigned                   usage_cnt;               // Number of registered usages.
    pjsip_module              *usage[PJSIP_MAX_MODULE]; // Usages, priority sorted

    // Module specific data.
    void                      *mod_data[PJSIP_MAX_MODULE];
};
```

Code 34 Dialog Structure

9.2.3 Dialog Creation API

A dialog can be created by calling one of the following functions.

```
pj_status_t pjsip_dlg_create_uac(    pjsip_user_agent *ua,
                                     const pj_str_t *local_uri,
                                     const pj_str_t *local_contact_uri,
                                     const pj_str_t *remote_uri,
                                     const pj_str_t *target,
                                     pjsip_dialog **p_dlg);
```

Create a new dialog and return the instance in *p_dlg* parameter. After creating the dialog, application can add modules as dialog usages by calling **pjsip_dlg_add_usage()**.

Note that initially, the session count in the dialog will be initialized to zero.

```
pj_status_t pjsip_dlg_create_uas(    pjsip_user_agent *ua,
                                     pjsip_rx_data *rdata,
                                     const pj_str_t *contact,
                                     pjsip_dialog **p_dlg);
```

Initialize UAS dialog from the information found in the incoming request that creates a dialog (such as INVITE, REFER, or SUBSCRIBE), and set the local Contact to *contact*. If *contact* is not specified, the local contact is initialized from the URI in the To header in the request.

If the request has To tag parameter, dialog's local tag will be initialized from this value. Otherwise a globally unique id generator will be invoked to create dialog's local tag.

This function also initializes the dialog's route set based on the Record-Route header in the request, if present.

Note that initially, the session count in the dialog will be initialized to zero.

```
pj_status_t pjsip_dlg_fork(   pjsip_dialog *original_dlg,
                              pjsip_rx_data *rdata,
                              pjsip_dialog **new_dlg );
```

Create a new (forked) dialog on receipt on forked response in *rdata*. This function clones a new dialog from *original_dlg* (including authentication session), but the new dialog will have new remote tag as copied from the To header in the response. Upon return, the *new_dlg* will have been registered to the user agent. Applications just need to add modules as dialog's usages.

Note that initially, the session count in the dialog will be initialized to zero.

9.2.4 Dialog Session Management API

The following functions are used to manage dialog's session counter.

```
pj_status_t pjsip_dlg_inc_session(   pjsip_dialog *dlg );
```

Increment the number of sessions in the dialog. Note that initially (after created) the dialog already has the session counter set to one.

```
pj_status_t pjsip_dlg_dec_session(   pjsip_dialog *dlg );
```

Decrement the number of sessions in the dialog. Once the session counter reach zero and there is no pending transaction, the dialog will be destroyed. Note that this function may destroy the dialog immediately if there is no pending transaction when this function is called.

9.2.5 Dialog Usages API

The following functions are used to manage dialog usages in a dialog.

```
pj_status_t pjsip_dlg_add_usage( pjsip_dialog *dlg,
                                 pjsip_module *module,
                                 void *mod_data );
```

Add a module as dialog usage, and optionally set the module specific data.

```
pj_status_t pjsip_dlg_set_mod_data( pjsip_dialog *dlg,
                                    int module_id,
                                    void *data );
```

Attach module specific data to the dialog.

```
void* pjsip_dlg_get_mod_data( pjsip_dialog *dlg,
                              int module_id);
```

Get module specific data previously attached to the dialog. Application can also get value directly by accessing *dlg->mod_data[module_id]*.

9.2.6 Dialog Request and Response API

```
pj_status_t pjsip_dlg_create_request(   pjsip_dialog *dlg,
                                        const pjsip_method *method,
                                        int cseq,
                                        pjsip_tx_data **tdata);
```

Create a basic/generic request with the specified *method* and optionally specify the *cseq*. Use value –1 for *cseq* to have the dialog automatically put next cseq number for the request. Otherwise for some requests, e.q. CANCEL and ACK, application must put the CSeq in the original INVITE request as the parameter. This function will also put Contact header where appropriate.

```
pj_status_t pjsip_dlg_send_request ( pjsip_dialog *dlg,
                                     pjsip_tx_data *tdata,
                                     pjsip_transaction **p_tsx );
```

Send request message to remote peer. If the request is not an ACK request, the dialog will send the request statefully, by creating an UAC transaction and send the request with the transaction. Also when the request is not ACK or CANCEL, the dialog will increment its local cseq number and update the cseq in the request according to dialog's cseq.

Note that *on_tsx_state* callback of the dialog usages may be called before this function returns.

If *p_tsx* is not null, this argument will be set with the transaction instance that was used to send the request.

This function decrements the transmit data's reference counter regardless the status of the operation.

```
pj_status_t pjsip_dlg_create_response(  pjsip_dialog *dlg,
                                        pjsip_rx_data *rdata,
                                        int st_code,
                                        const pj_str_t *st_text,
                                        pjsip_tx_data **tdata);
```

Create a response message for the incoming request in *rdata* with status code *st_code* and optional status text *st_text*. This function is different than endpoint's API `pjsip_endpt_create_response()` in that the dialog function adds Contact header and Record-Route headers in the response where appropriate.

```
pj_status_t pjsip_dlg_modify_response(  pjsip_dialog *dlg,
                                        pjsip_tx_data *tdata,
                                        int st_code,
                                        const pj_str_t *st_text);
```

Modify previously sent response with other status code. Contact header will be added when appropriate.

