HP-UX C SIP Stack Programmer's Guide

HP-UX 11i v2

Version 4.5

Manufacturing Part Number: 5992-3333

November 2007

Legal Notices

CONTENTS

11 *Working with Transmitters*

12 *Event Notification*

About This Document

This document discusses the SIP protocol in brief and describes how to use SIP stack libraries for developing SIP applications. It describes how to create, compile, and run applications that use the C SIP APIs on systems running HP-UX 11i v2.

The document printing date and part number indicate the document's current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

The latest version of the document will be available at:

http://www.docs.hp.com.

Document updates can be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new edition, subscribe to the appropriate support service.

Contact your HP sales representative for details.

Intended Audience

This document is intended for developers who wish to build SIP Stack applications. Readers are expected to be familiar with the following:

- System administration concepts
- UNIX operating system concepts
- Networking concepts

Document Organization

The *HP-UX C SIP Stack Programmer's Guide* is organized as follows:

Chapter 1	**Session Initiation Protocol (SIP)** Introduces SIP Stack and its key constituents.
Chapter 2	**Sip Stack** Describes the HP implementation of SIP Stack features at various levels.
Chapter 3	**API Conventions** Describes how to use API structures and functions using an object-oriented methodology that are implemented in C.

Chapter 4	**Creating An Application** Describes the basic code necessary to write an application using SIP Stack.
Chapter 5	**Sip Threading Model** Describes how SIP Stack can work in any of three threading modes; No-threads, Threadsafe, and Multithreaded.
Chapter 6	**Working With Call-legs (Dialogs)** Describes how SIP Stack uses call-leg APIs between two User Agents.
Chapter 7	**Working With Transactions** Describes how to use functions and function callbacks in a Transaction API of SIP Stack. It also describes the second purpose of the Transaction API that is writing SIP Stack server, using which you can implement a Proxy server, Redirect server and Registrar.
Chapter 8	**Working With Register-Clients** Describes how a Register-Client API of the SIP Stack enables you to register with a Proxy or a SIP Stack server, refresh registration when needed, and send authentication information to the server.
Chapter 9	**Working With SIP Messages** Describes how to use the flexible API for working with SIP Stack messages and message parts, such as headers and addresses.
Chapter 10	**Authentication** Describes how the authentication mechanism enables a User Agent Client (UAC) to prove authenticity to servers or proxies which require authentication.
Chapter 11	**Working With Transmitters** Describes how to use transmitter objects (transmitters) for message sending in a SIP Stack.
Chapter 12	**Event Notification** Describes how the Event Notification feature provides an extensible framework by which SIP Stack nodes can request notification from remote nodes.
Chapter 13	**Refer** Describes how to use the REFER method in SIP Stack defined by RFC 3515.
Chapter 14	**Working With The Transport Layer** Focuses on the connection-oriented reliable transport types and explains how to maintain a persistent connection and how to use the TCP, and TLS transports with the SIP Stack.

Chapter 15	**Working With DNS** Describes how to use DNS procedures to allow a client to resolve a SIP Stack Uniform Resource Identifier (URI) into the IP address, port, and transport protocol.
Chapter 16	**Working With Resolvers** Describes how to use resolvers to produce data that is related to DNS.
Chapter 17	**SIP Stack Log** Describes how to use the logging module that produces output for debugging, monitoring and tracking of the activity of applications built with the SIP Stack.
Chapter 18	**Memory Pool** Describes what are memory pools and the different types of memory pools used for different needs.
Chapter 19	**Advanced Features** Describes all the advance features available in SIP Stack.
Chapter 20	**Configuration** Describes how to configure the SIP Stack.
Chapter 21	**Working With the Mid-Layer** Describes the three major groups of the Mid-layer API; Mid-layer Management API, Mid-layer Timer API, and Mid-layer Select API.
Chapter 22	**Sample Applications** Describes the usage of some sample applications in SIP Stack.

Typographic Conventions

This document uses the following typographic conventions:

`monospace`	Computer output, files, directories, software elements such as command options, function names, and parameters. Read tunables from the `/etc/vx/tunefstab` file.
italic	New terms, book titles, emphasis, and variables replaced with a name or value.
%	C shell prompt
$	Bourne/Korn shell prompt
#	Superuser prompt (all shells)
\	Continued input on the following line; you do not type this character.
[ ]	In command synopsis, brackets indicate an optional argument. `ls [ -a ]`

|	In command synopsis, a vertical bar separates mutually exclusive arguments. `mount [ suid | nosuid ]`
`Ctrl+A`	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the plus.

Related Information

Additional information about SIP Stack is available at:

`http://docs.hp.com`

This website contains the following documents about SIP Stack:

- *HP-UX SIP Release Notes*
- *HP-UX C SIP Stack Reference Guide*
- *HP-UX C SIP Stack Messages Layer Reference Guide*
- *JSR32 SIP Programmer's Guide*

HP Welcomes Your Comments

HP welcomes your comments concerning this document. We are committed to providing documentation that meets your needs.

Send your comments or suggestions to: `feedback@fc.hp.com`

Include the document title, manufacturing part number, and any comment on the error found in this document. Also, include what we did right, so we can incorporate it into other documents.

1

SESSION INITIATION PROTOCOL (SIP)

INTRODUCTION

The Session Initiation Protocol (SIP) is a signaling protocol for initiating, managing and terminating voice and video sessions across packet networks. Borrowing from Internet protocols, such as HTTP and SMTP, SIP is text-encoded and highly extensible. SIP can be extended to accommodate features and services such as call control services, mobility and interoperability with existing telephony systems.

SIP is being standardized by the SIP Working Group and others, within the Internet Engineering Task Force (IETF). The protocol is published as RFC 3261 and includes numerous extensions to the basic SIP Protocol.

This section describes the key constituents of SIP.

SIP ENTITIES

A SIP network is composed of five types of logical SIP entities. Each entity has specific functions and participates in SIP communication as a client (initiates requests), as a server (responds to requests), or as both. One “physical device” can have the functionality of more than one logical SIP entity. For example, a network server working as a Proxy server can also function as a Registrar at the same time.

The logical SIP entities are:

- User Agent
- Proxy Server
- Redirect Server
- Registrar Server
- Back-to-Back User Agent (B2BUA)

User Agent

In SIP, a User Agent (UA) is the endpoint entity. User Agents initiate and terminate sessions by exchanging requests and responses. RFC 3261 defines the User Agent as an application, which contains both a User Agent client and User Agent server, as follows:

- User Agent Client (UAC)—a client application that initiates SIP requests.
- User Agent Server (UAS)—a server application that contacts the user when a SIP request is received and that returns a response on behalf of the user.

Some of the devices that can have a UA function in a SIP network are workstations, IP-phones, telephony gateways, call agents, and automated answering services.

Proxy Server

A Proxy server is an intermediary entity that acts as both a server and a client, for the purpose of making requests on behalf of other clients. Requests are serviced either internally or by passing them on, possibly after translation, to other servers. A Proxy interprets, and, if necessary, rewrites a request message before forwarding it.

Redirect Server

A Redirect server is a server that accepts a SIP request, maps the SIP address of the called party into zero (if there is no known address) or more new addresses and returns them to the client. Unlike Proxy servers, Redirect Servers do not pass the request onto other servers.

Registrar

A Registrar is a server that accepts REGISTER requests for the purpose of updating a location database with the contact information of the user specified in the request.

B2BUA

A B2BUA is a logical entity that receives a request, processes it as a User Agent Server (UAS) and, in order to determine how the request should be answered, acts as a User Agent Client (UAC) and generates requests. A B2BUA must maintain call state and actively participate in sending requests and responses for dialogs in which it is involved. The B2BUA has tighter control of the call than a Proxy—for example, a Proxy cannot disconnect a call or alter the messages.

Messages

The following sections deal with SIP messages.

Message Types

There are two types of SIP messages:

- Requests—sent from the client to the server.

- Responses—sent from the server to the client.

Requests

Table 1-1 Request Messages

Method	Description
INVITE	Initiates a call, changes call parameters (re-INVITE).
ACK	Confirms a final response for INVITE.
BYE	Terminates a call.
CANCEL	Cancels searches and “ringing”.
OPTIONS	Queries the capabilities of the other side.
REGISTER	Registers with the Location Service.
INFO	Sends mid-session information that does not modify the session state.

Responses

Response messages contain numeric response codes. The SIP response code set is partly based on HTTP response codes. There are two types of responses and six classes, as follows:

Response Types

- Provisional (1xx class)—provisional responses are used by the server to indicate progress, but they do not terminate SIP transactions.
- Final (2xx, 3xx, 4xx, 5xx, 6xx classes)—final responses terminate SIP transactions.

Classes

- 1xx = Provisional—request received, continuing to process the request.
- 2xx = Success—the action was successfully received, understood and accepted.
- 3xx = Redirection—further action needs to be taken to complete the request.

- 4xx = Client Error—the request contains bad syntax or cannot be fulfilled at this server.
- 5xx = Server Error—the server failed to fulfill an apparently valid request.
- 6xx = Global Failure—the request cannot be fulfilled at any server.

Table 1-2 *Response Codes*

Number	Meaning
100	Trying
180	Ringing
200	OK
300	Multiple choices
30	Moved permanently
302	Moved temporarily
400	Bad request
401	Unauthorized
403	Forbidden
408	Request time-out
480	Temporarily unavailable
481	Call/Transaction does not exist
482	Loop detected
500	Server internal error
600	Busy everywhere

Table 1-2 Response Codes (Continued)

Number	Meaning
603	Decline
604	Does not exist anywhere
606	Not acceptable

MESSAGE PARTS

SIP messages are composed of the following three parts:

- Start line
- Headers
- Message body

START LINE

Every SIP *message* begins with a Start Line. The Start Line conveys the message type (method type in requests, and response code in responses) and the protocol version. The Start Line may be either a Request-line (requests) or a Status-line (responses), as follows:

- The Request-line includes a Request-URI, which indicates the user or service to which this request is being addressed.
- The Status-line holds the numeric Status-code and its associated textual phrase.

HEADERS

SIP header fields convey message attributes that provide additional information about the message. They are similar in syntax and semantics to HTTP header fields (in fact, some headers are borrowed from HTTP) and thus always take the format:

```
<name>:<value>
```

Headers can span multiple lines. Some SIP headers such as Via, Contact, Route and Record-Route can appear multiple times in a message or, alternatively, can take multiple comma-separated values in a single header occurrence.

MESSAGE BODY

A message body is used to describe the session to be initiated (for example, in a multimedia session this may include audio and video codec types and sampling rates), or alternatively it may be used to contain opaque textual or binary data of any type which relates in some way to the session. Message bodies can appear both in request and in response messages. SIP makes a clear distinction between signaling information, conveyed in the SIP Start Line and headers, and the session description information, which is outside the scope of SIP.

Possible body types include:

- Multipurpose Internet Mail Extensions (MIME)
- Others—to be defined in the IETF and in specific implementations

MESSAGE SAMPLES

The following samples show the message exchange between two User Agents for the purpose of setting up a voice call. SIP user alice@hp.com invites SIP user bob@acme.com to a call for the purpose of discussing lunch. Alice sends an INVITE request containing a body. Bob replies with a 200 OK response also containing a body.

REQUEST MESSAGE

Table 1-3 *Request Message Samples*

Request Message Line	Description
INVITE sip:bob@acme.com SIP/2.0	Request line: Method type, Request-URI (SIP address of called party), SIP version.
Via: SIP/2.0/UDP 172.20.1.1:5060; branch=z9hG4bK-2f059	Identifies the location where the response is to be sent.
Max-Forwards:70	Limits the number of hops the request will make on the way to its destination.
From: Alice A. <sip:alice@hp.com>;tag=1121137	User originating this request. Includes a unique tag.
To: Bob B. <sip:bob@acme.com>	User being invited, as specified originally.
Call-ID: 2388990012@alice_ws.hp.com	Globally unique ID of this call.
CSeq: 1 INVITE	Command sequence. Identifies transaction.
Contact:<sip:alice@pc33.hp.com>	Direct route to contact Alice in further requests.
Subject: Lunch today.	Call subject and/or nature.
Content-Length: 182	Number of bytes in the body.

Table 1-3 Request Message Samples (Continued)

Request Message Line	Description
(blank line)	Blank line marks end of SIP headers and beginning of body.
v=0	Version.
o=Alice 53655765 2353687637 IN IP4 128.3.4.5	Owner/creator and session identifier, session version address type and address.
s=Call from Alice.	Session subject.
c=IN IP4 alice_ws.hp.com	Connection information.
M=audio 3456 RTP/AVP 0 3 4 5	Media description: type, port, possible formats caller is willing to receive and send.

Response Message

Table 1-4 Response Message Samples

Response Message line	Description
SIP/2.0 200 OK	Status line: SIP version, response code, reason phrase.
Via: SIP/2.0/UDP 172.20.1.1:5060; branch=z9hG4bK-2f059	Copied from request.
From: Alice A. <sip:alice@hp.com>;tag=1121137	Copied from request.
To: Bob B. <sip:bob@acme.com>;tag=17462311	Copied from request. Includes unique tag to identify call-leg.
Call-ID: 2388990012@alice_ws.hp.com	Copied from request.
CSeq: 1 INVITE	Copied from request.
Contact:<sip:bob@172.20.1.77>	Direct route to contact Bob.
Content-Length: 200	

Table 1-4 *Response Message Samples (Continued)*

Response Message line	Description
(blank line)	Blank line marks end of SIP headers and beginning of the body.
v=0	Version.
o=Bob 4858949 4858949 IN IP4 192.1.2.3	Owner/creator and session identifier, session version address type and address.
s=Lunch	Session subject.
c=IN IP4 machine1.acme.com	Connection information.
m=audio 5004 RTP/AVP 0 3	Description of media streams the receiver of the call is willing to accept.

ENTITY INTERACTION

This section describes the interaction between SIP entities in various common session initiation scenarios.

SESSION ESTABLISHMENT AND TERMINATION

Figure 1-1 shows the interaction between two user agents during trivial session establishment and termination.

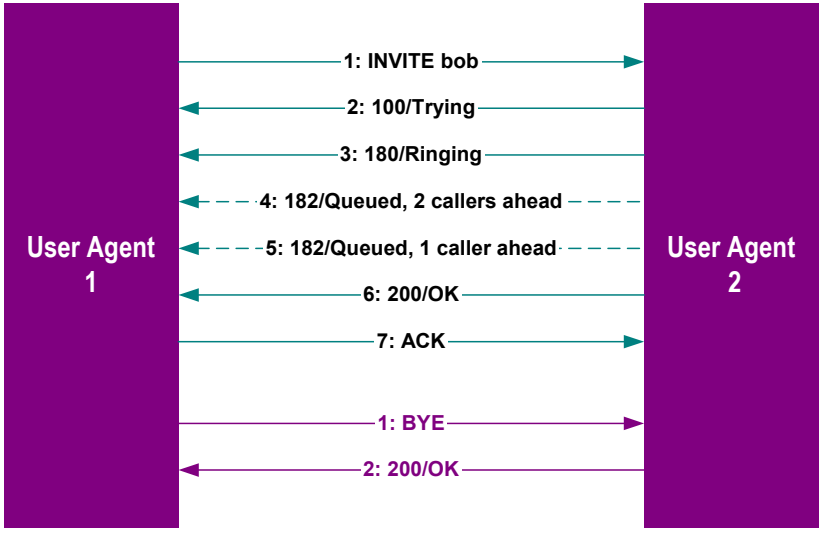


Figure 1-1 *SIP Session Establishment and Call Termination*

SESSION ESTABLISHMENT

CALL FLOW

1. UA1 sends an INVITE message to Bob's SIP address: sip:bob@acme.com. This message also contains a packet describing the media capabilities of the calling terminal.
2. UA2 receives the request and immediately responds with a 100-Trying response message.
3. UA2 starts "ringing" to inform Bob of the new call. Simultaneously a 180 (Ringing) message is sent to the UAC.
4. UA2 sends a 182 (Queued) call status message to report that the call is behind two other calls in the queue.
5. UA2 sends a 182 (Queued) call status message to report that the call is behind one other call in the queue.

6. Bob picks up the call and the UA2 sends a 200 (OK) message to the calling UA. This message also contains a packet describing the media capabilities of Bob's terminal.
7. UA1 sends an ACK request to confirm the 200 (OK) response was received.

SESSION TERMINATION

The session termination call flow proceeds as follows:

1. The caller decides to end the call and "hangs-up". This results in a BYE request being sent to UA2.
2. UA2 responds with 200 (OK) message and notifies Bob that the conversation has ended.

CALL REDIRECTION

Figure 1-2 below shows a simple call redirection scenario.

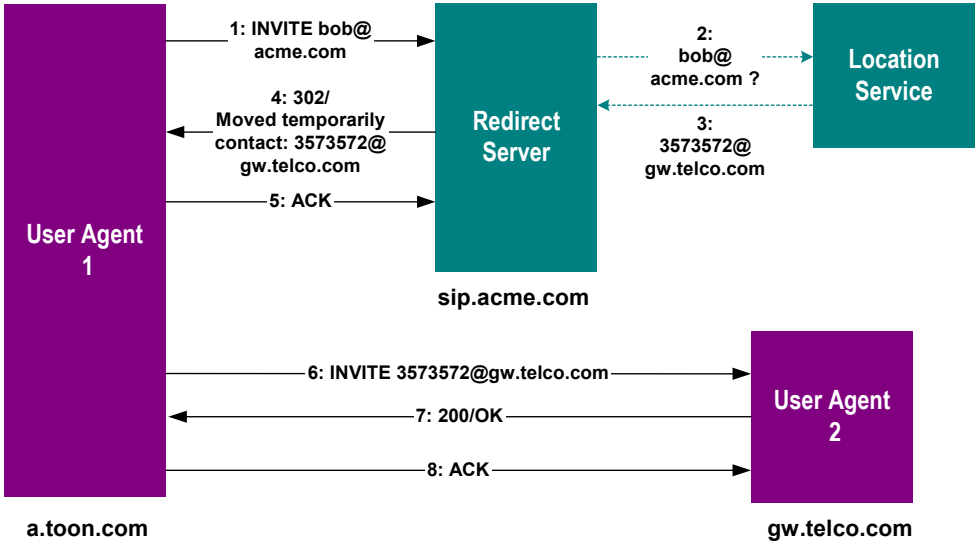


Figure 1-2 *Simple Call Redirection Using a Redirect Server*

Call Flow

1. First a SIP INVITE message is sent to bob@acme.com, but finds the Redirect server sip.acme.com along the signaling path.
2. The Redirect server looks up Bob's current location in a Location Service using a non-SIP protocol (for example, LDAP).
3. The Location Service returns Bob's current location: SIP address 3573572@gw.telco.com.
4. The Redirect Server returns this information to the calling UA using a 302 (Moved Temporarily) response. In the response message it enters a contact header and sets the value to Bob's current location, 3573572@gw.telco.com.
5. The calling UA acknowledges the response by sending an ACK message.
6. The calling UAC then continues by sending a new INVITE directly to gw.telco.com.
7. gw.telco.com is able to notify Bob's terminal of the call and Bob "picks up" the call. A 200 (OK) response is sent back to the calling UA.
8. The calling UA acknowledges with an ACK message.

CALL PROXYING

Figure 1-3 shows call set-up between two User Agents with the assistance of an intermediate Proxy server.

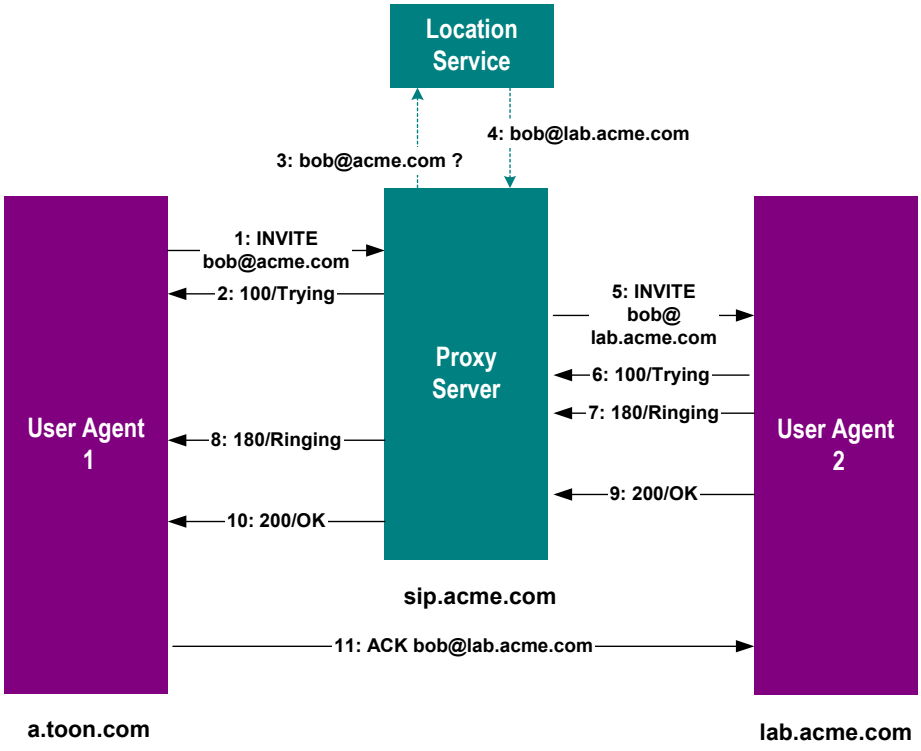


Figure 1-3 *Call Proxying Scenario*

CALL FLOW

1. An INVITE message is sent to bob@ acme.com, but finds the Proxy server sip.acme.com along the signaling path.
2. The Proxy server immediately responds with a 100 (Trying) provisional response.
3. The Proxy server looks-up Bob's current location in a Location Service using a non-SIP protocol (For example, LDAP).
4. The Location Service returns Bob's current location: SIP address bob@lab.acme.com.