```
pj_status_t pjsip_dlg_send_response( pjsip_dialog *dlg,
                                     pjsip_transaction *tsx,
                                     pjsip_tx_data *tdata);
```

Send response message statefully. The transaction instance MUST be the transaction that was reported on `on_rx_request()` callback.

This function decrements the transmit data's reference counter regardless the status of the operation.

9.2.7 Dialog Auxiliary API

```
pj_status_t pjsip_dlg_set_route_set( pjsip_dialog *dlg,
                                     const pjsip_route_hdr *route_set );
```

Set dialog's initial route set to *route_set* list. This can only be called for UAC dialog, before any request is sent. After dialog has been established, the route set can not be changed.

For UAS dialog,the route set will be initialized in `pjsip_dlg_create_uas()` from the Record-Route headers in the incoming request.

The *route_set* argument is standard list of Route headers (i.e. with sentinel).

```
pj_status_t pjsip_dlg_start_app_timer(  pjsip_dialog *dlg,
                                        int app_id,
                                        const pj_time_val *interval,
                                        void (*cb)(pjsip_dialog*,int));
```

Start application timer with this dialog with application specific id in *app_id* and callback to be called in *cb*. Application can only set one application timer per dialog. This timer is more usefull for dialog specific timer, because it will be automatically destroyed once the dialog is destroyed. Note that timer will also be copied to the forked dialog.

```
pj_status_t pjsip_dlg_stop_app_timer(   pjsip_dialog *dlg );
```

Stop application specific timer if exists.

```
pjsip_dialog* pjsip_rdata_get_dlg(   pjsip_rx_data *rdata );
```

Get the dialog instance in the incoming *rdata*. If an incoming message matches an existing dialog, the user agent must have put the matching dialog instance in the *rdata*, or otherwise this function will return NULL if the message didn't match any existing dialog.

```
pjsip_dialog* pjsip_tsx_get_dlg( pjsip_transaction *tsx );
```

Get the dialog instance in the specified transaction.

9.3 Examples

9.3.1 Invite UAS Dialog

The following examples uses basic/low-level dialog API to process an incoming dialog. The examples show how to:

- create and initialize incoming dialog,
- create UAS transaction to process the incoming INVITE request and transmit 1xx responses,
- transmit 2xx response to INVITE reliably,
- process the incoming ACK.

As usual, most error handlings are omited for brevity. Real-world application should be prepare to handle error conditions in all stages of the processing.

Creating Initial Invite Dialog

In this example we'll learn how to create a dialog for an incoming INVITE request and respond the dialog with 180/Ringing provisional response.

```
pj_bool_t on_rx_request(pjsip_rx_data *rdata)
{
   if (rdata->msg->line.request.method.id == PJSIP_INVITE_METHOD &&
       pjsip_rdata_get_dlg(rdata) == NULL)
   {
      // Process incoming INVITE!
      pjsip_dialog *dlg;
      pjsip_transaction *tsx;
      pjsip_tx_data *tdata;
      struct app_dialog *app_dlg;

      // Create, initialize, and register new dialog for incoming INVITE.
      // This also implicitly create UAS transaction for rdata.
      status = pjsip_dlg_create_uas( pjsip_ua_instance(), rdata, NULL, &dlg);

      // Register application as the only dialog usage
      status = pjsip_dlg_add_usage( dlg, &app_module, NULL );

      // Increment session.
      pjsip_dlg_inc_session(dlg);

      // Create 180/Ringing response
      status = pjsip_dlg_create_response( dlg, rdata, 180, NULL /*Ringing*/, &tdata);

      // Send 180 response statefully. A transaction will be created in &tsx.
      status = pjsip_dlg_send_response( dlg, pjsip_rdata_get_tsx(rdata), tdata);

      // As in real application, normally we will send 200/OK later,
      // when the user press the "Answer" button. In this example, we'll send
      // 200/OK in answer_dlg() function which will be explained later. In order
      // to do so, we must "save" the INVITE transaction. We do this by putting
      // the transaction instance in dialog's module data at index application
      // module's ID.
      //
      dlg->mod_data[app_module.id] = pjsip_rdata_get_tsx(rdata);

      // Done processing INVITE request
      return PJ_TRUE;
   }
   // Process other requests
   ...
}
```

Code 35 Creating Dialog for Incoming Invite

Answering Dialog

In this example we will learn how to send 200/OK response to establish the dialog.

```
static void answer_dlg(pjsip_dlg *dlg)
{
   pjsip_transaction *invite_tsx;
   pjsip_tx_data *tdata;

   invite_tsx = dlg->mod_data[app_module.id];

   // Modify previously sent (provisional) response to 200/OK response.
   // The previously sent message is found in tsx->last_tx.
   tdata = invite_tsx->last_tx;
   status = pjsip_dlg_modify_response( dlg, tdata, 200, NULL /*OK*/ );

   // You may modify the response before it's sent
   // (e.g. add msg body etc).
   ...

   // Send the 200 response using previous transaction.
   // Transaction will take care of the retransmission.
   status = pjsip_dlg_send_response( dlg, invite_tsx, tdata);

   // We don't need to keep pending invite tsx anymore.
   dlg->mod_data[app_module.id] = NULL;
}
```

Code 36 Answering Dialog

Processing CANCEL Request

In this example we will learn how to handle incoming CANCEL request.

```
pj_bool_t on_rx_request(pjsip_rx_data *rdata)
{
   ...
   if (rdata->msg->line.request.method.id == PJSIP_CANCEL_METHOD)
   {
      // See if we have pending INVITE transaction.
      pjsip_dialog *dlg;
      pjsip_transaction *invite_tsx;

      // All requests within a dialog will have the dialog instance
      // recorded in rdata.
      dlg = pjsip_rdata_get_dlg(rdata);
      if (!dlg) {
         // Not associated with any dialog. Respond statelessly with 481.
         status = pjsip_endpt_respond_stateless( endpt, rdata, 481, NULL, NULL,
                                                 NULL, NULL);
         return PJ_TRUE;
      }

      invite_tsx = dlg->mod_data[app_module.id];

      if (invite_tsx) {
         pjsip_tx_data *tdata;

         // Transaction found. Respond CANCEL (statefully!) with 200 regardless
         // whether the INVITE transaction has completed or not.
         status = pjsip_dlg_respond( dlg, rdata, 200, NULL /*OK*/);

         // Respond the INVITE transaction with 487/Request Terminated
         // only when INVITE transaction has not send final response.
         if (invite_tsx->status_code < 200) {
            tdata = invite_tsx->last_tx;
```

```
            status = pjsip_dlg_modify_response( dlg, tdata, 487, NULL  );

            // Send the 487 response.
            status = pjsip_dlg_send_response( dlg, invite_tsx, tdata);

            dlg->mod_data[app_module.id] = NULL;

            // Decrement session!
            pjsip_dlg_dec_session(dlg);
         }

      } else {
         // Transaction not found, respond CANCEL with 481 (statefully!)
         status = pjsip_dlg_respond ( dlg, rdata, 481, NULL );
      }

      // Done processing CANCEL request
      return PJ_TRUE;
   }

   // Process other requests
   ...
}
```

Code 37 Processing CANCEL Request

Processing ACK Request

In this example we will learn how to handle incoming ACK request.

```
pj_bool_t on_rx_request(pjsip_rx_data *rdata)
{
   ...
   if (rdata->msg->line.request.method.id == PJSIP_ACK_METHOD &&
       pjsip_rdata_get_dlg(rdata) != NULL)
   {
      // Process the ACK request
      pjsip_dialog *dlg = pjsip_rdata_get_dlg(rdata);
      ...

      return PJ_TRUE;
   }
   ...
}
```

Code 38 Processing ACK Request

9.3.2 Outgoing Invite Dialog

The following sets of example demonstrate how to work with outgoing INVITE dialog.

Creating Initial Dialog

```
static pj_status_t make_call( const pj_str_t *local_info, const pj_str_t *remote_info)
{
   pjsip_dialog *dlg;
   pjsip_tx_data *tdata;

   // Create and initialize dialog.
   status = pjsip_dlg_create_uac( user_agent, local_info, local_info,
                                  remote_info, remote_info, &dlg );

   // Register application as the only dialog usage.
   status = pjsip_dlg_add_usage( dlg, &app_module, NULL);
```

```
   // Add session.
   pjsip_dlg_inc_session(dlg);

   // Send initial INVITE.
   status = pjsip_dlg_create_request( dlg, &pjsip_invite_method, -1, &tdata);

   // Modify the INVITE (e.g. add message body etc.. )
   ...

   // Send the INVITE request.
   status = pjsip_dlg_send_request( dlg, tdata, NULL);

   // Done.
   // Further responses will be received in on_rx_response.
   return status;
}
```

Code 39 Creating Outgoing Dialog

Receiving Response

```
static pj_bool_t on_rx_response( pjsip_rx_data *rdata )
{
   pjsip_dialog *dlg;
   dlg = pjsip_rdata_get_dlg( rdata );

   if (dlg != NULL ) {
      pjsip_transaction *tsx = pjsip_rdata_get_tsx( rdata );
      if ( tsx != NULL && tsx->method.id == PJSIP_INVITE_METHOD) {
         if (tsx->status_code < 200) {
            PJ_LOG(3,("app", "Received provisional response %d", tsx->status_code));
         } else if (tsx->status_code >= 300) {
            PJ_LOG(3,("app", "Dialog failed with status %d", tsx->status_code));
            pjsip_dlg_dec_session(dlg);
            // ACK for non-2xx final response is sent by transaction.
         } else {
            PJ_LOG(3,("app", "Received OK response %d!", tsx->status_code));
            send_ack( dlg, rdata );
         }
      }
      else if (tsx == NULL && rdata->msg_info.cseq->method.id == PJSIP_INVITE_METHOD
               && rdata->msg_info.msg->line.status.code/100 == 2)
      {
         // Process 200/OK response retransmission.
         send_ack( dlg, rdata );
      }
      return PJ_TRUE;
   }
   else
      // Process other responses not belonging to any dialog
      ...
}
```

Code 40 Receiving Response in Dialog

Sending ACK

```
static void send_ack( pjsip_dialog *dlg, pjsip_rx_data *rdata )
{
   pjsip_tx_data *tdata;

   // Create ACK request
   status = pjsip_dlg_create_request( dlg, &pjsip_ack_method,
                                      rdata->msg_info.cseq->cseq, &tdata );

   // Add message body
```

```
   ...

   // Send the request.
   status = pjsip_dlg_send_request ( dlg, tdata, NULL );
}
```

Code 41 Sending ACK Request

9.3.3 Terminating Dialog

The following sample shows one way to terminate INVITE dialog, e.g. by sending BYE.