5. The Proxy server decides to Proxy the call and creates a new INVITE transaction based on the original INVITE message, but with the Request-URI in the start line changed to bob@lab.acme.com. The Proxy server sends this request to the UA2.
6. The UA2 responds first with a 100 (Trying).
7. The UA2 responds with a 180 (Ringing) response.
8. The Proxy server forwards the 180 (Ringing) response back to the UA1.
9. When the call is accepted by the user (for example, by picking up the handset) UA2 sends a 200 (OK) response. In this example, UA2 inserts a Contact header into the response with the value bob@lab.acme.com. Further SIP communication will be sent directly to it and not via the Proxy server.
10. The Proxy forwards the 200 (OK) response back to the calling UAC.
11. The calling UA sends an ACK directly to UA2 at the lab (according to the Contact header it found in the 200 (OK) response).

2

SIP STACK

INTRODUCTION

This chapter describes the SIP Stack's architecture, the relationship between Stack objects, APIs, standards and different kinds of support, enhanced features and add-on module in SIP.

The HP implementation of SIP provides the following features:

- Enable media over IP communication
- A building block to develop wide range of applications from cellular phones to high-end servers
- Develop a complete solution for both native and embedded platforms, that implement call-level, transaction-level and message-level functionality

TOOLKIT COMPONENTS

The SIP Toolkit comprises the following two components, as illustrated in Figure 2-1:

- SIP Stack
- OS Abstraction layer (Common Core)

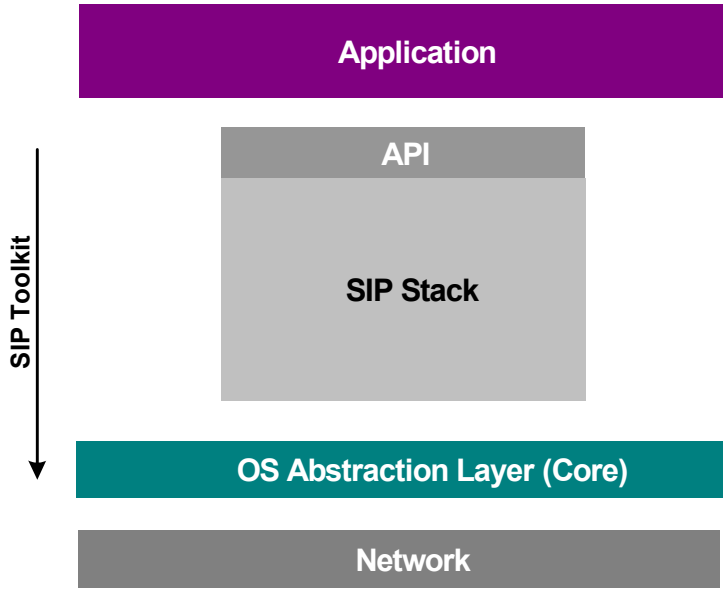


Figure 2-1 *SIP Toolkit Components*

Note Note The SIP Toolkit is coded in ANSI C.

SIP Stack Architecture

The SIP Stack consists of five main layers. The Stack Manager, User Agent, Transaction Layer and Message Layer each export a dedicated API. Figure 2-2 shows the relationship between the various layers under the application.

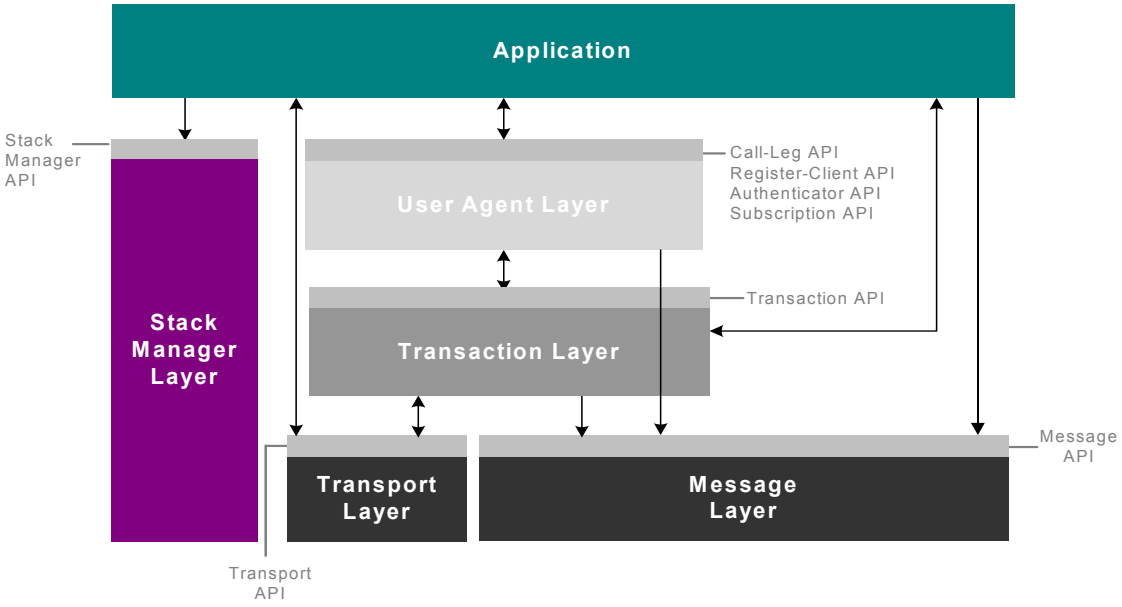


Figure 2-2 *SIP Stack Architecture*

Stack Manager Layer

The Stack Manager layer sets the system configuration, memory allocation, logging, and other resources. The Stack Manager is also responsible for the initialization and shutdown of all other layers.

User Agent Layer

The User Agent layer creates and manages *call-leg*, *subscription* and *register-clients*. The User Agent layer also maps incoming *transactions* to *call-leg* and *register-client* objects.

Note The User Agent layer can be used for both client and server applications.

TRANSACTION LAYER

The Transaction layer creates and manages *transaction* objects. Each *transaction* is responsible for maintaining states, and sending and receiving messages and retransmissions using the Transport layer. The Transaction layer also maps incoming messages to *transactions.*

TRANSPORT LAYER

The Transport layer handles SIP networking I/O. This layer manages UDP sockets and TCP connections, as specified in RFC 3261, and sends and receives messages.

MESSAGE LAYER

The Message layer handles parsing and encoding of SIP *messages*. The message layer allows browsing and editing of the contents of SIP *messages* and also comparison of message parts, such as SIP addresses.

PARSER ENGINE

SIP is a text-encoded protocol, which means that messages are sent on the wire in textual format—as opposed to binary encoding formats such as ASN.1 PER. Although encoding text messages is relatively simple, parsing them can be more complex because of the large degree of freedom provided by text grammars. SIP grammar is specified in RFC 3261, which uses ABNF[1] to represent the message structure.

The SIP Stack use a dedicated rule based LALR[2] parser implemented according to the syntax and grammar of the protocol. This parser is optimized for performance and is thread-safe so that multiple messages can be parsed simultaneously in different thread contexts.

1. Augmented Backus-Naur Form
2. Look Ahead Left-to-right parse, Rightmost-derivation

SIP Stack Objects

This section describes the main objects that the SIP Stack uses. Figure 2-3 illustrates the ownership relationship between SIP Stack objects.

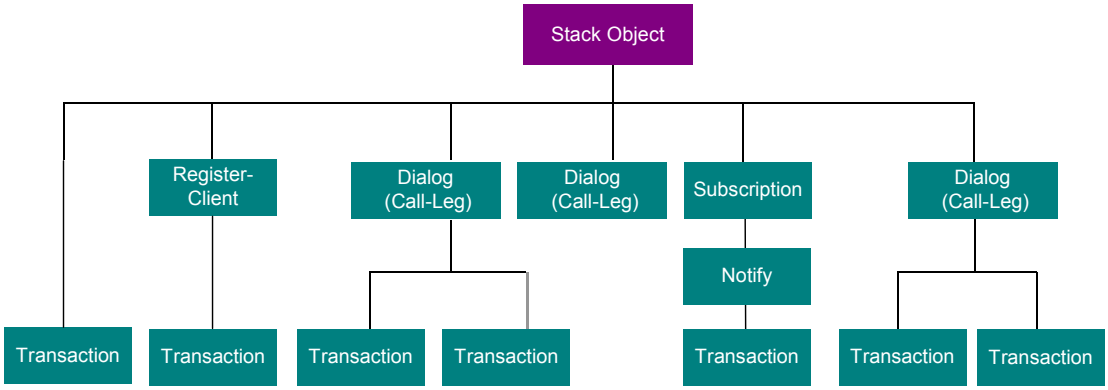


Figure 2-3 *Ownership Relationship Between SIP Stack Objects*

Stack Object

The SIP Stack object manages all SIP Stack initialization, configuration and memory management.

Dialog (Call-Leg)

A SIP call-leg maps to a single SIP call-leg, which is essentially a signaling relationship between two SIP endpoints. The following fields uniquely identify SIP call-legs:

- Call-ID
- SIP addresses of both endpoints (To and From headers)
- A special SIP identifier called “tag” attached to the call-leg. To and From headers

In the SIP Stack, a call-leg object (*call-leg*) stores states and manages *transactions* on behalf of the call-leg.

Register-Client

A SIP Registration is a process by which clients notify their current locations to Registrars that store this information in location servers. In the SIP Stack, a register-client object (*register-client*) is responsible for client-side registrations.

Transaction

A SIP *transaction* involves all messages sent between a client and server for the purpose of completing one signaling action, such as call establishment and call termination. The *call-leg* to which each *transaction* belongs and an identifier field called "CSeq" (command sequence) uniquely identifies the transaction.

In the SIP Stack, a *transaction* stores the *transaction* states and manages *transaction* progress through the use of event handling and state machines. A *transaction* can also be used outside the context of a specific *call-leg* to accomplish User Agent signaling action.

Subscription

A subscription object (*subscription*) manages SUBSCRIBE-initiated dialogs in the SIP Stack as defined in RFC 3265 (SIP Specific Event Notification). Each *subscription* handles the initial SUBSCRIBE request and any REFRESH requests that follow. *Subscriptions* are also used to send Notifications (NOTIFY requests—see below) as part of the Notifier-side functionality, and to inform of incoming notifications as Subscriber-side functionality.

Notification

A notify object (*notify*) is used to send and receive NOTIFY requests in accordance with RFC 3265 (SIP Specific Event Notification).

Application Programming Interface (API)

The SIP Stack provides an intuitive and object-oriented C API for the development of SIP-enabled applications. The API includes a set of optional callbacks that allow your application to intervene in different phases of object establishment and termination. The SIP Stack can be made fully functional and operational with a minimal amount of initialization and configuration.

API Modules

The SIP Stack APIs are divided into three layers, giving the user full control of the functionality of messages and SIP Stack state machines. In addition, the SIP Stack provides high level abstract APIs that handle protocol complexity and provide an interface for building SIP applications with minimal coding effort. Mid-level APIs enable sending and receiving messages from the primitive (transaction) level interface. Low level APIs provide access to SIP *messages* and enable the application to modify SIP Stack functionality. These APIs give the user tight control over the SIP Stack functionality, enabling it to deploy non-standard/unsupported functionality and to interwork with non-standard SIP applications. These different levels of APIs may be used in parallel, as described in more detail throughout this guide. The following sections describe the main API modules.

Stack Manager API

The Stack Manager API enables your application to perform the following:

- Stack configuration

- Resource management
- Stack initialization and shutdown
- Logging

HIGH-LEVEL API

The High-level API includes:

- Dialog (Call-leg) API
- Register-Client API
- Authentication API
- Subscription API

DIALOG (CALL-LEG) API

The Dialog (Call-leg) API enables you to create calls, terminate calls, modify existing calls (re-INVITE), authenticate calls and more. The Dialog (Call-leg) API also provides a set of callbacks and hooks that provide you with the ability to add your own call processing logic to the application without recompiling the SIP Stack.

REGISTER-CLIENT API

The Register-Client API enables you to register your User Agent to a Registrar. The Register-Client API provides a set of callbacks and hooks that enables you to take part in the registration process.

AUTHENTICATION API

The Authentication API enables you to verify the authenticity of the originator of incoming requests and to add authentication information to outgoing requests.

SUBSCRIPTION API

The Subscription API enables you to manage *subscriptions* and to send and receive SUBSCRIBE, REFER and NOTIFY requests.

MID-LEVEL API

The Mid-level API includes:

- Transaction API

TRANSACTION API

The Transaction API enables you to handle *transactions* that are not related to a *call-leg*, such as OPTIONS. Using the API, you can create new *transactions*, send outgoing requests, and respond to incoming requests. The Transaction API provides a set of callbacks and hooks that enables you to control some of the *transaction* behavior. You can use the Transaction API to implement SIP servers such as Registrars or Proxies.

LOW-LEVEL API

The Low-level API includes:

- Message API
- Transmitter API
- Transport API

MESSAGE API

The Message API provides access to SIP *message* content, enabling the application to browse and edit any part of the message. There are API functions for complete SIP *messages* and also for message parts, such as headers and URLs.

The message API is also the interface to the parsers and encoders provided with the SIP Stack. You can copy, store and re-use messages and message parts.

TRANSMITTER API

The Transmitter API enables you to send a single *message* that does not belong to a *transaction*. The *transmitter* is responsible for address resolution and the actual message sending, including DNS queries such as SRV and NAPTR.

TRANSPORT API

The Transport API provides control over connection persistency, TLS and TOS implementation, and dynamic local address functionality. It also provides low-level hook functions for tight control over SIP Stack functionality.

CONFORMANCE TO STANDARDS

SIP Stack was developed in conformity with IETF standards and includes SIP related features.

SIP STACK

The SIP Stack component of the SIP Stack was developed in conformity with the specifications of RFC 3261 and various SIP extensions.

Methods

The SIP Stack supports baseline SIP methods, such as INVITE, ACK, BYE, CANCEL, OPTIONS and REGISTER. In addition, the SIP Stack supports extension methods, such as REFER, NOTIFY, SUBSCRIBE, PRACK, MESSAGE, UPDATE, INFO and others.

Responses

The SIP Stack supports all response code classes (1xx to 6xx) specified in RFC 3261.

New Message Parameters

The SIP Stack can accept and encode new non-standard message fields which do not appear in the baseline specification, such as methods, response codes, headers and header parameters.

Security

The SIP Stack can authenticate any SIP request using the Digest authentication scheme in conformity with the SIP specification. Both Client authentication and Server authentication are supported. The SIP Stack also supports the TLS Transport layer which provides message encryption.

Multipart MIME Bodies

Multipart MIME is a generic mechanism that allows the encapsulation and transfer of arbitrary chunks of data, both text and binary, within text messages. Today, Multipart MIME is used successfully in e-mail and the web. The SIP specification allows the usage of Multipart MIME as *message* bodies. Multipart MIME is a necessary capability for some features, such as ISUP message tunneling through SIP, which is part of SIP-T. Multipart MIME is also useful for other purposes, such as sending thumbnail pictures of the caller inside of INVITE messages.

Advanced DNS Queries

RFC 3263 (Locating SIP Servers) defines procedures for using advanced SRV and NAPTR DNS queries to determine the transport protocol, IP address and port at which a specific SIP server is available. These procedures can be used to dynamically update server locations and to implement redundancy among servers for fault tolerance or load balancing.

Feature Support

The SIP Stack includes a set of IETF extensions that are of high importance, and that are otherwise difficult to implement.

SUBSCRIBE-NOTIFY (SIP Events)

RFC 3265 (SIP Specific Event Notification) is a SIP extension that allows for subscription and event notifications using SIP. SIP Events is an important infrastructure for services such as Presence and Message Waiting Indication.

REFER

The SIP Stack allows the implementation of transfer services through the use of the REFER extension method, as defined in RFC 3515. This includes the usage of NOTIFY messages. REFER processing is fully controllable by the application via the Subscription API.

SIP-T

SIP-T (SIP for Telephony, RFC 3372) is an umbrella specification defining how to interwork SIP with PSTN (SS7/ISUP) networks. The SIP-T specification does not define any new SIP extensions, but rather uses existing extensions (PRACK, INFO) and other SIP advanced features, such as Multipart MIME and 183 response request, in order to translate between ISUP and SIP *messages* and, in some cases, tunnel ISUP in SIP.

The SIP Stack provides all the necessary building blocks needed by a SIP-T compliant application (for example, Softswitches and PSTN GWs).

PRACK

PRACK (RFC 3262) is an IETF SIP extension that allows for the sending of provisional responses (1xx class) in a reliable way. One of the new elements introduced in this extension is the PRACK (PRovisional ACKnowledgment) method, which is used to indicate the reception of the provisional response, and RSeq and RAck message headers to identify the provisional response. PRACK is useful for such things as opening of one-way media sessions before call establishment, and for QoS negotiation before completing the INVITE transaction.

The SIP Stack implements PRACK, providing fully automated behavior with optional application control.

ENUM

In VoIP networks, an endpoint has an IP address, and may have a telephone number. When the telephone number is known, the corresponding IP address needs to be found. ENUM is a standard that enables translating telephone numbers to SIP addresses. Translation is done using the existing internet DNS (NAPTR) query mechanism. The ENUM mechanism is a requirement in cellular IMS networks, and many fixed and mobile operators also choose to implement ENUM.

The SIP Stack version 4.0 provides users with a fast implementation of the ENUM mechanism. Both client and server developers can use this feature.

CONNECTION REUSE

When SIP entities use a connection-oriented protocol to send a request, they typically originate their connections from an ephemeral port. The SIP Protocol includes mechanisms that insure that responses to a request, and new requests sent in the original direction, reuse an existing connection. However, new requests sent in the opposite direction are unlikely to reuse the existing

connection. This frequently causes a pair of SIP entities to use one connection for requests sent in each direction, and can result in potential scaling and performance problems. The Connection Reuse draft tries to solve this by adding an alias parameter in the Via header. This parameter tells the application on the remote side to map the existing connection for future requests from the original side.

The SIP Stack version 4.0 implements the Connection Reuse draft and enables heavily loaded applications to save connection ports and make faster connections.

Parsing TEL, PRES and IM URIs

A SIP address can be described in several formats called URIs. A “tel” URI is a way to describe a SIP endpoint using its phone number. A “pres” URI is a SIP address used for Presence user agents (UAs). An “im” URI is used for instant messaging. Each URI can appear in many variations and therefore the parsing of these URIs is not a simple task.

The SIP Stack version 4.0 frees application developers from the task of parsing these URIs. The SIP Stack handles any form of the URIs and hands the data to the application in a friendly data structure.

Service-route and Path Headers

The information is transmitted in the initial REGISTER (or after roaming), using Path and Service-Route headers.

The SIP Stack version 4.0 enables the encoding and decoding of the Path and Service-Route headers.

SIP Session Timer

The Session Timer extension enables SIP servers and endpoints to know if an endpoint is no longer in a session (for example in case of a crash) and to limit the duration of a session. An endpoint that has the Session Timer extension activated will send periodic keep-alive messages to notify that it is active or to extend the duration of the session.

Replaces Header

The Replaces extension defines a new header for use with SIP multi-party applications and call control. The Replaces header is used to logically replace an existing SIP dialog with a new SIP dialog. This primitive can be used to enable a variety of features, such as Attended Transfer and Retrieve from Call Park.

Extensibility

The SIP Stack provides rich extensibility capabilities that allow the application developer to easily implement any number of the many extensions to SIP. These include:

- General URL scheme support—general framework to support any type of URL scheme, such as IM: and TEL:.

- INFO method (RFC 2976)—for mid-call information exchange without changing the call state.
- MESSAGE method (draft-ietf-sip-message-0x)—for instant messaging.
- UPDATE method (RFC 3311)—to allow a client to update parameters of a session (such as the set of media streams and their codecs) without changing the dialog state. Sample code for sending/receiving this method is included.

TRANSPORT SUPPORT

The SIP Stack supports SIP over UDP, TCP, and TLS, and provides full UDP reliability as defined in the SIP specification. You can configure the SIP Stack to listen to several UDP, TCP and TLS local addresses.