```
static void send_bye( pjsip_dialog *dlg  )
{
   pjsip_tx_data *tdata;

   // Create BYE request
   status = pjsip_dlg_create_request( dlg, &pjsip_bye_method, -1, &tdata );

   // Send the request.
   status = pjsip_dlg_send_request ( dlg, tdata, NULL );

   // Decrement session.
   // Dialog will be destroyed once the BYE transaction terminates.
   pjsip_dlg_dec_session(dlg);
}
```

Chapter 10:SDP Offer/Answer Framework

The SDP offer/answer framework in PJSIP is based on RFC 3264 "An Offer/Answer Model with the Session Descriptor Protocol (SDP)". The main function of the framework is to facilitate the negotiating of media capabilities between local and remote parties, and to get agreement on which set of media to be used in one invite session.

Note that although it is mainly used by invite session, the framework itself is based on a generic SDP negotiation framework (**`pjmedia_sdp_neg`**), so it should be able to be used by other types of applications. The dialog invite session provides integration of SDP offer/answer framework with SIP protocol; it correctly interpret the message bodies in relevant messages (e.g. INVITE, ACK, PRACK, UPDATE) and translates them to SDP offer/answer negotiation.

This chapter describes the low level SDP negotiator framework, which is declared in **`<pjmedia/sdp_neg.h>`** header file.

10.1 SDP Negotiator Structure

The **`pjmedia_sdp_neg`** structure represents generic SDP offer/answer session, and is used to negotiate local's and remote's SDP.

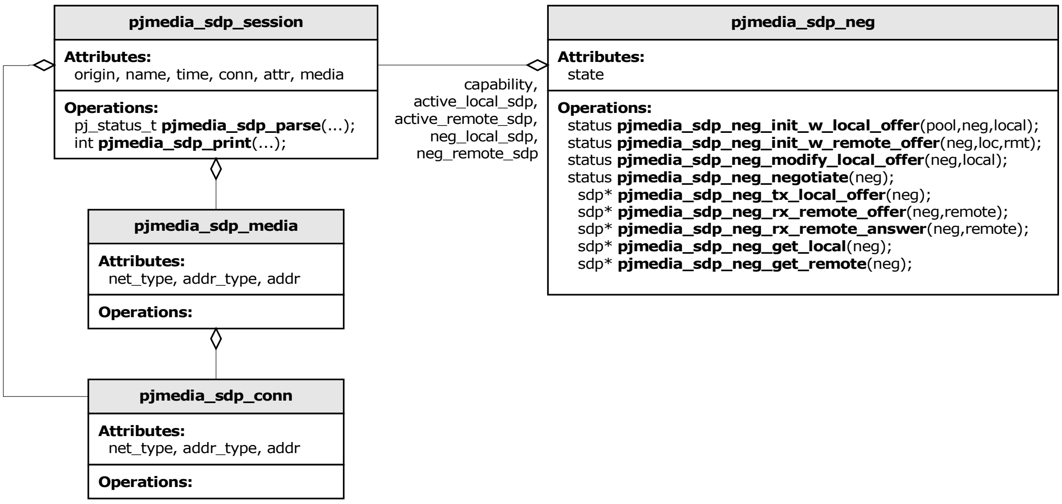

Figure 16 SDP Negotiator "Class Diagram"

The **`pjmedia_sdp_neg`** structure keeps three SDP structures:

- **`initial_sdp`**: which is the initial capability of local endpoint. This SDP is passed to the negotiator during creation, and the contents generally will not be changed throughout the session (even after negotiation). The negotiator uses this SDP in the negotiation when it receives new offer from remote (as opposed to receiving updated SDP from remote).
- **`active_local_sdp`**: contains local SDP after it has been negotiated with remote. The dialog MUST use this to start its local media instead of the initial SDP.
- **`active_remote_sdp`**: contains the SDP currently used by peer/remote.

The negotiator also has two other SDP variables which are only used internally during negotiation process, namely **`neg_local_sdp`** and **`neg_remote_sdp`**. These are temporary SDP description, and application MUST NOT refer to these variables.

10.2 SDP Negotiator Session

The general state transition of SDP offer/answer session is shown in the following diagram.

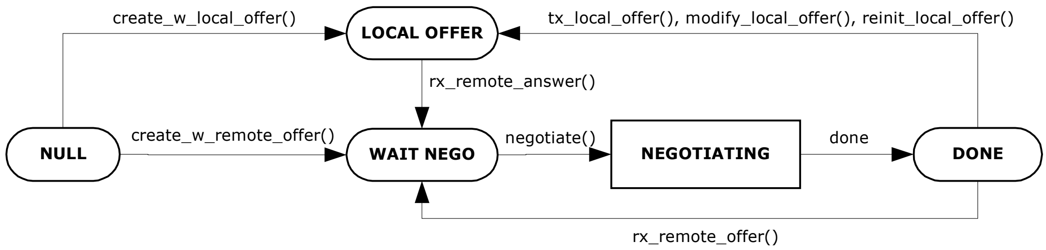

Figure 17 SDP Offer/Answer Session State Diagram

The negotiation session starts with `PJMEDIA_SDP_NEG_STATE_NULL`. If the dialog has a local media description ready and want to offer the media to remote (normally this is the case when the dialog is acting as UAC), it creates the SDP negotiator by passing the local SDP to the function **`pjmedia_sdp_neg_create_w_local_offer()`**. This function will set the initial capability of local endpoint, and set the negotiation session state to `PJMEDIA_SDP_NEG_STATE_LOCAL_OFFER`. The initial SDP then can be sent to remote party in the outgoing INVITE request. Once dialog has received remote's SDP, it must call **`pjmedia_sdp_neg_rx_remote_answer()`** with providing the remote's SDP. The negotiation function can then be called.

If the dialog already has remote media description in hand (normally this is the case when dialog is acting as UAS), it can create the SDP negotiator session by passing both local and remote SDP to **`pjmedia_sdp_neg_create_w_remote_offer()`**. After this, the negotiation function can be called.

After the session has been established, both local and remote party may modify the session. The negotiator can handle one of these two situations:

- The dialog has received SDP from remote. In this case, the dialog will call **`pjmedia_sdp_neg_rx_remote_offer()`** and passing the remote's SDP to this function. After this the negotiation function can be called. The negotiation function's return value determines whether there is modification needed in the local media.
- The local party wants to send SDP to remote. Dialog can further choose one of the following actions:
    - If it just wants to send currently active local SDP without modification, it should call **`pjmedia_sdp_neg_tx_local_offer()`** to get the active local SDP, send the SDP, then wait for the remote's answer.
    - If it wants to modify currently active local media (e.g. changing stream direction, change active codec, etc), it should get the active local media with **`pjmedia_sdp_neg_get_local()`**, modify it, call **`pjmedia_sdp_neg_modify_local_offer()`** to update the offer, send the local SDP, then wait for the remote's answer.