TLS

TLS is a security protocol which is typically layered on top of connection-oriented transports such as TCP. TLS allows client/server applications to communicate over TCP in a way that is designed to prevent eavesdropping, tampering, or message forgery. TLS 1.0 is defined in RFC 2246.

TLS is now an integral part of baseline SIP (RFC 3261) as a recommended feature of User Agents and a mandatory feature for SIP servers. TLS provides a solution to many of the security issues SIP applications face and it ties in well with the existing SSL/TLS infrastructure that serves HTTP applications.

PERSISTENT CONNECTION

In many cases, a single TCP/TLS connection can be reused for different messages, *transactions* or dialogs. The frequent opening and closing of TCP connections is not desirable because of the extra messaging overhead of the TCP handshake (and even more so in TLS connections). For this reason, SIP permits connection persistency, which is the reuse of an open connection. The SIP Stack enables connection reuse by identifying that a message can be sent on an existing open connection. The following levels of persistency are available:

- Transaction user
- Transaction persistency
- No persistency

IPv6

The SIP Stack supports IP version 6 (IPv6). You can configure the SIP Stack to listen to IPv6 and IPv4 addresses simultaneously, and to receive and send SIP *messages* using IPv6 packets.

ENHANCED FEATURES

The enhanced features of the SIP Stack are described in the sections below.

Enhanced SIP Parser

The enhanced parser functionality provides more flexibility and allows message "correction" by the application. This functionality is available both for sending and receiving SIP *messages*.

Multithreading

SIP Stack can run in internally multithreaded mode (configurable). Your application may either be single-threaded or multithreaded. SIP Stack uses locks to ensure multithreading safety at the level of individual objects, such as *call-legs* and *transactions*.

Configuration

The SIP Stack configuration enables you to configure the following groups of parameters:

- Memory allocation—you can predetermine the resources your application will require during the SIP Stack operation, such as the number of calls, *register-clients*, *transactions*, messages, and transport buffer size.
- Logging levels
- Network parameters
- Timer values
- Other

Logging

The SIP Stack includes a logging module that produces textual output messages for the purpose of debugging, monitoring and tracking the activity of the SIP Stack.

The logger can be configured to work on different levels of detail. You can select a global log level for the entire SIP Stack. You can also assign different log levels to the subcomponents of the SIP Stack, such as Call-leg layer, Transaction layer, Message layer, transport and parser.

Memory Requirements

The SIP Stack is an optimized implementation of the SIP specification with emphasis on minimal memory usage during run-time. The SIP Stack can be fine-tuned for embedded systems and systems with low resources. The SIP Stack has the following characteristics:

- Small code footprint
- Low per-call memory consumption
- Minimal call-stack requirements
- Emphasis on ROM usage (over RAM)
- No dynamic memory allocations
- Efficient internal resource management

3

API CONVENTIONS

INTRODUCTION

The SIP Stack provides a comprehensive set of Application Programming Interface (API) structures and functions using an object-oriented methodology that are implemented in C.

CONVENTIONS

The SIP Stack uses specific conventions for each of the following:

- Return status codes
- Data types
- Parameters

STATUS CODES

Many API functions return status codes in order to indicate the success or failure of the requested operation. Status return values are always of the RvStatus type as shown in Table 3-1.

Table 3-1 *Status Codes*

Value	Meaning
RV_OK	The function was completed successfully.
RV_ERROR_UNKNOWN	Failure of unspecified type.
RV_ERROR_OUTOFRESOURCES	The function can not be executed due to limited resources.
RV_ERROR_BADPARAM	A parameter passed to a function is illegal.

Table 3-1 *Status Codes*

Value	Meaning
RV_ERROR_NULLPTR	The required pointer parameter was a NULL pointer.
RV_ERROR_OUTOFRANGE	A parameter that was passed to a function is out of range.
RV_ERROR_DESTRUCTED	The referred object was already terminated.
RV_ERROR_NOTSUPPORTED	The request is not supported under the current configuration.
RV_ERROR_UNINITIALIZED	The object is uninitialized.
RV_ERROR_TRY_AGAIN	The action cannot be completed—try again later.
RV_ERROR_ILLEGAL_ACTION	The requested action is illegal—usually an illegal action occurring in the current state.
RV_ERROR_NETWORK_PROBLEM	Action failed due to network problems.
RV_ERROR_INVALID_HANDLE	A handle passed to a function is illegal.
RV_ERROR_NOT_FOUND	The requested item cannot be found.
RV_ERROR_INSUFFICIENT_ BUFFER	The buffer is too small.
RV_ERROR_ILLEGAL_SYNTAX	The parser identified a syntax error, or a parser error occurred.
RV_ERROR_OBJECT_ALREADY_ EXISTS	An object with this unique set of values already exists.

A success value is always non-negative and error values are always negative.

The following files contain RvStatus definitions:

- rverror.h
- RV_SIP_DEF.h

To use these definitions, your application needs to include the *RV_SIP_DEF.h* file which automatically includes the *rverror.h* file.

TYPES

For reasons of cross-platform compatibility, the SIP Stack libraries use only the following basic data types:

- RvUint64
- RvUint32
- RvUint16
- RvUint8
- RvInt64
- RvInt32
- RvInt16
- RvInt8
- RvInt
- RvUint
- RvLong
- RvUlong
- RvChar
- RvBool—The Boolean data type, RvBool may take two values, RV_TRUE and RV_FALSE

FUNCTION PARAMETERS

Function parameters are marked as either input, output or input/output parameters:

- IN—input parameter. A parameter that the function uses but does not modify.
- OUT—output parameter. The parameter value is set by the function.
- INOUT—input/output parameter. A parameter that the function uses and then modifies.

4

CREATING AN APPLICATION

INTRODUCTION

This chapter describes the basic code necessary to write an application using the SIP Stack, including:

- Initialization
- Event Processing
- Destruction
- Configuration
- Getting Module Handles

INITIALIZATION

Before performing any SIP-related activity, you must initialize the SIP Stack. The SIP Stack Manager API function, RvSipStackConstruct(), performs SIP Stack initialization. RvSipStackConstruct() receives the following parameters:

- *cfgStructSize*—input parameter. The size of the configuration structure.
- *RvSipStackCfg*—input/output parameter. Set the parameters in the *RvSipStackCfg* structure with your configuration values. You can use RvSipStackInitCfg() to initialize the configuration structure with the SIP Stack default values.
- *RvSipStackHandle*—output parameter. You should supply this parameter when working with the Stack Manager API functions.

RvSipStackConstruct() performs the following:

- Constructs the layers of the SIP Stack.

- Allocates memory.
- Initializes timers, the log file and the interface to the network.

To initialize the SIP Stack

1. Declare the *RvSipStackCfg* structure and the *RvSipStackHandle*.
2. Call RvSipStackInitCfg() to fill the configuration structure with the SIP Stack default values.
3. Set your own configuration values in the configuration structure.

   The remainder of the SIP Stack configuration will be adjusted according to the changes you have made.
4. Initialize the SIP Stack by calling the RvSipStackConstruct() function.

Sample Code

The following code demonstrates how to initialize the SIP Stack.

```
/*===========================================================================*/
RvSipStackCfg         stackCfg;
RvSipStackHandle      hStack;
RvStatus AppInitSipStack()
{
        RvStatus rv;

        RvSipStackInitCfg(sizeof(stackCfg), &stackCfg));
        stackCfg.maxCallLegs = 5;
        stackCfg.maxTransactions = 15;

        rv = RvSipStackConstruct(sizeof(stackCfg), &stackCfg, &hStack);
        if(rv != RV_OK)
        {
                printf("Failed to initialize the Stack");
        }
        return rv;
}
/*===========================================================================*/
```

In the above code, the *maxCallLegs* and *maxTransactions* settings limit the SIP Stack to creating a maximum of 5 calls and 15 *transactions* simultaneously.

The SIP Stack configuration parameters will adjust to the new numbers you set. For more information about the SIP Stack configuration, see the Configuration chapter.

EVENT PROCESSING

After initializing the SIP Stack, you will need to instruct the SIP Stack to process events as they occur. You do this by calling the function, RvSipStackProcessEvents() from the same thread that constructed the SIP Stack.

You can alternatively use the functions, RvSipStackSelect() or RvSipStackSelectUntil(). These functions perform one call to the select() operating system function. Both of the above functions should be called in an application loop.

Sample Code

The following code demonstrates how to instruct the SIP Stack to process incoming events.

```
/*-------------------------------------------------------------------------*/
int main()
{
        RvStatus rv;

      /*Initialize the SIP stack*/
        rv = AppInitSipStack();
        if(rv != RV_OK)
         {
           printf("failed to initialize the SIP Stack\n");
           return -1;
         }

            . . .

      /* Commands the Stack to process the events queue.*/
        RvSipStackProcessEvents();
        return 0;
}
/*-------------------------------------------------------------------------*/
```

DESTRUCTION

The RvSipStackDestruct() function destroys the SIP Stack and frees all resources that the Stack uses. This function must be called from the same thread that constructed the Stack.

CONFIGURATION

The SIP Stack lets you configure SIP Stack resources and behavior. Configuration parameters can be categorized as:

- Resource allocation
- Log filters
- Network configuration
- Timer configuration
- Behavior configuration
- Multithreaded configuration
- Proxy configuration

RESOURCE ALLOCATION

The SIP Stack allocates all required memory upon initialization. This means that you must predetermine the resources that the SIP Stack will require during operation. These resources include the number of calls, *transactions*, register-clients, subscriptions, size, and number of the network buffers and more.

LOG FILTERS

You can control how the SIP Stack produces Log messages by setting filters either individually for each layer of the SIP Stack or for all layers simultaneously. You set the logging filters by configuring the *xxxLogFilters* fields in *RvSipStackCfg* with any combination of the filters listed in Table 4-1.

Table 4-1 *Log Filters*

Filter	Description
RVSIP_LOG_INFO_FILTER	A description of SIP Stack activity.
RVSIP_LOG_DEBUG_FILTER	Provides detailed log messages of SIP Stack activity.
RVSIP_LOG_WARN_FILTER	A warning about a possible non-fatal error.
RVSIP_LOG_EXCEP_FILTER	A fatal error has occurred that prevents the SIP Stack from continuing to operate.

Table 4-1 *Log Filters*

RVSIP_LOG_ERROR_FILTER	An error such as faulty application behavior, insufficient allocations, or illegal network activity. These errors are identified and handled by the SIP Stack.
RVSIP_LOG_LOCKDBG_FILTER	The details of all the locking operations of the Stack.
RVSIP_LOG_ENTER_FILTER	Indicates that an API function was called. This filter effects the logging of the Core and ADS modules only.
RVSIP_LOG_LEAVE_FILTER	Indicates that an API function call was completed. This filter effects the logging of the Core and ADS modules only.

Sample Code

The following code demonstrates how to instruct the call-leg layer to print only INFO and ERROR messages to the Log.

```
/*=========================================================================*/
RvSipStackCfg stackCfg;
RvSipStackInitCfg(sizeof(stackCfg), &stackCfg));
stackCfg.callLogFilters = RVSIP_LOG_INFO_FILTER | RVSIP_LOG_ERROR_FILTER;
/*=========================================================================*/
```

Sample Code

The following code demonstrates how to set the Log filters for all layers of the SIP Stack simultaneously. In this sample, each layer is instructed to print only the DEBUG and WARN messages to the Log.

```
/*=========================================================================*/
RvSipStackCfg stackCfg;
RvSipStackInitCfg(sizeof(stackCfg), &stackCfg);
stackCfg.defaultLogFilters = RVSIP_LOG_DEBUG_FILTER |  RVSIP_LOG_WARN_FILTER;
/*=========================================================================*/
```

Note For more information about working with the Log, see the the SIP Stack Log chapter.

Network Configuration

The SIP Stack allows you to configure the following settings:

- Local IP address—the SIP Stack listening address.

  If you set the local IP address to “0.0.0.0”, the socket will be opened with IP=0.

- Local Port—the SIP Stack listening ports.

  If you set the port number to zero, the Stack Manager layer will use the default port (5060).

Note A local address and port can be configured both for UDP and TCP transports.

- Additional UDP and TCP listening Addresses—the SIP Stack can be configured to listen to additional UDP and TCP addresses.

  For more information, see Multihomed Host of the Advanced Features chapter.

- TLS addresses—if you compile the SIP Stack with TLS transport, you can define several TLS addresses to which the SIP Stack will listen.

- Outbound proxy configuration—the SIP Stack lets you configure a default outbound proxy that will be used for all outgoing requests (unless record-route is used). You can supply the outbound proxy as an IP address or host name. You should also supply the outbound port and transport type.

Timer Configuration

The SIP specification makes use of different timers for different activities. You can configure the SIP Stack to use the standard default values, or you can change some of the values of the timers to suit your application. For more information about the SIP Stack timer configuration, see the Configuration chapter.

Behavior Configuration

Some of the SIP Stack behavior can be configured upon initialization. For example, the manualAckOn2xx parameter determines whether or not ACK on a 2xx response is sent automatically. Fore more information on the behavior configuration, see the Configuration chapter.

Proxy Configuration

The SIP Stack can be used to implement proxy applications. In this case, the SIP Stack needs to be configured with the isProxy=RV_TRUE parameter and with all other proxy-related parameters. Fore more information on the proxy configuration, see the Configuration chapter.

Multithreaded Configuration

The SIP Stack can be configured to span several internal threads. When working in multithreaded mode, the main thread inserts events into a processing queue. The processing threads then take the events from the queue and process them. Your application should configure the number of processing threads and the queue size. Fore more information on the multithreaded configuration, see the Configuration and SIP Stack Threading Model chapters.

Getting Module Handles

In order to use an API of a specific module, you need the manager of that module. For example, in order to use the Call-leg Manager API, you need a handle to the *Call-legMgr*. The Stack Manager API provides functions that enable you to get the required module handle.

Sample Code

The following code demonstrates how to get the *Call-legMgr* handle.

```
/*============================================================*/
RvSipCallLegMgrHandle AppGetCallLegMgr (
                        RvSipStackHandle       hStack)
{
        RvSipCallLegMgrHandle hCallLegMgr;
        RvSipStackGetCallLegMgrHandle (hStack, &hCallLegMgr);
        Return hCallLegMgr;
}
/*============================================================*/
```

5

SIP STACK THREADING MODEL

MULTITHREADING MODES

The SIP Stack can work in one of three threading modes:

- **No-threads mode**—both the SIP Stack and the application work on the same single thread and therefore there is no need to protect the SIP Stack objects with locks. To avoid object locking, you need to set RV_THREADNESS_TYPE to RV_THREADNESS_SINGLE in the *rvusrcongif.h* file found in the *common/config* directory.
- **Threadsafe mode**—the SIP Stack is not multithreaded but the application is. Therefore the SIP Stack protects its objects using locks. In this mode, the *numberOfProcessingThreads* configuration parameter is set to zero and the RV_THREADNESS_TYPE should be set to RV_THREADNESS_MULTI.
- **Multithreaded mode**—the SIP Stack works with one main thread and several processing threads. All SIP Stack objects are protected by locks.

MULTITHREADED MODE

This chapter focuses on the multithreaded mode of the SIP Stack. To work in multithreaded mode, the application should set the *numberOfProcessingThreads* configuration parameter to a value bigger than zero, and set the RV_THREADNESS_TYPE to RV_THREADNESS_MULTI. When working in multithreaded mode, the main thread waits for incoming events (it sits on the select loop). When an event arrives, the main thread inserts the event into a processing queue and notifies the processing threads that an event is waiting in the queue. One of the processing threads will take this event from the queue and

process it. The processing queue is used also in the no-thread and thread safety modes. However, in these modes, it is the main thread that both inserts events into the queue and then takes them out for further processing.

The following events are inserted to the processing queue:

- Message received event—each received message is inserted into the queue and will be processed by one of the processing threads.
- TCP events—read, write, connect and close.
- Object termination events—all SIP Stack objects are terminated in an a-synchronic manner using the processing queue.
- Object states—states that the SIP Stack needs to handle in an a-synchronic manner are inserted into the processing queue. For example, the *connection* states and the MsgSendFailure state of the *transaction*.
- Timer expiration events

Mandatory Actions for Multithreaded Applications

Multithreaded applications **must** ensure the following actions (modes 2 and 3):

- The SIP Stack is constructed from the main thread. The application must also destruct the SIP Stack from the main thread.
- RvSipStackProcessEvents() or any of the select() functions must be called from the main thread.
- The application must lock all application objects. Whenever a callback is called, the application must immediately lock the relevant application object before further processing. The object should be unlocked before the callback returns.
- The application should be prepared to handle events (callbacks) that are received from different threads, including application threads.

Configuration Parameters

The following configuration parameters are related to all the multithreading modes of the SIP Stack:

- *numOfProcessingThreads*
- *numOfReadBuffers*
- *ProcessingQueueSize*

For more information about multithreaded configuration parameters, see the Configuration chapter.

6

WORKING WITH CALL-LEGS (DIALOGS)

INTRODUCTION

A call-leg is a peer-to-peer SIP relationship between two User Agents (UAs) that persists for some time.

The Call-leg API relates to the following entities:

- Call-leg (*call-leg*)
- Call-leg Manager (*Call-legMgr*)

CALL-LEG

A *call-leg* represents a SIP dialog as defined in RFC 3261, and is uniquely identified by the Call-ID, From and To tags. Your application can initiate calls, react to incoming calls and disconnect calls using the Call-leg API. The API functions enable you to send and receive messages with different methods, such as INFO, and to respond to such messages. A *call-leg* is a stateful object which can assume any state from a set defined in the Call-leg API. A Call-leg state represents the state of the session set up between two SIP UAs.

CALL-LEG MANAGER

The *Call-legMgr* manages the collection of all *call-legs*. The *Call-legMgr* is mainly used for creating new call-legs.

WORKING WITH HANDLES

All *call-legs* and the *Call-legMgr* are identified using handles. You must supply these handles when using the Call-Leg API. RvSipCallLegMgrHandle defines the *Call-legMgr* handle. You receive this handle by calling RvSipStackGetCallLegMgrHandle().

RvSipCallLegHandle defines a *call-leg* handle. For outgoing calls, you receive the *call-leg* handle from RvSipCallLegMgrCreateCallLeg(). For incoming calls, you receive the *call-leg* handle from the RvSipCallLegCreatedEv() callback.

CALL-LEG API

The Call-leg API contains a set of functions and function callbacks that allow you to set or examine *call-leg* parameters and control *call-leg* behavior.

CALL-LEG PARAMETERS

You can set or examine *call-leg* parameters via Call-leg Set and Get API functions. The following parameters are available:

To Header, From Header, Call-ID, and CSeq

When creating an outgoing call, you must set the To and From headers of the *call-leg*. A call-leg is uniquely identified by its To and From tags and its Call-ID. You do not have to set the tags or the Call-ID yourself. The SIP Stack will generate random tags and the Call-ID for you. The *call-leg* automatically handles the CSeq, and increases it by one for each outgoing request.

Remote Contact Address

The address for contacting the remote party. For outgoing calls, the remote contact is used as the Request-URI. If you do not set the remote contact, the To address is taken. For incoming calls, the remote contact address is the contact address taken from the received INVITE message. You should set the remote contact only once, for outgoing calls only, and only when the call is in the IDLE state. The SIP Stack automatically updates the remote contact when a refresh request or a 2xx response to a refresh request is received.

The remote contact is also used to specify the transport of an initial request. To send an initial request using TCP, you must set the remote contact transport to TCP.

Local Contact Address

The address that the caller sends to the remote party as a contact address in the Contact header. If you do not set a contact address, the From address is taken. The SIP Stack inserts the contact into every outgoing request, except for BYE and CANCEL. The contact is also added to response messages of different requests, each according to its RFC rules. For incoming calls, the local contact address is taken from the INVITE message Request-URI field, but you can change it using a Set function.

Outbound Address and Local Address

The addresses the *call-leg* uses for sending requests. If you set the *call-leg* outbound address, all requests with no Route headers will be sent to this address regardless of the message Request-URI. The local address defines the address from which the Request will be sent (the network card). This is also the address

that will be placed in the top Via header of the Request message. If the local address is not set, the *call-leg* uses a default local address taken from the SIP Stack configuration.

State

Indicates the state of the session setup between two SIP UAs. You can only access the state parameter with a Get function and it is not modifiable.

Direction

Indicates whether the *call-leg* represents the incoming or outgoing side of the session. You can only access the direction parameter with a Get function and it is not modifiable.

Outbound Message

The outbound message is a handle to a *message* that the *call-leg* will use for the next outgoing message. Before calling an API function that causes a message to be sent, the application can get the outbound *message* and add headers and a body. (Note that at this stage, the *message* is empty.) You cannot use the outbound message to set headers that are part of the *call-leg* or *transaction* key, such as To, From, Call-ID and Via headers. Such headers will be overwritten by the SIP Stack.

Received Message

The last message (Request or Response) that was received by the *call-leg*. You can get this message only in the context of the *call-leg* state-changed callback function when the new state indicates that a *call-leg transaction* received a new message. For example, when the *call-leg* enters the OFFERING state you can get the incoming Invite Request *message*.

This parameter cannot be used for requests that the SIP Stack answers automatically, such as CANCEL.

Call-leg Transaction Timers and Retransmission Count

The SIP Stack configuration determines the value of the timers and retransmission count for all the SIP Stack transactions. The application can use a set function to change the different timer values of the *call-leg transactions*. The application can also control the number of retransmissions that the *transactions* perform.

Persistency Definition and Used Connection

When working with TCP, the application can instruct the *call-leg* to try and send all outgoing requests on one TCP connection. The application can also query the *call-leg* about the *connection* used to send each request. For more information on the persistency level and persistent connection API functions, see Persistent Connection Handling of the Working with the Transport Layer chapter.

New Message Element Handles

Some of the call-leg fields are message parts. For example, the To header field is a Party header object and the Local Contact field is a SIP address object. Before setting a parameter of this type in the call-leg, you should request a new handle for the parameter from the call-leg. After initializing the message part, you can set it back in the call-leg.

CALL-LEG CONTROL

The API functions that are listed below provide call-leg control.

BASIC FUNCTIONALITY

The following API functions provide basic call-leg functionality:

RvSipCallLegMake()

After creating a call-leg, you can use this function to set the To and From headers in the call-leg and send the INVITE request. This function receives the To and From headers in a textual format.

RvSipCallLegConnect()

After creating a call-leg and setting the To and From headers, you can use RvSipCallLegConnect() for generating and sending the required INVITE message to connect the call.

RvSipCallLegAccept()

You can use RvSipCallLegAccept() in the OFFERING state to accept an incoming call. You can also use this function to accept an incoming re-INVITE request.

RvSipCallLegReject()

You can use RvSipCallLegReject() in the OFFERING state to reject an incoming call. You can also use this function to reject an incoming re-INVITE request.

RvSipCallLegCancel()

Sends CANCEL on an outgoing INVITE message if the final response was not yet received. This function can be used in one of the following *call-leg* states:

- PROCEEDING
- PROCEEDING_TIMEOUT

It can also be used in one of the following re-INVITE states:

- REINVITE_PROCEEDING
- REINVITE_PROCEEDING_TIMEOUT

RvSipCallLegAck()

When the SIP Stack is configured to work in a manual ACK mode, the *call-leg* will not send an ACK message after receiving an INVITE 2xx response. After receiving the 2xx response, the *call-leg* will assume the REMOTE_ACCEPTED state. The application must use the RvSipCallLegAck() function to trigger the *call-leg* to send the ACK.

RvSipCallLegProvisionalResponse()

You can use RvSipCallLegProvisionalResponse() to send a provisional response before sending a final response. For example, when the call-leg is in the OFFERING state, you can use this function to send a 100 Trying or a 180 Ringing provisional response.

RvSipCallLegAuthenticate()

If a *call-leg* receives a 401 or 407 response indicating that a request was unauthenticated by the server or Proxy, the *call-leg* assumes the UNAUTHENTICATED state. You can use RvSipCallLegAuthenticate() in the UNAUTHENTICATED state to re-send the request with authentication information. If you do not wish to proceed with the authentication process, you should terminate the *call-leg*.

Note This function should not be used to authenticate a re-INVITE or general request inside a *call-leg*.

RvSipCallLegDisconnect()

You can call RvSipCallLegDisconnect() at any point in the life span of a call-leg in order to disconnect the call. The *call-leg* reaction to this function depends on the current *call-leg* state. For example, if the call-leg is in the CONNECTED state, calling this function causes a BYE message to be sent. The call-leg then moves to the DISCONNECTING state. If the *call-leg* is in the IDLE state, calling RvSipCallLegDisconnect() terminates the *call-leg*. In this case, the *call-leg* assumes the TERMINATED state.

RvSipCallLegTerminate()

The RvSipCallLegTerminate() function causes a *call-leg* to be terminated without sending any messages (CANCEL or BYE). The *call-leg* will assume the TERMINATED state. Calling this function causes an abnormal termination of the *call-leg*. All *transactions* related to the *call-leg* will be terminated as well.

PRACK Functionality

The following API functions allow the application to support RFC 3262. To use these functions you must configure the SIP Stack to support the 100rel option tag.

RvSipCallLegProvisionalResponseReliable()

You can use RvSipCallLegProvisionalResponseReliable() to send a reliable provisional response (1xx class other then 100) to the remote party. This function can be called only when an INVITE request is received, such as in the OFFERING state and the MODIFY_REINVITE_RECEIVED state.

RvSipCallLegSendPrack()

When the SIP Stack is configured to work in a manual PRACK mode, the application is responsible for generating the PRACK message whenever a reliable provisional response is received. When a reliable provisional response is received, the *call-leg* PRACK state machine assumes the REL_PROV_RESPONSE_RCVD state. You should then call the RvSipCallLegSendPrack() function to send the PRACK message to the remote party. The *call-leg* PRACK state machine will then assume the PRACK_SENT state. When calling this function, the SIP Stack is responsible for adding the RAck header to the PRACK message.

RvSipCallLegSendPrackResponse()

When the SIP Stack is configured to work in a manual PRACK mode, the application is responsible for responding to any PRACK request that is received for a previously-sent reliable provisional response. When a PRACK request is received, the *call-leg* PRACK state machine assumes the PRACK_RCVD state. You should then call the RvSipCallLegSendPrackResponse() function to send a response to the PRACK request. The *call-leg* PRACK state machine will then assume the PRACK_FINAL_RESPONSE_SENT state.

Enhanced Functionality

The following API functions provide enhanced functionality.

RvSipCallLegUseFirstRouteForInitialRequest()

Sometimes an application may want to use a pre-loaded Route list, when sending an initial INVITE request. The application can do this with the following actions:

1. Add the route headers to the outbound message.
2. Call the RvSipCallLegUseFirstRouteForInitialRequest() function. Calling this function notifies the Stack that it should take the outbound message Route list into consideration, while calculating the remote target.

RvSipCallLegSetRejectStatusCodeOnCreation()

You can use this function synchronously from the RvSipCallLegCreatedEv() callback to instruct the Stack to automatically reject the INVITE request that created this *call-leg*. In this function you should supply the reject status code. The *call-leg* will be destructed automatically when the RvSipCallLegCreatedEv() returns and the *transaction* will be responsible for rejecting the request. You will not get any further callbacks that relate to this *call-leg*. (You will not get the RvSipCallLegMsgToSendEv() for the reject response message, or the Terminated state for the *call-leg*.)

Remarks:

- This function should not be used for rejecting a request in a normal scenario. For this, you should use the RvSipCallLegReject() function. You should use this function only if your application is incapable of handling this new *call-leg* at all, for example, in when an application is in an out-of-resource situation.

- When this function is used to reject a request, you cannot use the outbound message mechanism to add information to the outgoing response message. If you wish to change the response message, you must use the regular reject function in the OFFERING state.

RvSipCallLegSetForceOutboundAddrFlag()

You can use this function to force the *call-leg* to send every outgoing request to the outbound address, regardless of the message content or object state.

EVENTS

The Call-leg API supplies several events, in the form of callback functions, to which your application may listen and react. In order to listen to an event, your application should first define a special function called the event handler and then pass the event handler pointer to the *Call-legMgr*. When an event occurs, the call-leg calls the event handler function using the pointer.

The following events are supplied with the Call-leg API:

RvSipCallLegCreatedEv()

Notifies the application that a new incoming call-leg has been created. The newly created *call-leg* always assumes the IDLE state. Your application can exchange handles with the SIP Stack using this callback.

Note You must not terminate the *call-leg* from this event.

RvSipCallLegStateChangedEv()

This event is probably the most useful of the events that SIP *call-leg* reports. Through this function, you receive notifications of SIP call-leg state changes and the associated state change reason. For example, upon receipt of an OFFERING state notification, your application may decide whether to accept or reject the call.

RvSipCallLegMsgToSendEv()

The call-leg calls this event whenever a *call-leg* related message is ready to be sent. You can use this callback for changing or examining a message before it is sent to the remote party. The address resolution process will start only after this callback returns.

Note You must not terminate the *call-leg* from this event.

RvSipCallLegFinalDestResolvedEv

Indicates that the *call-leg* is about to send a message after the destination address was resolved. This callback supplies the final *message* and the *transaction* that is responsible for sending this message. Changes in the message at this stage will not effect the destination address. When this callback is called, the application can query the *transaction* about the destination address using the RvSipTransactionGetCurrentDestAddress() function. If the application wishes, it can update the sent-by part of the top-most Via header. The application must not update the branch parameter.

Note To change the destination address resolved from the *message*, the application must use the Transmitter API. The application should first get the *transmitter* from the *transaction* using the RvSipTransactionGetTransmitter() API function. In can then manipulate the DNS list and the current destination address of the *transmitter* before the message is sent.

RvSipCallLegMsgReceivedEv()

The call-leg calls this event whenever a *call-leg* related message has been received and is about to be processed. You can use this callback to examine incoming messages.

Note You must not terminate the *call-leg* from this event.

RvSipCallLegPrackStateChangedEv()

Notifies the application of a *call-leg* PRACK event. The PRACK state indicates the status of a PRACK process. By default, this callback is only a notification to the application and no response is expected for the different states. When configuring the SIP Stack to work in a manual PRACK mode, the application is

responsible for moving the PRACK state machine using the Call-leg PRACK API. You can find the different PRACK states in the RvSipCallLegPrackState enumeration.

RvSipCallLegProvisionalResponseRcvdEv()

The *call-leg* calls this callback whenever a provisional response has been received.

BYE Request Callbacks

The following two callbacks are called when a BYE request is received. If the application does not implement the following callbacks the SIP Stack automatically accepts the BYE and terminates the *call-leg*.

RvSipCallLegByeCreatedEv()

Notifies the application that a new BYE *transaction* has been created. The application can exchange the *transaction* handles with the SIP Stack using this callback. The application can specify weather it wishes to handle the request. If so, it will be notified of the BYE *transaction* states and will have to respond to the BYE request.

RvSipCallLegByeStateChangedEv()

Through this callback function you receive notifications of SIP *call-leg* server BYE state changes and the associated state change reason. The application can decide whether to accept or reject the BYE request with the functions RvSipCallLegByeAccept() and RvSipCallLegByeReject().

For more information on the above functions, see the *SIP Stack Reference Guide*.

Call-leg State Machine

The Call-leg state machine represents the state of the session setup between two SIP UAs. The RvSipCallLegStateChangedEv() callback reports *call-leg* state changes and state change reasons. The state change reason indicates how the *call-leg* reached the new state.

Basic Call-leg States

Some of the *call-leg* states are basic and common to most *call-leg* scenarios. Advanced *call-leg* states depend on SIP Stack configuration. The *call-leg* associates with the following basic states:

RVSIP_CALL_LEG_STATE_IDLE

The initial state of the Call-leg state machine. Upon call-leg creation, the call-leg assumes the IDLE state. It remains in this state until RvSipCallLegConnect() is called, whereupon it should move to the INVITING state.

RVSIP_CALL_LEG_STATE_INVITING

After calling RvSipCallLegConnect(), which will send an INVITE request, the *call-leg* enters the INVITING state. The *call-leg* remains in this state until it receives a final response from the remote party. If a 1xx class response is received, the *call-leg* moves to the PROCEEDING state. If a 2xx class response is received, the call-leg assumes the CONNECTED state. If a 3xx class response is received, the *call-leg* moves to the REDIRECTED state. If the call is rejected with a 4xx, 5xx or 6xx class response, the call-leg assumes the DISCONNECTED state.

RVSIP_CALL_LEG_STATE_PROCEEDING

The call assumes the PROCEEDING state when it receives a provisional response in the INVITING state. If a 2xx class response is received, the *call-leg* assumes the CONNECTED state. If a 3xx class response is received, the *call-leg* moves to the REDIRECTED state. If the call is rejected with a 4xx, 5xx or 6xx class response, the *call-leg* assumes the DISCONNECTED state. If no response is received and the provisional timer expired, the *call-leg* moves to the PROCEEDING_TIMEOUT state if the SIP Stack is configured with enableInviteProceedingTimeoutState=RV_TRUE. Otherwise, the *call-leg* will be terminated.

RVSIP_CALL_LEG_STATE_CANCELLING

If the application calls the RvSipCallLegCancel() or RvSipCallLegDisconnect() functions on a *call-leg* while in the PROCEEDING or PROCEEDING_TIMEOUT states, a CANCEL request is sent and the *call-leg* assumes the CANCELLING state. If a positive 200 response is received on the Invite request despite the cancel attempt, the *call-leg* generates a BYE request and moves to the DISCONNECTING state. If, however, a non-200 response is received, the *call-leg* moves to the DISCONNECTED state.

RVSIP_CALL_LEG_STATE_CANCELLED

Upon receiving a CANCEL request in the OFFERING state, the SIP Stack will automatically accept the CANCEL and move to the CANCELLED state. If the SIP Stack is configured to work in the manual behavior mode, your application will be responsible for responding with 487. Otherwise, 487 will be sent automatically. If you call the RvSipCallLegAccept() function in this state, the *call-leg* will move to the ACCEPTED state.

RVSIP_CALL_LEG_STATE_REDIRECTED

A *call-leg* in the INVITING state may receive a 3xx class response. In this case, the call-leg assumes the REDIRECTED state. The *call-leg* automatically sends an ACK message and then updates the remote contact to the first contact address found in the 3xx message. At this point, you may confirm the redirection by calling the RvSipCallLegConnect() function again and the request will be sent to the updated remote contact. You can also decide to terminate the call using the RvSipCallLegTerminate() function.

RVSIP_CALL_LEG_STATE_UNAUTHENTICATED

A *call-leg* sending an INVITE request to a server or Proxy may receive a 401 or 407 response. In this case, the *call-leg* assumes the UNAUTHENTICATED state. At this point, you may re-send your request with authentication information by calling the RvSipCallLegAuthenticate() function. You can also terminate the call using the RvSipCallLegTerminate() function.

RVSIP_CALL_LEG_STATE_OFFERING

Upon receipt of the initial INVITE by an incoming call, the call-leg assumes the OFFERING state. In this state, it is up to you to decide whether to accept or reject the call using the Call-Leg API.

RVSIP_CALL_LEG_STATE_ACCEPTED

If you accept a call in the OFFERING state, the *call-leg* assumes the ACCEPTED state. The *call-leg* moves to the CONNECTED state upon receipt of an ACK message from the calling party.

RVSIP_CALL_LEG_STATE_CONNECTED

The *call-leg* is connected. This state indicates a successful session setup. The *call-leg* reaches this state either when a 200 final response is received on an initial INVITE, or when an ACK is received on a 200 final response. While in

this state, you can use the RvSipCallLegReInviteCreate() and RvSipCallLegReInviteRequest() functions to initiate a re-INVITE. You can also cause the call to disconnect by calling the RvSipCallLegDisconnect() function. Calling the RvSipCallLegDisconnect() function causes the state to change to DISCONNECTING.

RVSIP_CALL_LEG_STATE_DISCONNECTING

If RvSipCallLegDisconnect() is called while in the CONNECTED state, a BYE request is sent to the remote party and the call assumes the DISCONNECTING state. Upon receipt of a response on the BYE request, the *call-leg* assumes the DISCONNECTED state.

RVSIP_CALL_LEG_STATE_DISCONNECTED

When a *call-leg* receives a BYE request from the remote party, or a final response to a previously sent BYE request, it assumes the DISCONNECTED state. The *call-leg* can also reach the DISCONNECTED state when an incoming call is rejected or when there is a time-out. This state notifies you that the call has disconnected and is about to be terminated. This is the final state at which you can still reference the *call-leg* using the Call-Leg API functions.

RVSIP_CALL_LEG_STATE_TERMINATED

This state is the final *call-leg* state. When a call is terminated, the *call-leg* assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *call-leg*.

Basic Call-leg State Machine

Figure 6-1 illustrates the Basic Call-leg state machine, showing the main states involved in connecting and disconnecting calls.

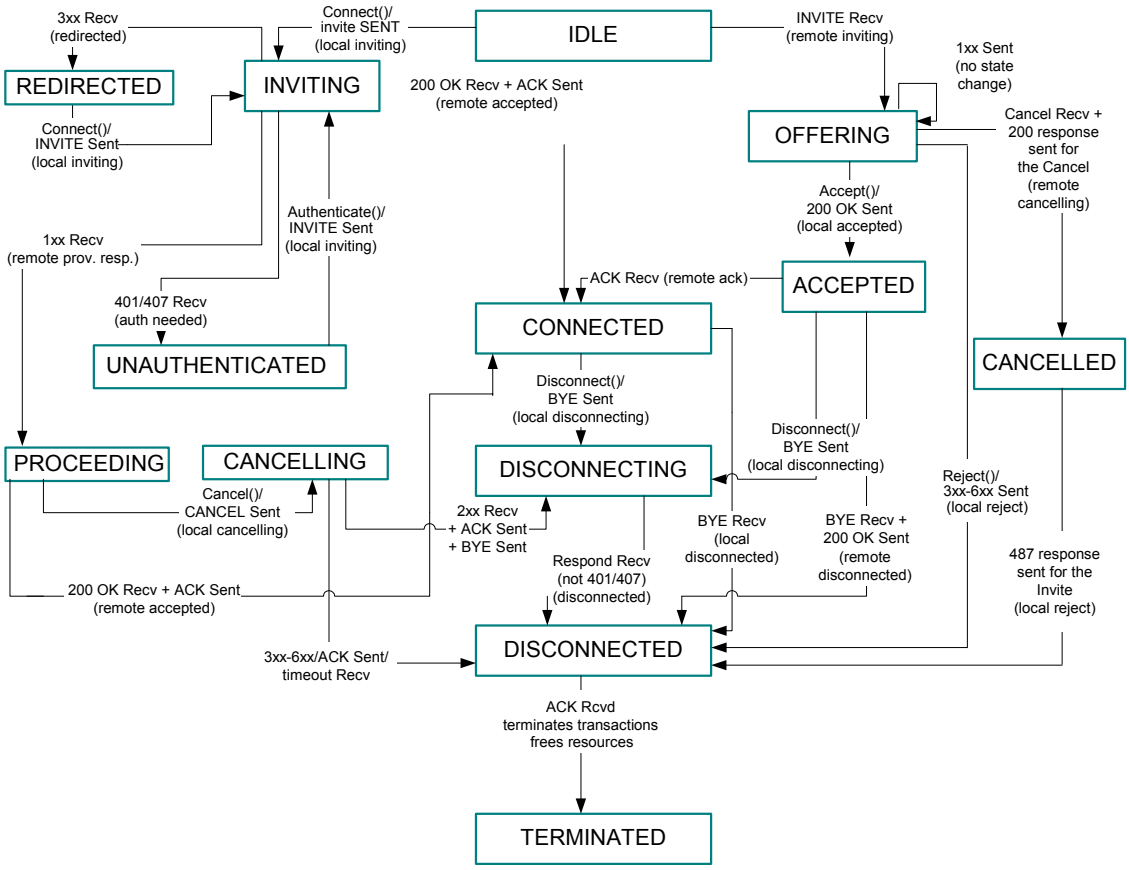


Figure 6-1 *Basic Call-leg State Machine*

Advanced Call-leg States

The *call-leg* associates with the advanced states that appear below. Figure 6-2, Figure 6-3 and Figure 6-4 illustrate the Advanced Call-leg state machines.

RVSIP_CALL_LEG_STATE_PROCEEDING_TIMEOUT

This state is assumed only if the *enableInviteProceedingTimeoutState* configuration parameter is set to RV_TRUE when the SIP Stack initializes. The call moves to this state from the PROCEEDING state when the provisional timer expires before receiving a final response. In this state, the application can cancel the *call-leg* or terminate it.

PROCEEDING-TIMEOUT STATE MACHINE

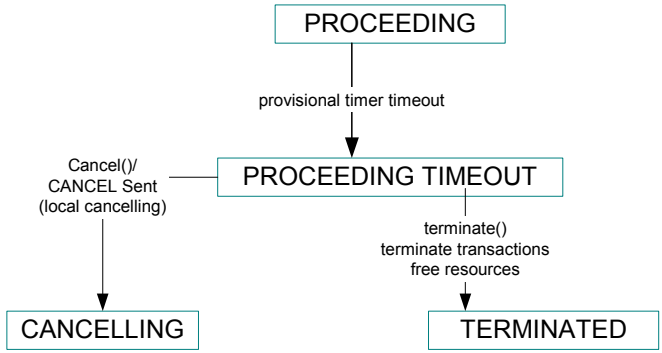


Figure 6-2 *Proceeding-Timeout State Machine*

RVSIP_CALL_LEG_STATE_REMOTE_ACCEPTED

This state is assumed only if the *manualAckOn2xx* configuration parameter is set to RV_TRUE when the SIP Stack initializes. If the *call-leg* received a 2xx response to an initial INVITE request and the SIP Stack is configured to work in a manual ACK mode, the SIP Stack will assume the REMOTE_ACCEPTED state. In this state, the application must call the RvSipCallLegAck() function is order to send the ACK request to the remote party. After the ACK is sent, the *call-leg* will assume the CONNECTED state.