- The dialog may want to completely change the local media (e.g. changing IP address, changing codec set, adding new media line). This is different than updating current media described above because it will change **`initial_sdp`**, so that future negotiation will be based on this new SDP. If the dialog wants to do this, it calls **`pjmedia_sdp_neg_reinit_local_offer()`** with the new local SDP, send the SDP, then wait for remote's answer.

After the dialog has sent offer to remote party, it should receive answer back from the remote party. The dialog must provide the remote's SDP to the negotiator so that the negotiation function can be called. The dialog provides the remote's answer by calling **`pjsip_sdp_neg_rx_remote_answer()`**.

If remote has rejected local's offer (e.g. returning 488/"Not Acceptable Here" response), dialog MUST still call **`pjsip_sdp_neg_rx_remote_answer()`** with providing NULL in remote's SDP argument, and call the negotiation function so that the negotiator session can revert back to previously active session descriptions, if any.

10.3 SDP Negotiation Function

The dialog calls **`pjmedia_sdp_neg_negotiate()`** to negotiate the offer and the answer, after it has provided both local's and remote's SDP to be used for the negotiation (i.e. negotiator state is `PJMEDIA_SDP_NEG_STATE_WAIT_NEGO`). This function may return one of the following result:

- **`PJ_SUCCESS`**, (i.e. zero) if it has successfully established an agreement between local and remote SDP. In this case, both local's and remote's *active* SDP will be stored in the session for future reference, and application can query these active SDPs to start the local media.
- **`PJMEDIA_ESDPNOCHANGE`**, if it found out that there is no modification needed in currently used SDPs (both local and remote). In this case, the previously agreed SDP sessions will not be modified either.
- **`PJMEDIA_ESDPFAIL`**, if it couldn't find agreement on local and remote capabilities. In this case, if the session is keeping a previously agreed SDP, these SDP (local and remote) will not be modified. If dialog is acting as UAS for this session, it should respond the request with 488/Not Acceptable Here response to the offer.
- **`PJMEDIA_ESDPNOOFFER`**, if negotiator has not sent/received any offer yet.
- **`PJMEDIA_ESDPNOANSWER`**, if negotiator has not received remote's answer yet.
- or other non-zero value to indicate other errors.

In all cases, the negotiation function will set the negotiator's state to `PJMEDIA_SDP_NEG_STATE_DONE`.

Chapter 11:Dialog Invite Session and Usage

11.1 Introduction

The dialog invite session is a high level invite session management, which can be used by application to manage invite session (including SDP management). The invite session is designed to completely abstract the basic dialog, so application should not need to use basic dialog API when it is using the invite session API.

A dialog invite session can be created by application on per dialog basis. The dialog invite session is managed by dialog invite usage, which is a PJSIP module. The dialog invite usage performs dispatching of events from the dialog to the corresponding invite session, and also handles forked dialog.

The dialog invite session and usage is implemented in a separate static library, i.e. **pjsip-ua** library. Application MUST include **<pjsip-ua/sip_inv.h>** to use the dialog invite session/usage functionality. Alternatively, applications can include a single header file **<pjsip-ua.h>** to get all the functionalities in **pjsip-ua** library.

11.1.1 Terms

Dialog invite session is an invite session inside a dialog. If application decides to use the high level invite session management, it needs to create one and only one instance of dialog invite session for each dialog.

Dialog invite usage is a PJSIP module, registered to PJSIP endpoint. When a dialog has dialog invite session, this module needs to be registered to the particular dialog as the dialog usage. This will be achieved automatically during invite session creation.

11.1.2 Features

The dialog invite session provides the following features for the application:

- Session progress reporting (e.g. session progressing, connected, confirmed, disconnected),
- Automatic authentication handling (e.g. retry the request on receipt of 401/407 response),
- SDP offer and answer handling,
- High-level forking handler,
- Session timeout (i.e. Expires header),
- Session extensions, such as session timer, and reliable provisional response.

11.1.3 Invite Session State

The dialog invite usage provides callback to notify application about session progress. This is particularly usefull for telephony applications, where the session's state is normally associated with telephony call state.

The progress of an invite session is shown in the following state transition diagram.

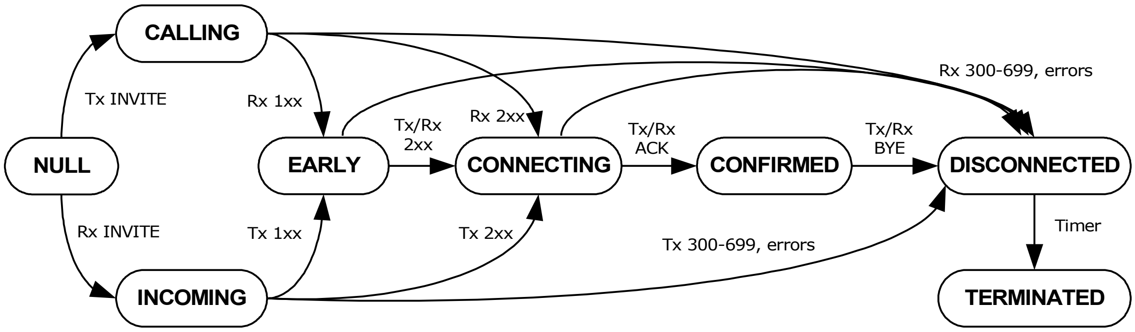