REMOTE-ACCEPTED STATE MACHINE

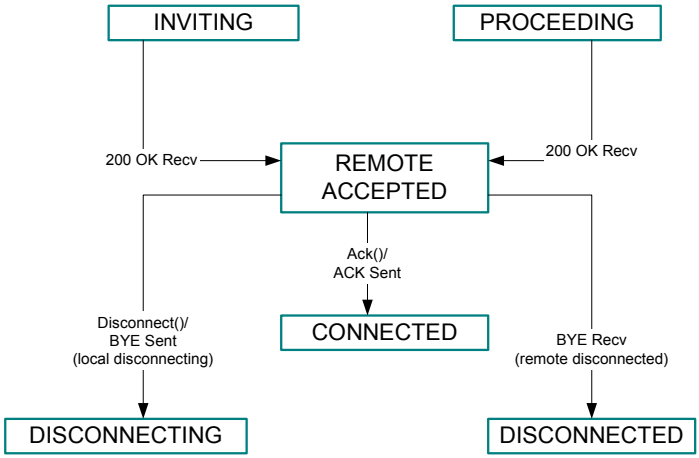


Figure 6-3 *Remote-Accepted State Machine*

RVSIP_CALL_LEG_STATE_MSG_SEND_FAILURE

This state is assumed only if the SIP Stack is compiled with the Enhanced DNS feature. The call moves to this state when the *call-leg* failed to send a request (the *call-leg* received a network error, 503 response or timeout on the request). In this state your application can:

- Continue DNS—try to send the request to the next address in the *transaction* DNS list. For more information see the Working with DNS chapter.
- Give Up—terminate the *call-leg* or return to the previous state of the call and not send the request, depending on the previous state of the *call-leg*. For more information see the Working with DNS chapter.
- Terminate the *call-leg*.

This state can be reached from the INVITING, PROCEEDING or DISCONNECTING states.

MSG-SEND-FAILURE STATE MACHINE

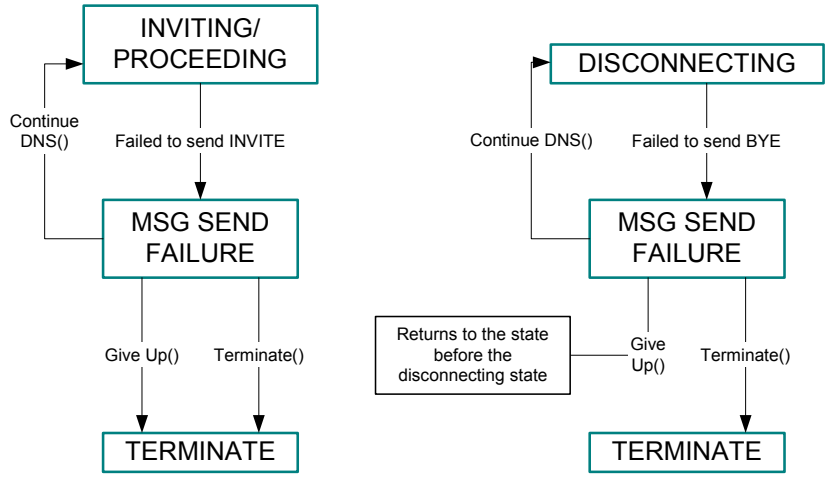


Figure 6-4 *Msg-Send-Failure State Machine*

CALL-LEG MANAGER API

The *Call-legMgr* controls the SIP Stack collection of *call-legs*. You use the Call-Leg Manager API to register application callbacks with the SIP Stack and to create new *call-legs*.

RvSipCallLegMgrCreateCallLeg()

You can use RvSipCallLegMgrCreateCallLeg() to create a new *call-leg*.

RvSipCallLegMgrSetEvHandlers()

You use RvSipCallLegMgrSetEvHandlers() to set your event handler (callback function) pointers in the SIP Stack.

The Call-leg API declares prototypes for all call-leg callback functions. For example:

```
/*-------------------------------------------------------------------------*/
typedef void
     (RVCALLCONV * RvSipCallLegStateChangedEv)(
                IN      RvSipCallLegHandle                  hCallLeg,
                IN      RvSipAppCallLegHandle               hAppCallLeg,
                IN      RvSipCallLegState                   eState,
                IN      RvSipCallLegStateChangeReason       eReason);
/*-------------------------------------------------------------------------*/
```

According to the prototypes, you can implement any callbacks you find necessary. All callback functions are gathered together in a structure called RvSipCallLegEvHandlers. This structure is where you should set your callback function pointers and is given as a parameter to RvSipCallLegMgrSetEvHandlers(). The *call-leg* notifies of an event using the callback functions you implemented.

If you are not interested in certain events, enter a zero value in RvSipCallLegEvHandlers for callback functions which manage those events. The *call-leg* will not notify you when these events occur.

Although you can set the event handlers at any time, it is customary to set them immediately after RvSipStackConstruct() so that the application receives all necessary notifications immediately.

Registering Application Callbacks

To register an application callback, you must first define the callback according to the prototype. The following sample code demonstrates the implementation of two callback functions.

Sample Code

```
/*-----------------------------------------------------------------------------*/
/*Implements the call-leg created event handler. Prints the handle of the new
 incoming call-leg.*/

void RVCALLCONV AppCallLegCreatedEvHandler(
                IN  RvSipCallLegHandle        hCallLeg,
                OUT RvSipAppCallLegHandle     *phAppCallLeg)
{
  printf("Incoming call-leg %x was created\n",hCallLeg);
}
/*-----------------------------------------------------------------------------*/
```

Sample Code

```
/*-----------------------------------------------------------------------------*/
  /*Implements the call-leg state change event handler. Accepts all incoming calls.*/

void RVCALLCONV AppCallLegStateChangedEvHandler(
                IN  RvSipCallLegHandle               hCallLeg,
                IN  RvSipAppCallLegHandle            hAppCallLeg,
                IN  RvSipCallLegState                eState,
                IN  RvSipCallLegStateChangeReason    eReason)
{
```

```
        if (eState == RVSIP_CALL_LEG_STATE_OFFERING)
        {
                RvSipCallLegAccept(hCallLeg);
        }
}
/*===========================================================================*/
```

The following steps describe how to register your application callbacks.

To register application callbacks

1. Declare a RvSipCallLegEvHandlers structure.
2. Initialize all the structure members to NULL using memset().
3. Set the application defined callback in RvSipCallLegEvHandlers.
4. Call RvSipCallLegMgrSetEvHandlers().

Sample Code

The following code demonstrates an application implementation of callback registration:

```
/*===========================================================================*/

void SetCallLegEvHandlers(RvSipCallLegMgrHandle hMgr)
{

      /*step 1*/
      RvSipCallLegEvHandlers appEvHandlers;

      /*step 2*/
      memset(&appEvHandlers, 0, sizeof
             (RvSipCallLegEvHandlers));

      /*step 3*/
      appEvHandlers.pfnCallLegCreatedEvHandler =  AppCallLegCreatedEvHandler;
      appEvHandlers.pfnStateChangedEvHandler   =  AppStateChangedEvHandler;

      /*step 4*/
      RvSipCallLegMgrSetEvHandlers(hMgr,
                      &appEvHandlers,
                      sizeof(appEvHandlers));
}
```

```
/*=========================================================================================*/
```

Exchanging Handles with the Application

The SIP Stack enables you to create your own handle to a *call-leg*. This will prove useful when you have your own application *call-leg* database. You can provide the SIP Stack with your *call-leg* handle, which it must supply when calling your application callbacks. You can use RvSipCallLegMgrCreateCallLeg() to exchange handles for outgoing calls and RvSipCallLegCreatedEv() to exchange handles for incoming calls. You can change your application handle at any time with the RvSipCallLegSetAppHandle() function.

Initiating a Call

The following steps describe how to initiate a call:

To initiate a call

1. Declare a handle for the new *call-leg*.
2. Call the RvSipCallLegMgrCreateCallLeg() function. This enables you to exchange handles with the SIP Stack.
3. Call the RvSipCallLegMake() function. This function sends an INVITE request to the remote party.

Sample Code

The following code demonstrates an implementation of the call invitation procedure.

```
/*=========================================================================================*/
void AppConnectCall(IN RvSipCallLegMgrHandle hCallLegMgr)
{
  RvSipCallLegHandle      hCallLeg; /*Handle to the call-leg.*/
  RvChar                  *strFrom =    "From:sip:user1@172.20.0.1:5060";
  RvChar                  *strTo =      "To:sip:user2@172.20.10.11:5060";

  RvStatus                rv;
  /*---------------------------
   Creates a new call-leg.
  ---------------------------*/
  rv = RvSipCallLegMgrCreateCallLeg(hCallLegMgr,NULL,&hCallLeg);
  if(rv != RV_OK)
  {
                printf("Failed to create new call-leg\n");
                return;
```

```
    }
    printf("Outgoing call-leg %x was created\n",hCallLeg);
    /*-----------------------------------------------------------------------------
     Calls the make function with the To and From strings in order to connect the call.
    -----------------------------------------------------------------------------/
    printf("Connecting a call: %s -> %s\n", strFrom, strTo);
    rv = RvSipCallLegMake(hCallLeg,strFrom,strTo);
    if(rv != RV_OK)
    {
            printf("Connect failed for call-leg");
            return;
    }
}
/*=============================================================================*/
```

MAKING A TCP CALL

The transport type of an initial INVITE is determined by the transport parameter of the Request-URI. The Request-URI is taken from the remote contact parameter of the *call-leg*. If the remote contact was not set, the address of the To header is taken. According to RFC 3261, the address in the To and From headers must not include the transport parameter. Therefore, to send an INITIAL invite using TCP transport, you must set the *call-leg* remote contact parameter and indicate that the transport is TCP. In the same way, to receive further requests on TCP, the local contact must be set with the TCP transport.

The following steps describe how to initiate a TCP call. In this example, the user wishes to send and receive requests using TCP and therefore, both the remote contact and the local contact are set.

To make a TCP call

1. Create a call as described in steps 1, and 2 of the sample, Initiating a Call.
2. Get two address handles from the *call-leg* and fill them with the local and remote contact addresses. Both addresses indicates that the transport is TCP.
3. Set the local and remote contact addresses to the *call-leg*.
4. Call the RvSipCallLegMake() function. This functions sends the INVITE request to the remote party. The request is sent using TCP transport.

Sample Code

The following sample code demonstrates a TCP call.

```
/*=========================================================================================*/
void AppConnectTCPCall(IN RvSipCallLegMgrHandle hCallLegMgr)
{
  /*Handles to the call-leg and the contact addresses.*/
  RvSipCallLegHandle hCallLeg                          = NULL;
  RvSipAddressHandle hLocalContactAddress              = NULL;
  RvSipAddressHandle hRemoteContactAddress             = NULL;

  RvChar   *strFrom = "From:sip:user1@172.20.1.49:5060";
  RvChar   *strTo   = "To:sip:user2@172.20.1.49:5060";
  RvChar   *strLocalContactAddress  = "sip:user1@172.20.1.49:5060;transport=TCP";
  RvChar   *strRemoteContactAddress = "sip:user2@172.20.1.49:5060;transport=TCP";
  RvStatus rv, rv1;

  /*---------------------------
  Creates a new call-leg.
  ---------------------------*/
  rv = RvSipCallLegMgrCreateCallLeg(hCallLegMgr,NULL,&hCallLeg);
  if(rv != RV_OK)
  {
          printf("Failed to create new call-leg\n");
          return;
  }
  printf("Outgoing call-leg %x was created\n",hCallLeg);

  /*---------------------------------------
   Gets address handles from the call-leg.
   ---------------------------------------*/
  rv = RvSipCallLegGetNewMsgElementHandle(hCallLeg,RVSIP_HEADERTYPE_UNDEFINED,
                                          RVSIP_ADDRTYPE_URL,&hLocalContactAddress);

  rv1 = RvSipCallLegGetNewMsgElementHandle(hCallLeg,RVSIP_HEADERTYPE_UNDEFINED,
                                           RVSIP_ADDRTYPE_URL,&hRemoteContactAddress);
  if(rv != RV_OK || rv1 != RV_OK)
  {
          printf("Failed to create contact addresses\n");
          return;
  }
```

```
  /*-----------------------------------------------------
     Fills the address handles with the contact information.
     -----------------------------------------------------*/
  rv  = RvSipAddrParse(hLocalContactAddress,strLocalContactAddress);
  rv1 = RvSipAddrParse(hRemoteContactAddress,strRemoteContactAddress);

  if(rv != RV_OK || rv1 != RV_OK)
  {
          printf("Failed to fill contact addresses\n");
          return;
  }

  /*-----------------------------------------
     Sets the contact address to the call-leg.
    -----------------------------------------*/
  rv  = RvSipCallLegSetLocalContactAddress(hCallLeg,hLocalContactAddress);
  rv1 = RvSipCallLegSetRemoteContactAddress(hCallLeg,hRemoteContactAddress);

  if(rv != RV_OK || rv1 != RV_OK)
  {
          printf("Failed to set contact addresses to call-leg\n");
          return;
  }

  /*-------------------------------------------------------------------------------
  Calls the make function with the To and From strings in order to connect the call.
  -------------------------------------------------------------------------------*/
  printf("Connecting a call: %s -> %s\n", strFrom, strTo);
  rv = RvSipCallLegMake(hCallLeg, strFrom, strTo);
  if(rv != RV_OK)
  {
          printf("Connect failed for call-leg");
          return;
  }
}
/*=============================================================================*/
```

Sample Code

The following sample code demonstrates sending a re-INVITE request. In this example an outgoing re-INVITE object is created in a given *call-leg* and is used to send a re-INVITE request.

```
/*=========================================================================================*/
RvStatus AppCallModify(IN RvSipCallLegHandle hCallLeg)
{
    RvStatus  rv = RV_OK;
    RvSipCallLegInviteHandle hReInvite;

    /* 1. Create a re-INVITE object */
    rv = RvSipCallLegReInviteCreate(hCallLeg, NULL, &hReInvite);
    if (RV_OK != rv)
    {
        printf("RvSipCallLegReInviteCreate() failed (rv=%d)", rv);
        return rv;
    }
    /* 2. Here we can set parameters to the call-leg outbound msg */

    /* 3. Send the re-INVITE request */
    rv = RvSipCallLegReInviteRequest(hCallLeg, hReInvite);
    if (RV_OK != rv)
    {
        printf("RvSipCallLegReInviteRequest() failed (rv=%d)", rv);
        return rv;
    }
    return rv;
}
/*=========================================================================================*/
```

USING THE OUTBOUND MESSAGE MECHANISM

The outbound message mechanism enables adding fields to the *message* before calling the function that actually sends the message. The following sample code demonstrates how to add a “Subject: Hello” header to the initial INVITE request.

Note In this sample, the return value of the SIP Stack API functions is not checked for simplicity. You must always check return values.

```
/*=========================================================================*/
void AppConnectCallWithSubject(IN RvSipCallLegMgrHandle hCallLegMgr)
{
        RvSipCallLegHandle     hCallLeg;
        RvSipMsgHandle         hMsg;
        RvSipOtherHeaderHandle hSubject;
```

```
        RvChar  *strFrom = "From:sip:user1@172.20.1.49:5060";
        RvChar  *strTo =   "To:sip:user2@172.20.1.49:5060";

        /*Creating a new call-leg.*/
        RvSipCallLegMgrCreateCallLeg (hCallLegMgr,NULL,&hCallLeg);

        /*Get the outbound message object.*/
        RvSipCallLegGetOutboundMsg(hCallLeg,&hMsg);

        /*Construct Other header in message*/
        RvSipOtherHeaderConstructInMsg(hMsg,RV_TRUE,&hSubject);

        /*Set the other header fields.*/
        RvSipOtherHeaderSetName(hSubject,"Subject");
        RvSipOtherHeaderSetValue(hSubject,"Hello");

        /*Send Invite call by calling the Make() function.*/
        printf("Connecting a call: %s -> %s\n", strFrom, strTo);
        RvSipCallLegMake(hCallLeg,strFrom,strTo);
}
/*=========================================================================*/
```

CALL-LEG RE-INVITE

Modifying an existing call to alter the session media description and settings is done by a re-INVITE procedure. A re-INVITE process can occur only after the *call-leg* received a 2xx final response for its initial INVITE request, and when there are no other pending re-INVITE processes.

RE-INVITE OBJECT

A re-INVITE object handles a re-INVITE procedure. The re-INVITE object is identified using a dedicated handle. You must supply this handle when using the Re-Invite API. RvSipCallLegInviteHandle defines a Stack re-INVITE object. For an outgoing re-INVITE you receive this handle from RvSipCallLegReInviteCreate(). For incoming re-INVITE you receive this handle from the RvSipCallLegReInviteCreatedEv() callback.

RE-INVITE CONTROL

The following API functions provide re-INVITE control:

RvSipCallLegReInviteCreate()

Creates a new re-INVITE object in a *call-leg*.

RvSipCallLegReInviteRequest()

Begins a re-INVITE procedure that alters the session media description and settings. You can call this function only after the *call-leg* received a 2xx final response for its initial INVITE request, and when there are no other re-INVITE pending processes. You can use the outbound message mechanism to set headers and a body to the outgoing re-INVITE.

Remarks: This function does not handle the call-leg Session-Timer parameters. Thus, when used during a Session Timer call, it turns off the Session Timer mechanism. Consequently, in order to keep the mechanism up, use the RvSipCallLegSessionTimerInviteRefresh() function instead.

RvSipCallLegReInviteAck()

When the SIP Stack is configured to work in a manual ACK mode, the *call-leg* will not send an ACK message after receiving a 2xx response to a re-INVITE. After receiving the 2xx response, the re-INVITE object will assume the MODIFY_REINVITE_REMOTE_ACCEPTED state. The application must use the RvSipCallLegReInviteAck() function to trigger the re-INVITE object to send the ACK.

RvSipCallLegReInviteTerminate()

Causes a re-INVITE object to be terminated without sending any messages (CANCEL or BYE). The re-INVITE will assume the TERMINATED modify state.

RvSipCallLegReInviteSetAppHandle()

Sets an application handle to the re-INVITE object.

Note Other re-INVITE functionality, such as accepting, rejecting, cancelling, and sending a provisional response for a re-INVITE is done with the basic Call-leg API functions.

RE-INVITE EVENTS

The following event is supplied for a *call-leg* re-INVITE object:

RvSipCallLegReInviteStateChangedEv()

Through this function you receive notifications of the modify state change of the re-INVITE object.

CALL-LEG RE-INVITE (MODIFY) STATE MACHINE

The Call-leg Re-Invite (Modify) state machine represents the state of a re-INVITE process between two SIP UAs. A re-INVITE process can occur only under the following conditions:

- A 2xx response was sent/received for the initial INVITE (REMOTE_ACCEPTED or CONNECTED state for an outgoing *call-leg*, ACCEPTED or CONNECTED for incoming *call-leg*).
- There is no other pending modify process. (Pending Modify means that a re-INVITE request was sent/received, but a final response was not yet sent/received).

The RvSipCallLegReInviteStateChangedEv() callback reports *call-leg* modify state changes and state change reason. The state change reason indicates how the *call-leg* reached the new modify state. The re-INVITE process associates with the following modify states:

RVSIP_CALL_LEG_MODIFY_STATE_IDLE

The initial state of the Re-INVITE state machine. Upon creation of a re-INVITE object, it assumes the IDLE modify state. It remains in this state until RvSipCallLegReInviteRequest() is called, whereupon it should move to the REINVITE_SENT modify state.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_RCVD

A re-INVITE request was received.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_RESPONSE_SENT

A response was sent for an incoming re-INVITE request.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_SENT

A re-INVITE request was sent.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_RESPONSE_RCVD

A final response was received on an outgoing re-INVITE request.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_REMOTE_ACCEPTED

The remote party accepted the re-INVITE request. This state will be reported only if the *manualAckOn2xx* configuration parameter is set to RV_TRUE. In this case, the ACK message will not be sent automatically and the application must initiate the ACK message by calling the RvSipCallLegReInviteAck() function.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_CANCELLING

A CANCEL request was sent on the re-INVITE request.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_PROCEEDING

A provisional response was received on an outgoing re-INVITE request. The Modify state machine moves to this state from the REINVITE_SENT state (meaning that this state is assumed only when the first provisional response is received).

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_CANCELLED

Upon receiving a CANCEL request in the REINVITE_RCVD state, the SIP Stack will automatically accept the CANCEL and move to the REINVITE_CANCELLED state. A 487 response will be sent automatically as a response to the INVITE request unless the SIP Stack is configured to work in manual behavior mode, in which your application will be responsible for the INVITE final response.

RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_PROCEEDING_TIMEOUT

This state is assumed only if the SIP Stack is configured to enable the INVITE_PROCEEDING_TIMEOUT state. The Modify state machine moves to this state from the REINVITE_PROCEEDING state when the provisional timer expires before receiving a final response to the re-INVITE request. In this state the application can cancel the re-INVITE request or terminate the *call-leg*.

RVSIP_CALL_LEG_MODIFY_STATE_MSG_SEND_FAILURE

This state is assumed only if the SIP Stack is compiled with the Enhanced DNS feature. The Modify state machine moves to this state from the REINVITE_SENT state when the re-INVITE request receives a network error, 503 response, or timeout on the request. In this state your application can decide to:

- Continue DNS—send an ACK request if the failure is because of a 503 response, and try to send the request to the next address in the *transaction* DNS list. For more information see the Working with DNS chapter.
- Give up—send an ACK request if the failure is because of a 503 response, and terminate the re-INVITE object.
- Terminate the re-INVITE object.

RVSIP_CALL_LEG_MODIFY_STATE_ACK_RCVD

An ACK was received for a re-INVITE.

RVSIP_CALL_LEG_MODIFY_STATE_ACK_SENT

An ACK was sent for a re-INVITE.

RVSIP_CALL_LEG_MODIFY_STATE_TERMINATED

The final state of the re-INVITE object. Upon reaching this state, you can no longer reference the re-INVITE object.

Basic Call-leg Re-Invite (Modify) State Machine

The following diagrams illustrate the Call-leg Re-Invite (Modify) state machine. Figure 6-5 illustrates the basic state machine. Figure 6-6 illustrates the state machine with message send failure and Figure 6-7 illustrates the state machine with a network error or timeout.

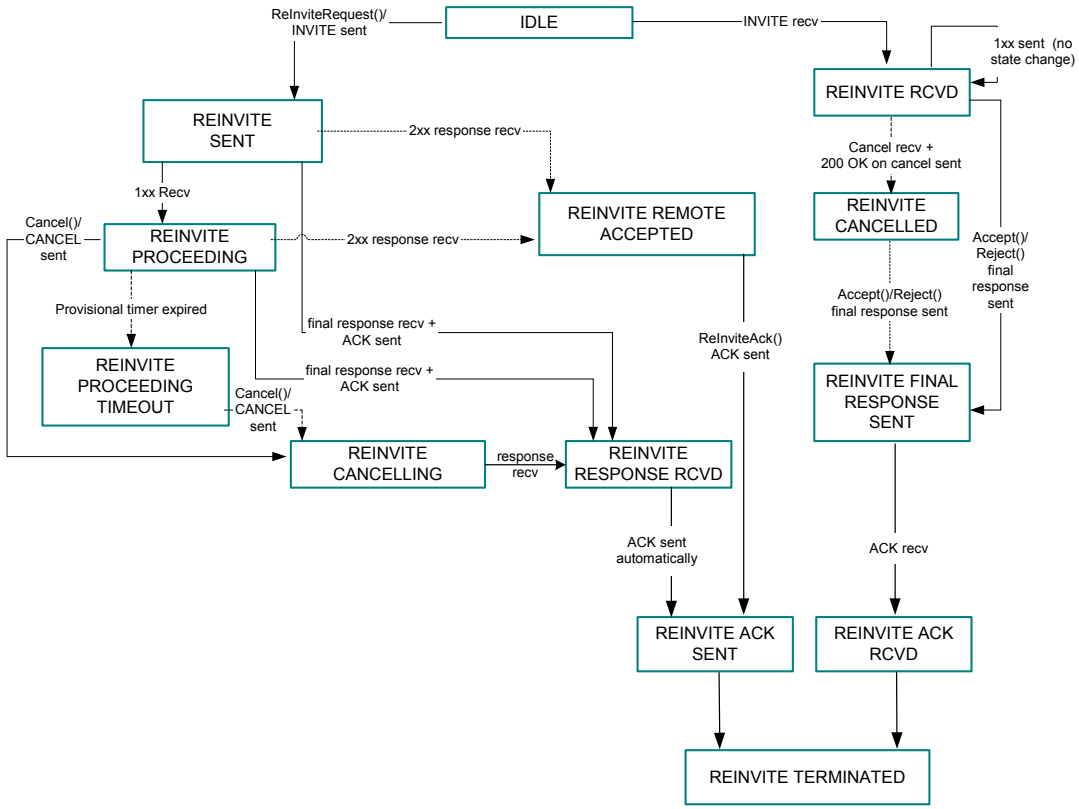


Figure 6-5 *Call-leg Re-Invite (Modify) State Machine*

MSG-SEND-FAILURE STATE MACHINE

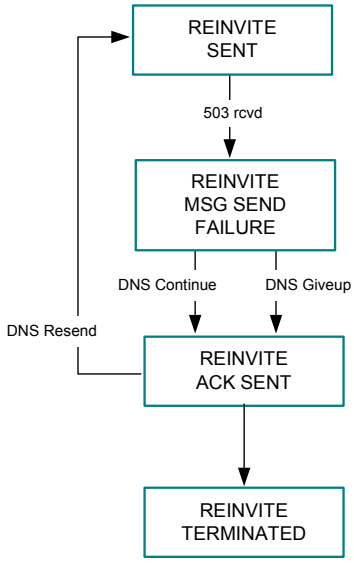


Figure 6-6 *Call-leg Re-INVITE Message Send Failure State Machine*

MSG-SEND-FAILURE STATE MACHINE WITH NETWORK ERROR OR TIMEOUT

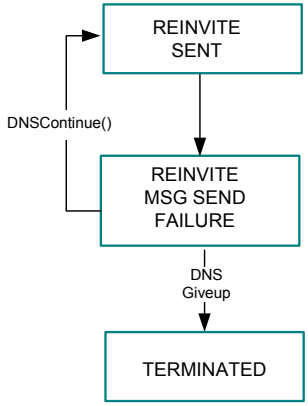


Figure 6-7 *Call-leg Re-INVITE Message Sent Failure State Machine with Network Error or Timeout*

Authenticating a Re-INVITE

When a re-INVITE procedure is ended with a 401 or 407 final response, the application needs to re-send the re-INVITE with authentication information. In this case, the RVSIP_CALL_LEG_MODIFY_STATE_ACK_SENT state will have the RVSIP_CALL_LEG_REASON_AUTH_NEEDED reason. The application should repeat the process of sending the re-INVITE, (prepare the outbound message, and call the RvSipCallLegReInviteCreate() and RvSipCallLegReInviteRequest() functions again). The *call-leg* will automatically add the needed authentication information (credentials) to the new outgoing re-INVITE, and the request will be sent to the remote party. For more information on the authentication process, see the Authentication chapter.

Call-leg PRACK State Machine

The Call-leg PRACK state machine represents the state of a PRACK process between two SIP User Agents. A PRACK process starts when the UAS triggers a reliable provisional response in the OFFERING or REINVITE_RCVD states. The RvSipCallLegPrackStateChangedEv() callback reports *call-leg* PRACK state changes and state change reasons.

The application can work in automatic or manual PRACK modes. In automatic mode, a PRACK request will be sent automatically on receipt of reliable provisional response. When a PRACK request is received, the SIP Stack will automatically send the response. In manual mode, the application should use the PRACK state machine and the Call-leg PRACK API to trigger the PRACK request and response. To work in manual PRACK mode, you should set the *manualPrack* configuration parameter to RV_TRUE when the SIP Stack initializes.

The *call-leg* associates with the following PRACK states:

RVSIP_CALL_LEG_PRACK_STATE_UNDEFINED

There is currently no active PRACK process.

RVSIP_CALL_LEG_PRACK_STATE_REL_PROV_RESPONSE_RCVD

A reliable provisional response was received.

RVSIP_CALL_LEG_PRACK_STATE_PRACK_SENT

A PRACK request was sent.

RVSIP_CALL_LEG_PRACK_STATE_PRACK_FINAL_RESPONSE_RCVD

A PRACK final response was received.

RVSIP_CALL_LEG_PRACK_STATE_PRACK_RCVD

A PRACK request was received.

RVSIP_CALL_LEG_PRACK_STATE_PRACK_FINAL_RESPONSE_SENT

A PRACK final response was sent.

CALL-LEG TRANSACTIONS

A *call-leg transaction* is a *transaction* that is sent in the context of a *call-leg* but does not change the *call-leg* state. Such a *transaction* uses the *call-leg* identifiers (To, From and Call-ID), an increased CSeq, Route list, and Authentication information when it exists. An example of a *call-leg transaction* is the INFO *transaction* sent in the context of a *call-leg*.

The Call-leg API contains a set of functions and function callbacks that enables you to handle *transactions* inside a *call-leg*.

CALL-LEG TRANSACTION CONTROL

The following API functions provide *call-leg transaction* control:

RvSipCallLegTranscCreate()

Creates a new general *transaction* that is related to the supplied *call-leg*. The *transaction* will have the *call-leg* characteristics, such as To header, From header, Call-ID, and local and outbound addresses. The application can define an application handle to the *transaction* and supply it to the Stack when calling this function. The application handle will be supplied back to the application when the RvSipCallLegTranscStateChangedEv() callback is called.

RvSipCallLegTranscRequest()

Sends a Request message with a given method using a given transaction. You can use this function at any *call-leg* state for sending requests, such as INFO. The request will have the To header, From header and Call-ID of the *call-leg*, and will be sent with a correct CSeq step. The request will be record routed if needed.

Note Before calling this function you should create a new *call-leg transaction* using the RvSipCallLegTranscCreate() function. You can then use the *transaction* outbound message mechanism to add headers and a body to the outgoing request. If you supply the function with a NULL transaction, the SIP Stack will create a new *call-leg transaction* automatically. In this case you will not be able to use the *transaction* outbound message or to replace handles with the transaction.

Note A *transaction* that was supplied by the application will not be terminated in the case of failure. It is the responsibility of the application to terminate the *transaction* using the RvSipCallLegTranscTerminated() function.

RvSipCallLegTranscResponse()

Sends a response to a *call-leg* related transaction. When a *call-leg* receives a general request, such as INFO (but not BYE or PRACK), the *call-leg* first notifies the application that a new *call-leg transaction* was created using the RvSipCallLegTranscCreatedEv() callback. At this stage, the application can specify whether or not it wishes to handle the transaction, and can also replace handles with the Stack. The *call-leg* will then notify the application about the new *transaction* state, General-Request-Rcvd, using theRvSipCallLegTranscStateChangedEv() callback.

In this state, the application should use the RvSipCallLegTranscResponse() function to send a response to the request.

RvSipCallLegTranscTerminate()

Terminates a *transaction* related to a specified *call-leg*.

CALL-LEG TRANSACTION EVENTS

The following events are supplied with the Call-leg API:

RvSipCallLegTranscCreatedEv()

Notifies the application that a new general *transaction* (other than BYE or PRACK) was created and relates to the specified *call-leg*. In this callback, the application can replace handles with the *call-leg transaction* and specify whether it wishes to handle the incoming request. If so, the application will be informed of the *transaction* states where it will have to respond to the request.

If the application indicates that it does not wish to handle the request, the *call-leg* will apply its default behavior according to the request method. For example, requests such as REFER and SUBSCRIBE will be handled using dedicated state machines and callbacks. Unknown requests will be responded to with 501.

RvSipCallLegTranscStateChangedEv()

Notifies the application that the state of a general *transaction* that belongs to the specified *call-leg* has changed. When the state indicated that a request was received, the application should use the RvSipCallLegTranscResponse() function and respond to the request.

CALL-LEG TRANSACTION STATE MACHINE

The Call-leg Transaction state machine represents the state of a general *transaction* that belongs to a *call-leg*. The RvSipCallLegTranscStateChangedEv() callback reports *call-leg transaction* state changes and state change reason. The state change reason indicates how the *transaction* reached the new state.

A *call-leg transaction* associates with the following states:

RVSIP_CALL_LEG_TRANSC_STATE_IDLE

The IDLE state is the initial state of a *call-leg* transaction.

RVSIP_CALL_LEG_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD

The REQUEST_RCVD state indicates that a general request was received. The application should use the RvSipCallLegTranscResponse() function to respond to the transaction.

RVSIP_CALL_LEG_TRANSC_STATE_SERVER_GEN_FINAL_RESPONSE_SENT

After calling RvSipCallLegTranscResponse(), a final response is sent and the *transaction* assumes the FINAL_RESPONSE_SENT state.

RVSIP_CALL_LEG_TRANSC_STATE_CLIENT_GEN_REQUEST_SENT

After the application created a new *call-leg* transaction, it should call the RvSipCallLegTranscRequest() function with a specific method. This function will cause a request to be sent and the *transaction* will assume the GENERAL_REQUEST_SENT state.

RVSIP_CALL_LEG_TRANSC_STATE_CLIENT_GEN_PROCEEDING

A *call-leg transaction* that received the first provisional response will assume the PROCEEDING state.

RVSIP_CALL_LEG_TRANSC_STATE_CLIENT_GEN_FINAL_RESPONSE_RCVD

After a final response is received on a *call-leg* transaction, the *transaction* assumes the FINAL_RESPONSE_RCVD state.

RVSIP_CALL_LEG_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE

This state is assumed only if the SIP Stack works with the Enhanced DNS feature. The *call-leg transaction* moves to this state when it fails to send a request. A failure is defined as a timeout, network error or a request that was responded to with 503.

In this state the application can:

- Continue DNS—try to send the request to the next address in the *transaction* DNS list. For more information see the Working with DNS chapter.
- Give Up—terminate the *call-leg* transaction.

RVSIP_CALL_LEG_TRANSC_STATE_TERMINATED

This state is the final *call-leg transaction* state. When the *call-leg transaction* is terminated, it assumes the TERMINATED state. The application can no longer reference the *transaction*.

TRANSACTION STATE MACHINE

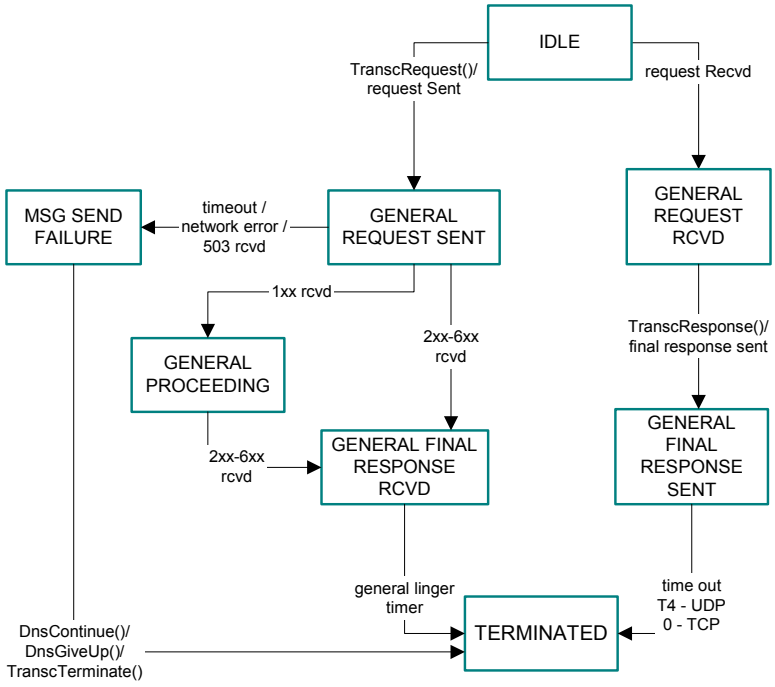


Figure 6-8 *Call-leg Transaction State Machine*

Sample Code

The following sample code demonstrates how to send an INFO request within the context of a *call-leg*.

```
/*============================================================*/
RvStatus AppSendInfo(IN RvSipCallLegHandle hCallLeg)
{
    RvStatus             rv;
    RvSipTranscHandle    hNewTransc;

    rv = RvSipCallLegTranscCreate(hCallLeg,NULL,&hNewTransc);
    if(rv != RV_OK)
    {
        printf("failed to create a new call-leg transaction\n");
        return rv;
    }

    rv = RvSipCallLegTranscRequest(hCallLeg,"INFO",&hNewTransc);
    if(rv != RV_OK)
    {
        printf("failed to send INFO\n");
        return rv;
    }
    return RV_OK;
}
/*============================================================*/
```

Sample Code

The following sample code demonstrates the application implementation of the RvSipCallLegTranscCreatedEv() callback. In this implementation, the application specifies that it wishes to handle only INFO requests and instructs the SIP Stack to handle all other requests. The application does not exchange handles with the Stack.

```
/*=========================================================================*/
void RVCALLCONV  AppCallLegTranscCreatedEvHandler(
               IN  RvSipCallLegHandle       hCallLeg,
               IN  RvSipAppCallLegHandle    hAppCallLeg,
               IN  RvSipTranscHandle        hTransc,
               OUT RvSipAppTranscHandle     *hAppTransc,
               OUT RvBool                   *bAppHandleTransc)
{
     RvChar method[50];

     RvSipTransactionGetMethodStr(hTransc,50,method);

     /*Handles only INFO requests. Leaves the rest of the requests for the
     Stack to handle.*/

     if(strcmp(method,"INFO")==0)
     {
           *bAppHandleTransc = RV_TRUE;
      }
      else
      {
           *bAppHandleTransc = RV_FALSE;
      }
      *hAppTransc = NULL;

}
/*=========================================================================================*/
```

Sample Code

The following sample code demonstrates an application implementation of the RvSipCallLegTranscStateChangedEv() event. As indicated in the *transaction* created callback, the state changed callback will be called only for an INFO request. In this sample, the application accepts the request.

```
/*=========================================================================*/
static void RVCALLCONV AppCallLegTranscStateChangedEvHandler(
           IN  RvSipCallLegHandle                hCallLeg,
           IN  RvSipAppCallLegHandle             hAppCallLeg,
           IN  RvSipTranscHandle                 hTransc,
           IN  RvSipAppTranscHandle              hAppTransc,
           IN  RvSipCallLegTranscState           eTranscState,
           IN  RvSipTransactionStateChangeReason eReason)
{
   switch(eTranscState)
   {
   case RVSIP_CALL_LEG_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD:
        RvSipCallLegTranscResponse(hCallLeg,hTransc,200);
        break;
   default:
        break;
    }
}
/*=========================================================================*/
```

AUTHENTICATING A CALL-LEG TRANSACTION

A *call-leg transaction* can receive a 401 or 407 response to its request. To send the request again with authentication information, the application should do the following:

1. Create a new *call-leg* transaction.
2. Fill the outbound message of this *transaction* with the same information that was set to the unauthenticated transaction.
3. Call the RvSipCallLegTranscRequest() function with the same method as the unauthenticated transaction.

The *call-leg* will automatically add the needed authentication information (credentials) to the outgoing request, and the request will be sent. The *call-leg* will also automatically add these credential to further outgoing requests.

For more information on the authentication process see the Authentication chapter.

CALL-LEG FORKING SUPPORT

A proxy might fork an initial INVITE request. As a result, the client may receive multiple responses from multiple User Agent Servers (UASs) for the same initial request. (According to the forwarding rules of proxies, a proxy must forward back all 1xx provisional response messages. The proxy must also forward back all 2xx final response messages, or a single non-2xx final response message).

RFC 3261 defines that when multiple 1xx or 2xx responses are received from different remote UAs (because the INVITE forked), each 2xx establishes a different dialog. This section describes the forking support for *call-legs* in the SIP Stack.

TERMINOLOGY

- Original *call-leg*—The *call-leg* that sent the original INVITE request. The original *call-leg* owns the invite *transaction* It will handle the first response that is received from the proxy.
- Forked *call-leg*—A *call-leg* that was created by a response message. After the initial response was handled by the original *call-leg*, further responses that were received from other UA Servers due to a forking proxy will create new *call-legs*. These *call-legs* are referred to as forked *call-legs*. A forked *call-leg* might be an early-dialog (in the PROCEEDING state), or a confirmed dialog (in the CONNECTED state). The forked *call-leg* does not own an invite transaction.

OVERVIEW OF OPERATION

A User Agent Client (UAC) sends an INVITE request as usual. It creates a *call-leg*, sets the session parameters, and sends the initial INVITE request. This *call-leg* is identified by its Call-ID, and From tag values. The To tag value is empty, until a response message (provisional or final) will be received from a UAS.

HANDLING OF MULTIPLE 1XX RESPONSES DUE TO FORKING

Zero, one or multiple provisional responses may arrive before one or more final responses are received. Each response is distinguished by the tag parameter in the To header field. Provisional responses for an INVITE request create “early dialogs”. The first 1xx response will be mapped to the original *call-leg*, and will update the To tag value of that *call-leg*. Further responses from the same UAS server will contain the same to tag parameter, and therefore will be mapped to the original *call-leg*.

Any 1xx response sent by a different user-agent-server contains a different To tag value. Therefore, it creates a new forked *call-leg* in the PROCEEDING state. If a matching forked *call-leg* was already created (by a previous 1xx response), the response message will not create a new *call-leg*, but will be mapped to the already existing forked *call-leg*.