Figure 18 Invite Session State Diagram

The description of each state is as follows:

`PJSIP_INV_STATE_NULL`	This is the state of the session when it was first created. No messages have been sent/received at this point.
`PJSIP_INV_STATE_CALLING`	The session state after the first INVITE message is sent, but before any provisional response is received.
`PJSIP_INV_STATE_INCOMING`	The session state after the first INVITE message is received, but before any provisional response is sent.
`PJSIP_INV_STATE_EARLY`	The session state after dialog has sent or received provisional response messages for the INVITE request, only when To tag is present.
`PJSIP_INV_STATE_CONNECTING`	The session state after a final 2xx response has been sent or received.
`PJSIP_INV_STATE_CONFIRMED`	The session state after ACK request has been sent or received.
`PJSIP_INV_STATE_DISCONNECTED`	The session state when the session has been disconnected, either because of non-successful final response to INVITE or BYE request.
`PJSIP_INV_STATE_TERMINATED`	The session state when the session will be destroyed. All resources will be freed when this state is reached.

Figure 19 Invite Session State Description

11.1.4 Invite Session Creation

For outgoing dialog (i.e. caller), application needs to create an UAC dialog with `pjsip_dlg_create_uac()`. Application then creates the invite session for the dialog by calling `pjsip_inv_create_uac()`, passing the UAC dialog instance as one of the parameter. Application MUST NOT send the INVITE request before invite sesion has been created, or otherwise the invite session will loose some events.

For incoming dialog, application can first verify if the request can be accepted by calling `pjsip_inv_verify_request()`. This function verifies the Supported, Require, and the request body to make sure that it can accept the request. Should the request can not be accepted, it will create the appropriate rejection response. If the request can be accepted, the application creates the UAS dialog by calling `pjsip_dlg_create_uas()` function. Application then creates the invite session for this dialog by calling `pjsip_inv_create_uas()`, passing the UAS dialog instance as one of the parameter. Application MUST NOT send any responses before the invite sesion has been created, or otherwise the invite session will loose some events.

When an outgoing dialog forked, and if an invite session exists in the "original" dialog, the invite usage will module automatically create the invite session for the new (forked) dialog. Application will be notified about the creation of the new session via a callback.

The invite session creation functions (i.e. `pjsip_inv_create_uac()` and `pjsip_inv_create_uas()` functions) automatically registers the invite session usage to the dialog. Application does not need to call `pjsip_dlg_add_usage()` to register the invite usage module to the dialog.

11.1.5 Messages Handling

The invite session handles all SIP methods that may change the state of the invite session. For this version of PJSIP, the invite session handles **INVITE**, **BYE**, **ACK**, **CANCEL**, **UPDATE**, and **PRACK** methods.

Application MUST use invite session API to create and send request and response messages <u>with above methods</u>. This is necessary to ensure that the request and response message is handled correctly and it contains the appropriate features being used by the session (e.g. reliable provisional response).

Application can still use the base dialog API to create and send request and response message for methods other than above. For example, application can use the base dialog API to create and send **MESSAGE** request inside the dialog.

11.1.6 Extending the Dialog

As stated previously, the invite session handles **INVITE**, **BYE**, **ACK**, **CANCEL**, **UPDATE**, and **PRACK** messages that occurs in a dialog. When application wants to support or handle other types of messages, it must register itself to the dialog as dialog usage. This will enable the application to process incoming requests that are "unhandled" by existing dialog's uages.

It is important that application set its application module's priority correctly. Application priority should be set to to PJSIP_MOD_PRIORITY_APPLICATION. The invite usage has module priority set to (PJSIP_MOD_PRIORITY_APPLICATION-1). This would ensure that the invite usage is able to inspect the incoming requests first before application.

11.1.7 Extending the Invite Session

In the future, the invite session may be extended to support more SIP extensions, such as call transfer, dialog targetting, etc. At present, application should be able to perform these features by constructing the messages manually.

11.2 Reference

11.2.1 Data Structure

The invite session functionalities are declared in header file **<pjsip-ua/sip_inv.h>**.