HANDLING OF MULTIPLE 2XX RESPONSES DUE TO FORKING

Multiple 2xx responses may arrive at the UAC for a single INVITE request due to a forking proxy. If the dialog identifier in the 2xx response matches the dialog identifier of an existing *call-leg* (original or forked), the 2xx response is mapped to this *call-leg*. Otherwise, a new forked *call-leg* is created, and the 2xx response is mapped to the new *call-leg*.

In both cases, the 2xx response updates the state of the *call-leg* to REMOTE-ACCEPTED. After sending the ACK request, this *call-leg* state will be updated to CONNECTED.

HANDLING OF A NON-2XX FINAL RESPONSE

According to proxy rules, a single non-2xx final response may be received for the INVITE response. The non-2xx final response is always mapped to the original *call-leg*, regardless of which To tag it has. (This is because the ACK on a non-2xx response should be sent by the original transaction, and the original *transaction* is related to the original *call-leg*).

FORKED CALL-LEG TERMINATION

According to RFC 3261, The UAC considers the INVITE *transaction* completed 64*T1 seconds after the reception of the first 2xx response. At this point, all the early dialogs that have not transitioned to established dialogs are terminated. Once the INVITE *transaction* is considered completed, no more new 2xx responses are expected to arrive. All early dialogs are considered terminated upon reception of the non-2xx final response.

For precise implementation, it is recommended that on reception of the first 2xx, the application will set a timer to 64*T1 seconds, and when this timer expires, the application will terminate all non-established calls created by the same initial request. However, the application can also use the *call-leg* Forked-1xx-Timer which is a timer that is set for every forked *call-leg* created by an incoming 1xx response. If a 2xx response is received on this *call-leg*, the timer is released. However, if a 2xx is not received, the *call-leg* will be terminated on timer expiration. As a result, the application can be sure that all forked *call-legs* that did not receive a final response will be terminated.

(Note that the original *call-leg* does not have the forked-1xx-timer. This *call-leg* owns the invite transaction. If the original *call-leg* does not receive a final response, the *transaction* timer will expire, and the *transaction* termination will cause the *call-leg* to terminate as well.)

The application can supply the Forked-1xx-Timer timeout value in the SIP Stack configuration. This value will apply to all forked *call-legs*. The application can also change the timer value for each *call-leg* using the *call-leg* API. The application may also disable this timer by setting its value to 0.

A non-2xx final response should also terminate all *call-legs* created by the same initial request. Again, it is recommended that application will terminate all these *call-legs* when the non-2xx response is received. However, here again the application can relay on the forked-1xx-timer that will terminate all *call-legs* on its expiration.

Sending CANCEL

Only the original *call-leg* may cancel the initial request, since the original *call-leg* holds the initial invite transaction. Calling the Cancel() function on a forked *call-leg* returns an illegal-action error.

Call-leg Forking Support Events

The Call-leg API includes the following event for forking-support implementation.

RvSipCallLegCreatedDueToForkingEv()

A provisional or final response that is received for an initial INVITE may create a forked *call-leg*. This event informs the application of the creation of a new forked *call-leg*, and exchanges handle with the application.

If the application does not wish to handle this forked *call-leg*, it can indicate that the SIP Stack should terminate the new forked *call-leg*. In this case, the new *call-leg* will be destructed immediately. Otherwise the new forked *call-leg* will handle the new response message, update its state machine, send the ACK response if needed, and call to all the regular *call-leg* callback functions.

Call-leg Forking Support API

The Call-leg API includes a set of functions dedicated to forking-support implementation.

RvSipCallLegGetOriginalCallLeg()

Returns the handle to the original *call-leg* related to a given forked *call-leg*. If the given *call-leg* is an original *call-leg*, the *call-leg* will return its own handle.

RvSipCallLegSetForkingEnabledFlag(), RvSipCallLegGetForkingEnabledFlag()

Functions for setting/getting the *call-leg* forking-enable-flag. The forking-enabled-flag defines the *call-leg* behavior on receiving multiple responses due to proxy forking. If this flag is set to TRUE in the original *call-leg*, a new forked *call-leg* will be created for every 1xx/2xx response with a different, new To tag. If this flag is set to FALSE in the original *call-leg*, only first response message will be mapped to the original *call-leg* and update its To tag parameter. Afterwards, only 1xx/2xx responses with this To tag will be mapped to the *call-leg*. All other 1xx/2xx responses will be ignored. All 3xx-6xx responses will be mapped to this *call-leg* too. The default value for the forking-enabled-flag is RV_TRUE.

RvSipCallLegSetForked1xxTimerTimeout()

Sets the timeout value for the forked-1xx-timer. The forked-1xx-timer is set by a forked *call-leg* after receiving the first 1xx response. This timer is released when the *call-leg* receives a 2xx response. If the timer expires before 2xx reception, the *call-leg* is terminated. This timeout value defines how long the *call-leg* will wait for a 2xx response before termination.

This function enables application to define the timeout value for this timer, for a specific *call-leg*. (By default the timer timeout value is taken from SIP Stack configuration).

Call-leg Forking Support Process Flow

Figure 6-9 illustrates a typical forking scenario:

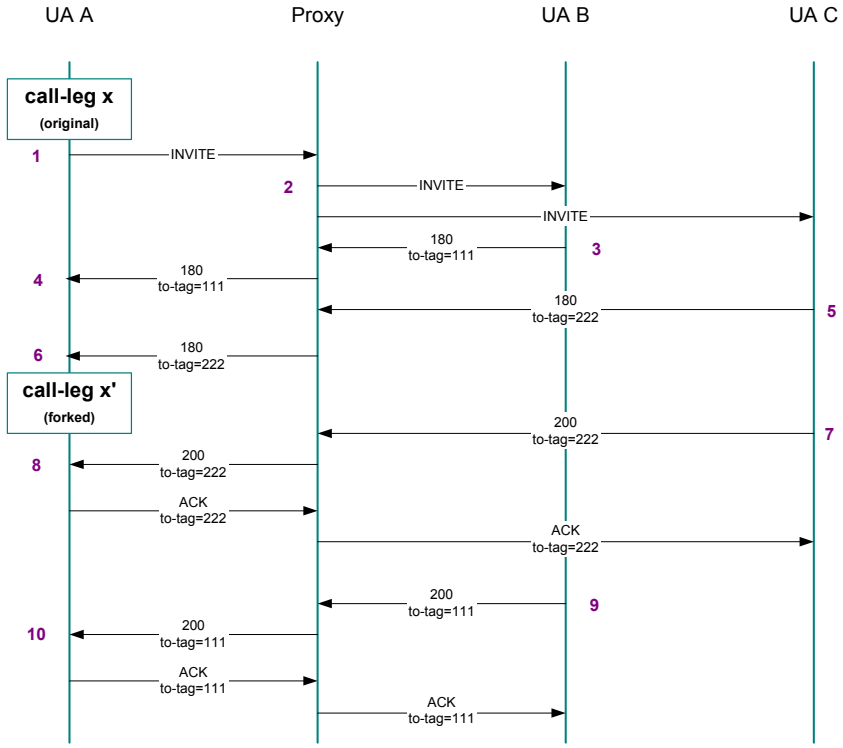


Figure 6-9 *Forking Process Call Flow*

The following is a description of the process of handling responses:

1. User Agent A creates a *call-leg* x, and uses it to send an initial INVITE request via proxy.
2. The proxy forks the INVITE request to two UASs, B and C.
3. UA B sends a 180 response, with the To tag value, 111. The proxy forwards this provisional response to the UAC A.
4. UA A updates *call-leg* x with the To tag, 111.
5. UA C sends a 180 response, with a To tag value of 222. The proxy forwards this provisional response to UA A.
6. UA A searches for a matching *call-leg* for the 180-response message. Such a *call-leg* does not exist, so UA A creates a new

forked *call-leg* x' and supplies the 180 message to the new *call-leg* for handling. The state of *call-leg* x' is updated to PROCEEDING, and its To tag value is updated to 222. *Call-leg* x' also sets the forked-1xx-timer.

7. UA C sends a 200 response, with the To tag value of 222. The proxy forwards this final response to UAC A.
8. UA A maps the 200 response to *call-leg* x'. *Call-leg* x' reset the forked-1xx-timer, creates an ACK *transmitter*, and uses it to send an ACK request. The state of *call-leg* x' is updated to CONNECTED.
9. UA B sends a 200 response, with the To tag value, 111. The proxy forwards this final response to UA A.
10. UA A maps the 200 response to *call-leg* x. *Call-leg* x sends an ACK request, and updates the state to CONNECTED.

At this point, the UAC has two connected *call-legs*, x and x', both of which were created by a single INVITE request. The application can now use both *call-legs* as usual.

Note The SIP Stack can create a forked *call-leg* only while the original *call-leg* exists, since it has to copy the parameters from the original *call-leg* to the new forked *call-leg*. If, for some reason, the original *call-leg* was terminated, a forked *call-leg* will not be created, and the response message will be discarded.

CALL-LEG FORKING SUPPORT CONFIGURATION PARAMETERS

The *call-leg* forking-support configuration parameters are as follows:

bEnableForking

Enables the forking feature in the SIP Stack. If this parameter is set to RV_FALSE, all 1xx and 2xx response messages will be mapped to the original *call-leg*. Every 1xx response and first 2xx response will update the *call-leg* To tag (the *call-leg* To tag value will be changed again and again). The 3xx-6xx response message will be handled as usual.

If this parameter is set to RV_TRUE, a new forked *call-leg* will be created for response messages with different To tags. The default value for this parameter is RV_FALSE.

forked1xxTimerTimeout

Defines the timeout value for the forked-1xx-timer. The forked-1xx-timer is set by a forked *call-leg* that received the first 1xx response. If a 2xx response is received on this *call-leg*, the timer is released. However, if 2xx is not received, the *call-leg* will be terminated on timer expiration. This timeout value defines how long the *call-leg* will wait for a 2xx response before termination.

Additional Functionality Of Call-leg Layer

The following section describes the advanced functionality that the Call-leg API supplies.

Call-leg Merging Functionality

A proxy may fork an initial INVITE request to several different proxies. As a result, the UAS may receive several request messages, with the same request identifier (Call-ID, From tag, To tag and Cseq), but with a different via-branch. Since the branch parameter in the Via header is different, a new *transaction* will be created for each request. These server *transactions*, except the first one, are called “nested transaction”.

Figure 6-10 illustrates the message flow that causes nested *transactions* to be created by a single request message of the client.

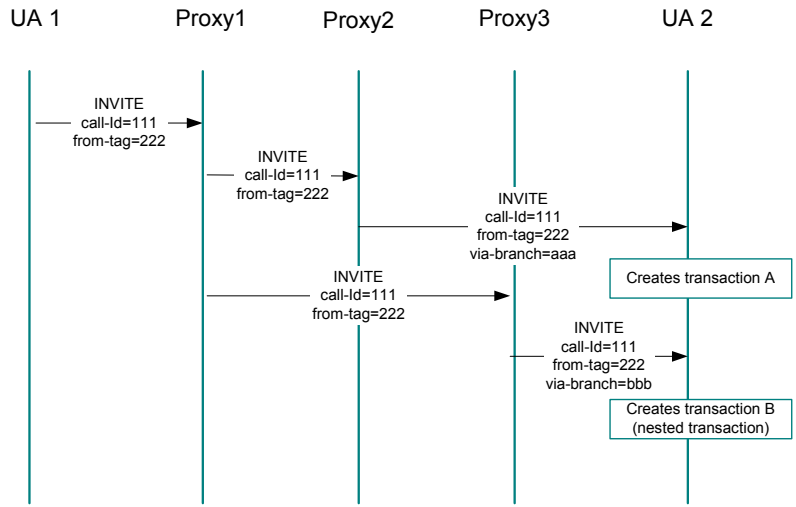


Figure 6-10 *Message Flow of Nested Transactions*

If the SIP Stack is configured to enable merging support, only the first request will create a transaction. All other requests will be rejected with a 482 response. Otherwise, a new server *transaction* will be created for each request. For more information, see Transaction Merging Support in the the Working with Transactions chapter.

Since all the new *transactions* have the same request identifiers, the first *transaction* will create a new *call-leg* and all other *transactions* will also be mapped to this *call-leg* for handling. The *call-leg* default behavior on receiving nested initial INVITE *transactions* for the same *call-leg* is to handle only the first transaction, and to reject all others with a 400 response.

Applications that want to create a separate *call-leg* for every nested *transaction* may use the callback function, RvSipCallLegNestedInitialReqRcvdEv(). If this callback is implemented, *call-leg* behavior will be as follows:

First transaction case

1. Create a *call-leg*.
2. Insert this *call-leg* to the call hash table. Note that this *call-leg* has no To tag parameter yet.
3. Change *call-leg* state to OFFERING.

Second transaction case

1. Call the RvSipCallLegNestedInitialReqRcvdEv() callback function.
2. If the application chooses not to create a new *call-leg*, reject with 400.
3. If the application chooses to create a new *call-leg*, create the new *call-leg* to handle the nested Invite transaction. Note that this *call-leg* is not in the hash table yet, because this new *call-leg* has the same call identifiers as the first *call-leg*.
4. Change *call-leg* state to OFFERING.
5. When the application sends 1xx or a final response to this *call-leg*, the *call-leg* will be inserted into the *call-legs* hash table, because now it has a To tag parameter that identifies it from the first *call-leg*.

7

WORKING WITH TRANSACTIONS

INTRODUCTION

A SIP *transaction* comprises all messages, from the first request sent by the client to the server to a final (non-1xx) response to the request sent by the server to the client.

The Transaction API of the SIP Stack contains a set of functions and function callbacks that can be used for two purposes. The first is for handling *transactions* that are related to the User Agent (UA) and *not* related to a *call-leg*. An example of such a *transaction* is OPTIONS.

Using the Transaction API, you can create and initialize a *transaction* and control a *transaction* according to the *transaction* state. You can create a *transaction* with any method (except CANCEL and ACK that have a dedicated API).

Note In order to send a *transaction* that is related to a *call-leg*, you should use the Call-leg API. For more information, see the Working with Call-legs (Dialogs) chapter.

The second purpose of the Transaction API is for writing a SIP server. Using the Transaction API, you can implement a Proxy server, Redirect server and Registrar. You can receive incoming requests and decide whether to redirect or proxy the request according to the request method and the *transaction* state. The Transaction API contains a set of functions and states dedicated specifically for the purpose of writing a Proxy server. Using these functions, you can fully implement both stateless and stateful proxies.

Transaction Entities

The Transaction API relates to the following two entities:

- Transaction (*transaction*)
- Transaction Manager (*TransactionMgr*)

Transaction

A *transaction* represents a SIP *transaction* as defined in RFC 3261. The *transaction* consists of a request (and its retransmissions) together with the response triggered by that request. Your application can initiate *transactions*, send requests and respond to incoming requests using the Transaction API.

A *transaction* is a stateful object, which can assume any state from a set defined in the Transaction API. The Transaction state machine represents the state of the *transaction* between the client and the server.

An Invite *transaction* also includes the ACK request when the final response to the Invite request is non-2xx. The ACK of a 2xx response is not part of the *transaction* and is handled separately.

Note This behavior is new from version 4.0. To keep the previous behavior, where the ACK was always part of the Invite transaction, you should set the *bOldInviteHandling* configuration parameter to RV_TRUE.

Transaction Manager

The *TransactionMgr* manages the collection of all *transactions*. The *TransactionMgr* is mainly used for creating new *transactions*.

Working with Handles

All *transactions* and the *TransactionMgr* are identified using handles. You must supply these handles when using the Transaction API.

RvSipTranscMgrHandle defines the *TransactionMgr* handle. You receive this handle by calling RvSipStackGetTransactionMgrHandle().

RvSipTranscHandle defines a *transaction* handle. For client *transactions*, you receive the *transaction* handle when creating a *transaction* with RvSipTranscMgrCreateTransaction(). For Server *transactions*, you receive the *transaction* handle from the RvSipTransactionCreatedEv() callback.

Transaction API

The Transaction API contains a set of functions and function callbacks that allow you to set or examine *transaction* parameters and to control a *transaction* request or response.

TRANSACTION PARAMETERS

You can set or examine *transaction* parameters via *transaction* Set and Get API functions. The following parameters are available:

To Header, From Header, Call-ID and CSeq

When creating a client transaction, you must set the To and From headers of the transaction. The Call-ID and CSeq are optional. You can either set them, or the SIP Stack will generate them for you.

Method

Specifies the method of the SIP request. For client *transactions*, you supply the method when calling the RvSipTransactionRequest() API function. For server *transactions* you can only access the method parameter with a Get function and the parameter is not modifiable.

Local Address

Defines the address from where the *transaction* will send the request (the network card). This is also the address that will be placed in the top Via header of a Request message. If the local address is not set, the *transaction* will use a default local address according to the SIP Stack configuration.

Outbound Details

Addressing details of an outbound proxy that the *transaction* should use. The outbound address is used only if the *transaction* sends a Request message. In this case, the *transaction* will use the outbound address as a remote address and the Request-URI will be ignored. The outbound address of the *transaction* is ignored if the request contains a Route header, or if the RvSipTransactionIgnoreOutboundProxy() function was called. You can force the usage of the outbound proxy regardless of the message content by calling the RvSipTransactionSetForceOutboundAddrFrag() function.

Received Message

The last message (Request or Response) that was received by the transaction. You can get this message only in the context of the *transaction* state-changed callback function when the new state indicates that the *transaction* received a new message.

Outbound Message

The outbound message is a handle to a *message* that the *transaction* will use for the next outgoing message. Before calling an API function that causes a message to be sent, the application can get the outbound *message* and add headers and a body.

Response Code

A 3-digit integer status code that indicates the outcome of the attempt to understand and satisfy a request. A server *transaction* will supply the response code when calling the RvSipTransactionResponse() API function. A client *transaction* can only access the response code parameter with a Get function and the parameter is not modifiable.

State

Represents the state of the *transaction* between the client and the server. You can only access the state parameter with a Get function and it is not modifiable.

Top Via Branch

The Top Via Branch is part of the *transaction* key and is used to match responses to their requests. When sending an outgoing request, the *transaction* automatically adds a top Via header including a Via branch to the Request message according to the rules defined in the protocol. If you set the Top Via branch parameter before calling the RvSipTransactionRequest() function, the *transaction* adds this branch to the request top Via header. This parameter has a Set function only.

Request URI

Indicates the Request-URI of the initial request of the transaction. You can only access the Request-URI parameter with a Get function and the parameter is not modifiable. You supply the Request-URI of client *transaction* when calling the RvSipTransactionRequest() API function.

Cancel Transaction/Cancelled Transaction

Associate a Cancel *transaction* with the *transaction* it cancels and vice versa. You can retrieve the handle to the INVITE *transaction* that is being cancelled from a CANCEL transaction. You can also retrieve the handle to the CANCEL *transaction* that cancels this INVITE from an INVITE transaction.

Transaction Timers and Retransmissions Count

According to RFC 3261, the *transaction* has to set different timers during its life cycle and perform different actions when the timers expire. For example, after sending a final response, the *transaction* has to set a timer to the value of 32,000 msec. When this timer expires, the *transaction* must terminate. The values of the *transaction* timers are taken from the Stack configuration and are determined upon initialization. The application can use a Set function and change the different timer values of the transaction. The application can also control the number of retransmissions performed by the transaction.

Persistency Definition and Used Connection

When working with TCP, the application can instruct the *transaction* to try and locate a suitable open *connection* in the *connection* hash before constructing a new *connection*. The application can also instruct the INVITE *transaction* to try and send both INVITE and ACK on the same *connection*. For more information on persistency level and Persistent Connection API functions, see Persistent Connection Handling of the Working with the Transport Layer chapter.

Current Local Address

For client *transactions*, the local address that was actually used for sending the request. For Server *transactions*, the local address on which the request was received. The *transaction* will always send the response from the same local address on which the request was received.

Current Destination Address

The address to which the *transaction* is about to send a message. This address is available only in the context of the RvSipTransactionFinalDestResolvedEv() callback.

Received From Address

The address from which the *transaction* has received the last incoming message (request or response).

Is UAC

Transactions have a client side and a server side. The client side is known as a client *transaction* and the server side as a server transaction. The client *transaction* sends the request, and the server *transaction* sends the response. The Is UAC parameter indicates whether or not the *transaction* is a client transaction.

New Header Handles

Some of the *transaction* fields are message parts. For example, the To header field is a Party header object. Before setting a parameter of this type in the transaction, you should request a new handle for the parameter from the transaction. After initializing the message part, you can set it back in the transaction.

Transmitter

Each *transaction* holds a single *transmitter* and uses it to send SIP *messages* and message retransmissions. The *transaction* creates its *transmitter* upon initialization and terminates it only upon destruction. The *transmitter* is responsible for all message sending activities, including address resolution, via handling, and the actual message sending. The application can get the *transmitter* from the *transaction* in the RvSipTransactionFinalDestResolvedEv(). Using the Transmitter API, the application can manipulate the DNS list before the message is sent. For more information see the Working with Transmitters chapter.