```
enum pjsip_inv_state
{
   PJSIP_INV_STATE_NULL,            // Before INVITE is sent or received.
   PJSIP_INV_STATE_CALLING,         // After INVITE is sent.
   PJSIP_INV_STATE_INCOMING,        // After INVITE is received
   PJSIP_INV_STATE_EARLY,           // After response with To tag is sent/received.
   PJSIP_INV_STATE_CONNECTING,      // After 2xx response is sent/received.
   PJSIP_INV_STATE_CONFIRMED,       // After ACK is sent/received.
   PJSIP_INV_STATE_DISCONNECTED,    // Session is terminated
   PJSIP_INV_STATE_TERMINATED,      // Session will be destroyed.
};

struct pjsip_inv_session
{
   pjsip_inv_state            state;      // Session state.
   pjsip_dialog              *dlg;        // The base dialog.
   pjmedia_sdp_neg           *neg;        // SDP negotiator.
   pj_uint32_t                options;    // Options in use, see pjsip_inv_option
};
```

Code 42 Invite Session Data Structure

The following code shows various options that can be applied to a session. The bitmask combination of these options need to be specified when creating a session. After the dialog is established (including early), the **options** member of **pjsip_inv_session** shows which capabilities are common in both endpoints.

```
enum pjsip_inv_option
{
   PJSIP_INV_SUPPORT_100REL   = 1,  // Indicate support for 100rel extension
   PJSIP_INV_SUPPORT_TIMER    = 2,  // Indicate support for session timer extension.
   PJSIP_INV_SUPPORT_UPDATE   = 4,  // Indicate support for UPDATE method.

   PJSIP_INV_REQUIRE_100REL   = 32, // Require 100rel extension.
   PJSIP_INV_REQUIRE_TIMER    = 64, // Require session timer extension.
};
```

Code 43 Invite Session Options

11.2.2 Invite Usage Module

The invite usage module MUST be initialized before any invite session can be created.

```
pj_status_t pjsip_inv_usage_init(    pjsip_endpoint *endpt,
                                     pjsip_module *app_module,
                                     const pjsip_inv_callback *callback);
```

Initialize the invite usage module and register it to the endpoint. The *callback* argument contains pointer to functions to be called on occurences of events in invite sessions.

```
pjsip_module* pjsip_inv_usage_instance(void);
```

Get the instance of the invite usage module.

11.2.3 Session Callback

The structure **`pjsip_inv_callback`** contains pointer to functions will can be registered by application to invite usage module to receive notification about invite session events.

The functions in the callback are as follows.

```
void on_state_changed(  pjsip_inv_session *inv_ses,
                        pjsip_event *e);
```

This callback is called when the invite sesion state has changed. Application should inspect the session state (**`inv_sess->state`**) to get the current state.

This callback is mandatory.

```
void on_new_session(    pjsip_inv_session *inv_ses,
                        pjsip_event *e);
```

This callback is called when the invite usage module has created a new dialog and invite because of forked outgoing request.

This callback is mandatory.

```
void on_tsx_state_changed( pjsip_inv_session *inv_ses,
                           pjsip_transaction *tsx,
                           pjsip_event *e );
```

This callback is called whenever any transactions within the session has changed their state. Application MAY implement this callback, e.g. to monitor the progress of an outgoing request.

This callback is optional.

```
void on_rx_offer(   pjsip_inv_session *inv_ses );
```

This callback is called when the invite session has received new offer from peer. Application can inspect the remote offer by calling negotiator's *pjmedia_sdp_neg_get_neg_remote()*, and set local answer by calling *pjmedia_sdp_neg_set_local_answer()*.

This callback is optional. When it's not specified, the default behavior is to negotiate remote offer with session's initial capability.

```
void on_media_update(  pjsip_inv_session *inv_ses, pj_status_t status );
```

This callback is called after SDP offer/answer session has completed. The status argument specifies the status of the offer/answer, as returned by *pjmedia_sdp_neg_negotiate()*.

This callback is optional (from the point of view of the framework), but all useful applications normally need to implement this callback.

11.2.4 Session Creation

```
pj_status_t pjsip_inv_create_uac(   pjsip_dialog *dlg,
                                    const pjmedia_sdp_session *local_sdp,
                                    unsigned options,
                                    pjsip_inv_session **inv_sess);
```

Create UAC invite session for the specified dialog in *dlg*. If application has determined its media capability, it can specify the SDP in *local_sdp*. Otherwise it can leave this to NULL, to let remote UAS specifies an offer. The *options* argument is bitmask combination of SIP features in *pjsip_inv_options* enumeration.

On successful return, the invite session will be put in *inv_sess* argument and the function will return PJ_SUCCESS. Otherwise the appropriate error status will be returned on failure.

```
pj_status_t pjsip_inv_verify_request(pjsip_rx_data *rdata,
                                    unsigned *options,
                                    const pjmedia_sdp_session *local_sdp,
                                    pjsip_dialog *dlg,
                                    pjsip_endpoint *endpt,
                                    pjsip_tx_data **tdata);
```

Application SHOULD call this function upon receiving the initial INVITE request in *rdata* before creating the invite session (or even dialog), to verify that the invite session can handle the INVITE request. This function verifies that local endpoint is capable to handle required SIP extensions in the request (i.e. Require header field) and also the media, if media description is present in the request.

Upon calling this function, the *options* argument SHOULD contain the desired SIP extensions to be applied to the session. Upon return, this argument will contain the SIP extension that *will* be applied to the session, after considering the Supported, Require, and Allow headers in the request.

If local media capability has been determined, and if application wishes to verify that it can handle the media offer in the incoming INVITE request, it SHOULD specify its local media capability in *local_sdp* argument. If it is not specified, media verification will not be performed by this function.

If everything has been negotiated successfully, the function will return PJ_SUCCESS. Otherwise it will return the reason of the failure.

This function is capable to create the appropriate response message when the verification has failed. If *tdata* is specified, then a non-2xx final response will be created and put in this argument upon return, when the verification has failed. If a dialog has been created prior to calling this function, then it MUST be specified in *dlg* argument. Otherwise application MUST specify the *endpt* argument (this is useful e.g. when application wants to send the response statelessly).

```
pj_status_t pjsip_inv_create_uas(   pjsip_dialog *dlg,
                                    pjsip_rx_data *rdata,
                                    const pjmedia_sdp_session *local_sdp,
                                    unsigned options,
                                    pjsip_inv_session **inv_sess);
```

Create UAS invite session for the specified dialog in *dlg*. Application MUST specify the received INVITE request in *rdata*. The invite session needs to inspect the received request to see if the request contains features that it supports.

Application SHOULD call the verification function before calling this function, to ensure that it can create the session successfully.

If application has determined its media capability, it can specify this capability in *local_sdp*. If SDP is received in the initial INVITE, the UAS capability specified in *local_sdp* doesn't have to match the received offer; the SDP negotiator is able to rearrange the media lines in the answer so that it matches the offer.

The *options* argument is bitmask combination of SIP features in *pjsip_inv_options* enumeration.

On successful return, the invite session will be put in *inv_sess* argument and the function will return PJ_SUCCESS. Otherwise the appropriate error status will be returned on failure.

11.2.5 Session Operations

```
pj_status_t pjsip_inv_invite( pjsip_inv_session *inv,
                              pjsip_tx_data **tdata );
```

Create the initial INVITE request for this session. This function can only be called for UAC session. The initial INVITE request will be put in *tdata* argument if it can be created successfully.

If local media capability is specified when the invite session was created, then this function will put an SDP offer in the outgoing INVITE request. Otherwise the outgoing request will not contain SDP body.

```
pj_status_t pjsip_inv_answer( pjsip_inv_session *inv,
                              int st_code,
                              const pj_str_t *st_text,
                              const pjmedia_sdp_session *local_sdp,
                              pjsip_tx_data **tdata );
```

Create a response message to the initial INVITE request. The *st_code* contains the status code to be sent, which may be a provisional or final response. If custom status text is desired, application can specify the text in *st_text*; otherwise if this argument is NULL, default status text will be used.

If application has specified its media capability during creation of UAS invite session, the *local_sdp* argument MUST be NULL. This is because application can not perform more than one SDP offer/answer session in a single INVITE transaction.

If application has not specified its media capability during creation of UAS invite session, it MAY or MUST specify its capability in *local_sdp* argument, depending whether *st_code* indicates a 2xx final response.

```
pj_status_t pjsip_inv_end_session(  pjsip_inv_session *inv,
                                    int st_code,
                                    const pj_str_t *st_text,
                                    pjsip_tx_data **tdata );
```

Create a SIP message to initiate invite session termination. Depending on the state of the session, this function may return CANCEL request, a non-2xx final response, or a BYE request. If the session has not answered the incoming INVITE, this function creates the non-2xx final response with the specified status code in *st_code* and optional status text in *st_text*.

```
pj_status_t pjsip_inv_reinvite(  pjsip_inv_session *inv,
```

```
                                  const pj_str_t *new_contact,
                                  const pjmedia_sdp_session *new_offer,
                                  pjsip_tx_data **tdata );
```

Create a re-INVITE request. If application wants to update its local contact and inform peer to perform target refresh with a new contact, it can specify the new contact in *new_contact* argument; otherwise this argument must be NULL.

Application MAY initiate a new SDP offer/answer session in the request when there is no pending answer to be sent or received. It can detect this condition by observing the state of the SDP negotiator of the invite session. If new offer should be sent to remote, the offer must be specified in *new_offer*, otherwise this argument must be NULL.

```
pj_status_t pjsip_inv_update (    pjsip_inv_session *inv,
                                  const pj_str_t *new_contact,
                                  const pjmedia_sdp_session *new_offer,
                                  pjsip_tx_data **tdata );
```

Create an UPDATE request. If application wants to update its local contact and inform peer to perform target refresh with a new contact, it can specify the new contact in *new_contact* argument; otherwise this argument must be NULL.

Application MAY initiate a new SDP offer/answer session in the request when there is no pending answer to be sent or received. It can detect this condition by observing the state of the SDP negotiator of the invite session. If new offer should be sent to remote, the offer must be specified in *new_offer*, otherwise this argument must be NULL.

```
pj_status_t pjsip_inv_send_msg(   pjsip_inv_session *inv,
                                  pjsip_tx_data *tdata,
                                  void *token );
```

Send request or response message in *tdata*. The token is an arbitrary application data that will be put in the transaction's **mod_data** array, at application module's index.

11.2.6 Auxiliary API

```
pjsip_inv_session* pjsip_dlg_get_inv_session(  pjsip_dialog *dlg );
```

Get the invite session instance associated with dialog *dlg*, or NULL.

```
pjsip_inv_session* pjsip_tsx_get_inv_session(  pjsip_transaction *tsx );
```

Get the invite session instance associated with transaction *tsx*, or NULL.

Chapter 12:Dialog Subscribe Usage

(TO BE DONE)