Transaction Control

The following API functions provide *transaction* control:

RvSipTransactionMake()

After creating a transaction, you can use this function to set the To, From and CSeq parameters in the *transaction* and send a request with a specified method and a given Request-URI. All parameters given to this function are in textual format.

RvSipTransactionRequest()

After creating a *transaction* and setting the To and From headers and CSeq sequence number, you can use this function for generating and sending a request with a specified method and a given Request-URI.

RvSipTransactionRespond()

Use this function to send a provisional or final response to a *transaction* that received an incoming request. You can use this function with any response code and reason phrase.

RvSipTransactionRespondReliable()

Use this function in the INVITE-REQUEST-RECEIVED state to send a reliable provisional response. You can use this function with response codes between 101 and 199 with any reason phrase. After sending the reliable provisional response, the *transaction* assumes the INVITE_RELIABLE_PROVISIONAL_SENT state. You can only send reliable provisional responses on Invite *transactions*. This function should be used only if the supported header list of the SIP Stack configuration includes the 100rel option tag.

RvSipTransactionCancel()

Use this function to cancel a *transaction* that reached the PROCEEDING state. You can use this function only on INVITE client *transactions*. You supply this function with the *transaction* you wish to cancel. A CANCEL *transaction* will be created and a CANCEL request will be sent to the remote party. The application is informed of the new CANCEL *transaction* using the RvSipTransactionInternalCreated() callback function.

RvSipTransactionAck()

Use this function to send an ACK request on a given INVITE transaction. This function can only be called on INVITE client *transactions* that reached the INVITE_FINAL_RESPONSE_RECEIVED state, after receiving a non-2xx final response.

Remark: ACK on a 2xx response should be sent as a stand-alone message using the Transmitter API.

RvSipTransactionTerminate()

Use this function to cause an immediate shut-down of the transaction. A *transaction* is a self-terminated object. The termination of a *transaction* depends on the *transaction* state and timer configurations. By using this function, the application can terminate a *transaction* before its normal termination. After calling this function, the *transaction* will assume the TERMINATED state.

Proxy Transaction Control

The Transaction API contains a set of functions dedicated to the implementation of a Proxy server. Using these functions, you can proxy requests and responses between two UAs.

RvSipTransactionSetKeyFromMsg()

Initializes the *transaction* key from a given message. The following fields will be set in the transaction:

- To and From (including tags)
- Call-ID
- CSeq step

Before calling this function you should call RvSipTranscMgrCreateTransaction() to create a new *transaction*. This function will normally be used by proxy implementations. Before a proxy server can proxy a request it should:

1. Create a new client *transaction* with RvSipTranscMgrCreateTransaction().
2. Get the received message from the server *transaction* using the RvSipTransactionGetReceivedMsg() function.
3. Set the *transaction* key using the RvSipTransactionSetKeyFromMsg() function. The proxy can then use the initialized client *transaction* to proxy the request.

RvSipTransactionRequestMsg()

Use this function to send a prepared Request message to the remote party. Proxy implementations will use this function in order to proxy a received request. The request is sent according to the Request-URI found in the *message*. The application is responsible for setting the correct Request-URI in the *message* and for applying the Record-Route rules when necessary.

Note This function cannot be used for sending a CANCEL request.

RvSipTransactionRespondMsg()

Use this function to send a prepared Response message to the remote party. Proxy implementations will use this function in order to proxy a received response. The response is sent according to the top most Via header found in the *message*.

RvSipTransactionIgnoreOutboundProxy()

Instructs the *transaction* to ignore its outbound proxy. Use this function when you are about to send a *message* whose Request-URI was calculated using a Route header. In such a case, the *transaction* should ignore its outbound proxy and use the *message* Request-URI.

RvSipTransactionSendToFirstRoute()

Use this function to instruct the *transaction* to send the message to the first route header in the Route list, and not to the Request-URI. The message should be sent to the first route header and not to the Request-URI when the message is sent to a loose route proxy.

EVENTS

The Transaction API supplies several events, in the form of callback functions, to which your application may listen and react. In order to listen to an event, your application should pass the event handler pointer to the *TransactionMgr*. When an event occurs, the *transaction* calls the event handler function using the pointer.

The following main events are supplied with the Transaction API:

RvSipTransactionCreatedEv()

This event notifies the application that a new server *transaction* has been created and could not be related to any open *call-leg*. The newly created *transaction* always assumes the IDLE state. You should decide whether you wish to handle this transaction. If so, your application can exchange handles with the SIP Stack using this callback. You will then be informed of *transaction* states using the RvSipTransactionStateChangedEv() callback function. If you choose not to handle the transaction, the SIP Stack will handle the *transaction* using its default behavior. In most cases, the SIP Stack will reply with 501 to the incoming request. Requests with To and From tags will be responded to with a 481 response.

RvSipTransactionStateChangedEv()

This event is probably the most useful of the events that the SIP *transaction* reports. Through this function, you receive notifications of SIP *transaction* state changes and the associated state change reason and your application can act upon the state. For example, upon receipt of a SERVER_GENERAL_REQUEST_RECEIVED state notification, your application can respond with a desired response code.

RvSipTransactionMsgToSendEv()

The *transaction* calls this event whenever a *transaction*-related message is ready to be sent. You can use this callback for changing or examining a message before it is sent to the remote party. The *transaction* will not notify you about retransmissions of messages.

RvSipTransactionMsgReceivedEv()

The *transaction* calls this event whenever a *transaction*-related message has been received and is about to be processed. You can use this callback to examine incoming messages. The *transaction* will not notify you if the message is a retransmission.

RvSipTransactionInternalClientCreatedEv()

Notifies the application that a new client *transaction* was created by the SIP Stack. The newly created *transaction* always assumes the IDLE state. This callback is called only for client *transactions* that are created automatically by the SIP Stack (not by calling the function, RvSipTranscMgrCreateTransaction()). Such *transactions* are the CANCEL and PRACK *transactions*.

RvSipTransactionCancelledEv()

Notifies that a CANCEL request is received on an INVITE or a General transaction.

RvSipTransactionOpenCallLegEv()

When a request that is suitable for opening a dialog (INVITE/REFER/SUBSCRIBE with no To tag) is received, the Transaction layer asks the application whether to open a *call-leg* for this transaction. For a proxy application, the callback is called for INVITE/REFER/SUBSCRIBE methods. It can be used by proxies that wish to handle specific requests in a *call-leg* context.

For UA applications, the callback is called only for initial REFER/SUBSCRIBE methods. Applications that do not want the SIP Stack implementation for REFER and SUBSCRIBE that opens a new dialog should implement this callback.

This callback will be called for the INVITE method as well only if the *bDynamicInviteHandling* configuration parameter is set to RV_TRUE. In this case, the application will be able to handle incoming INVITE requests above the Transaction layer.

RvSipTransactionFinalDestResolvedEv()

This event indicates that the *transaction* is about to send a message after the destination address was resolved. This event supplies the final *message*. Changes in the message at this stage will not effect the destination address. When this event is called, the application can query the *transaction* about the destination address using the RvSipTransactionGetCurrentDestAddress() API function. If the application wishes, it can update the "sent-by" part of the top-most Via header. The application must not update the branch parameter. To change the destination address resolved from the *message*, the application must use the Transmitter API. The application should first get the *transmitter* from the *transaction* using the RvSipTransactionGetTransmitter() API function. It can then manipulate the DNS list and current destination address of the *transmitter* before the message is sent. For more information see the Working with Transmitters chapter.

TRANSACTION STATE MACHINE

The Transaction state machine represents the state of the *transaction* between the client and the server. The state machine is divided into the following parts:

- Client General transaction
- Server General transaction
- Client INVITE transaction
- Server INVITE transaction
- Client CANCEL transaction
- Server CANCEL transaction

The RvSipTransactionStateChangedEv() callback reports *transaction* state changes and state change reasons. The state change reason indicates why the *transaction* reached the new state.

A *transaction* will assume either of the following two states, which are common to all state machines:

RVSIP_TRANSC_STATE_IDLE

The IDLE state is the initial state of the Transaction state machine. Upon *transaction* creation, the *transaction* assumes the IDLE state. It remains in this state until RvSipTransactionRequest() is called. If the request method is INVITE, the *transaction* will assume the CLIENT_INVITE_CALLING state. if the request is a general request, the *transaction* will assume the CLIENT_GENERAL_REQUEST_SENT state. A CANCEL *transaction* will assume the CLIENT_CANCEL_REQUEST_SENT state after the CANCEL request is sent.

RVSIP_TRANSC_STATE_TERMINATED

This is the final state of the transaction. When a *transaction* is terminated, the *transaction* assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *transaction*.

CLIENT GENERAL TRANSACTION

A *transaction* may assume any of the following states in the Client General State machine:

RVSIP_TRANSC_STATE_CLIENT_GEN_REQUEST_SENT

After calling RvSipTransactionRequest() with a method other than INVITE, which generates and sends a Request message, the *transaction* enters the CLIENT_GENERAL_ REQUEST_SENT state. The client *transaction* remains in this state until it receives a provisional or final response from the server. Receipt of a provisional response will cause the *transaction* to assume the CLIENT_GENERAL_PROCEEDING state. Receipt of a final response will cause the *transaction* to assume the CLIENT_GENERAL_RESPONSE_RECEIVED state. While in the REQUEST_ SENT state, the *transaction* retransmits the request message according to the rules defined in RFC 3261, and the values configured for T1, T2 and generalRequestTimeoutTimer (timer F). The retransmissions take place only if the transport is an unreliable transport. If no response is received when generalRequestTimeoutTimer is expired, the *transaction* is terminated automatically and assumes the TERMINATED state (in any transport).

RVSIP_TRANSC_STATE_CLIENT_GEN_PROCEEDING

Upon receipt of the first provisional response by a client *transaction* (not a retransmission), the *transaction* assumes the CLIENT_GENERAL_PROCEEDING state. The *transaction* will continue to retransmit the request message until the generalRequestTimeoutTimer is expired

using a consistent interval of T2 seconds as defined in RFC 3261 (only if the transport is unreliable). Receipt of a final response will move the *transaction* to the CLIENT_GENERAL_FINAL_RESPONSE_RCVD state. If no final response is received when the generalRequestTimeoutTimer is expired, the *transaction* is terminated and assumes the TERMINATED state.

RVSIP_TRANSC_STATE_CLIENT_GEN_FINAL_RESPONSE_RCVD

Upon receipt of a final response, the *transaction* assumes the CLIENT_GENERAL_FINAL_RESPONSE_RCVD state. When entering this state, a *transaction* timer is set to T4 if the transport is an unreliable transport. When T4 expires, the *transaction* is terminated and assumes the TERMINATED state. If the transport is a reliable transport, no timer is set and the *transaction* is terminated and assumes the TERMINATED state immediately when it reaches this state. (Timer K as defined in RFC 3261).

RVSIP_TRANSC_STATE_CLIENT_GEN_CANCELLING

When calling the RvSipTransactionCancel() function on a client general transaction, the *transaction* assumes the CLIENT_GENERAL_CANCELLING state. This state indicates that the canceling process has begun. In practice, a new Client CANCEL *transaction* is created and a CANCEL request is sent.

When entering this state, a *transaction* timer is set to cancelGeneralNoResponseTimer. When this timer expires, the *transaction* is terminated. Upon receipt of a final response, the *transaction* assumes the CLIENT_GENERAL_FINAL_RESPONSE_RCVD state.

CLIENT GENERAL TRANSACTION STATE MACHINE

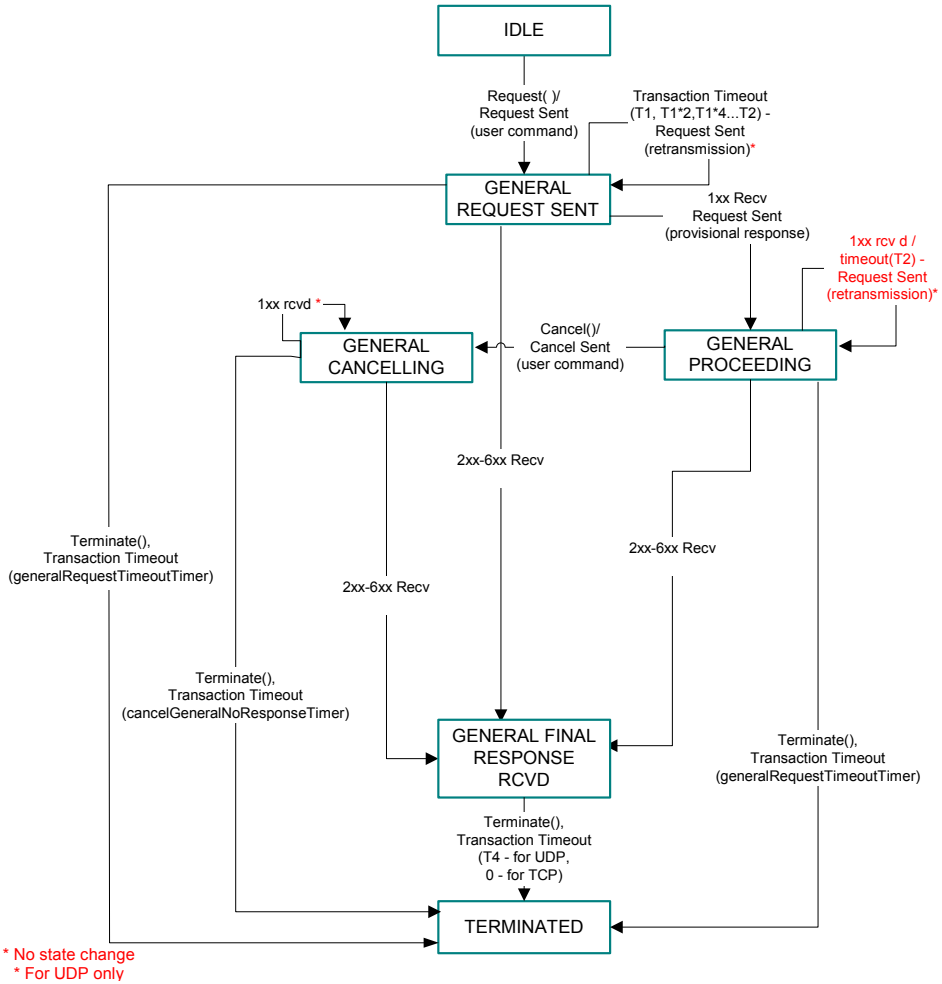


Figure 7-1 *Client General Transaction State Machine*

SERVER GENERAL TRANSACTION

A *transaction* may assume any of the following states in the Server General State machine:

RVSIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD

Upon receipt of a request (that is not a retransmission) by a server transaction, the *transaction* assumes the SERVER_GENERAL_REQUEST_RECEIVED state. In this state, it is up to you to respond to the request using the *transaction* API. You may begin with sending provisional responses. You must end with sending a final response, or terminating the transaction. The *transaction* does not set any timer in this state.

RVSIP_TRANSC_STATE_SERVER_GEN_FINAL_RESPONSE_SENT

When calling RvSipTransactionRespond(), the *transaction* generates and sends a response message. The *transaction* will then assume the SERVER_GENERAL_FINAL_RESPONSE_SENT state. When entering this state, a *transaction* timer is set to generalLingerTimer if the transport is an unreliable transport. When this timer expires, the *transaction* is terminated and assumes the TERMINATED state. If the transport is a reliable transport, the *transaction* is terminated and assumes the TERMINATED state immediately when it reaches this state. (Timer J as defined in RFC 3261).

SERVER GENERAL TRANSACTION STATE MACHINE

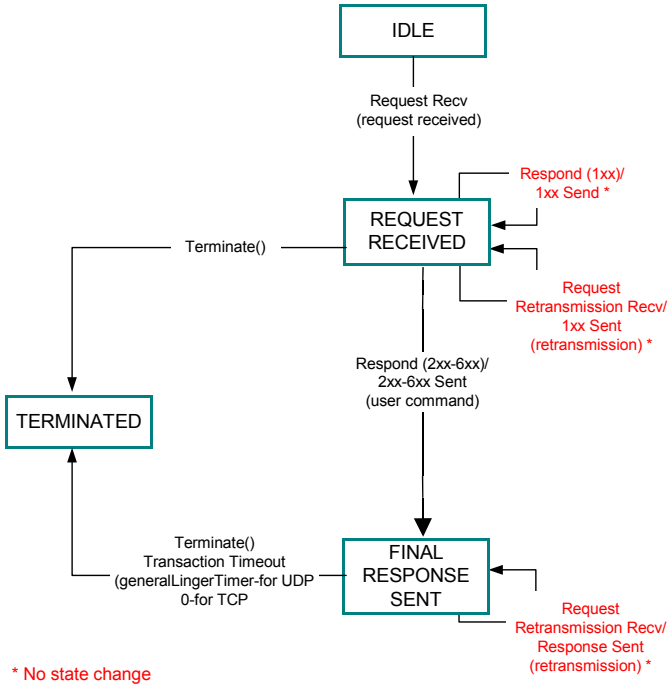


Figure 7-2 *Server General Transaction State Machine*

CLIENT INVITE TRANSACTION

A *transaction* may assume any of the following states in the Client Invite State machine:

RVSIP_TRANSC_STATE_CLIENT_INVITE_CALLING

After calling RvSipTransactionRequest() with the INVITE method which generates and sends an INVITE request message, the *transaction* enters the CLIENT_INVITE_CALLING state. The client *transaction* remains in this state until it receives a provisional or final response from the server. Receipt of a provisional response will cause the *transaction* to assume the CLIENT_INVITE_PROCEEDING state. Receipt of a final response will cause the *transaction* to assume the CLIENT_INVITE_FINAL_RESPONSE_RCVD state.

While in this state, the *transaction* retransmits the request message according to the rules defined in RFC 3261, and the value configured for T1. The retransmissions take place only if the transport is an unreliable transport. If no response is received when 64*T1 timer (Timer B according to RFC 3261) expires, the *transaction* is terminated automatically and assumes the TERMINATED state (in any transport).

RVSIP_TRANSC_STATE_CLIENT_INVITE_PROCEEDING

Upon receipt of the first provisional response by a client INVITE transaction, the *transaction* assumes the CLIENT_INVITE_PROCEEDING state. When entering this state, a *transaction* timer is set to provisionalTimer. When this timer expires, the *transaction* is terminated if the enableInviteProceedingTimeoutState configuration flag is set to RV_FALSE. If the enableInviteProceedingTimeoutState flag is set to RV_TRUE, the *transaction* will move to the CLIENT_INVITE_PROCEEDING_TIMEOUT state. Receipt of a final response in the Proceeding state will move the *transaction* to CLIENT_INVITE_FINAL_RESPONSE_RCVD state.

RVSIP_TRANSC_STATE_CLIENT_INVITE_PROCEEDING_TIMEOUT

This state is reached only if the enableInviteProceedingTimeoutState configuration flag is set to RV_TRUE. Upon reaching timeout on the CLIENT_INVITE_PROCEEDING state, the *transaction* assumes this state. In this state, you have to decide whether to terminate the *transaction* or to cancel it. If you decide to cancel the transaction, a CANCEL message is sent to the destination. In this state, you have to decide on one of the two options, otherwise the *transaction* will wait indefinitely.

RVSIP_TRANSC_STATE_CLIENT_INVITE_FINAL_RESPONSE_RCVD

Upon receipt of an INVITE final response, the *transaction* assumes the CLIENT_ INVITE_FINAL_RESPONSE_RCVD state. If the response is a 2xx response, the *transaction* will then assume the TERMINATED state. If the response is a non-2xx response, the application should initiate the ACK request by calling the RvSipTransactionAck() function. After sending the ACK, the *transaction* will move to the CLIENT_INVITE_ACK_SENT state.

RVSIP_TRANSC_STATE_CLIENT_INVITE_ACK_SENT

After calling RvSipTransactionAck() which generates and sends an ACK request message, the *transaction* enters the CLIENT_INVITE_ACK_SENT state. The *transaction* retransmits the ACK request according to the reliability mechanism defined in the protocol.

When entering this state, a *transaction* timer is set to inviteLingerTimer. When this timer expires, the *transaction* is terminated and assumes the TERMINATED state. (Timer D as defined in RFC 3261). In this way, the *transaction* will not wait indefinitely for retransmissions from the other party.

RVSIP_TRANSC_STATE_CLIENT_INVITE_CANCELLING

When calling the RvSipTransactionCancel() function on a client INVITE transaction, the *transaction* assumes the CLIENT_INVITE_CANCELLED state. This state indicates that the canceling process has begun. In practice, a new Client CANCEL *transaction* is created and CANCEL request is sent.

Upon receipt of an INVITE final response, the *transaction* assumes the CLIENT_ INVITE_FINAL_RESPONSE_RCVD state. When entering the CLIENT_INVITE_CANCELLING state, a *transaction* timer is set to CancelInviteNoResponseTimer. When this timer expires, the *transaction* is terminated and does not wait indefinitely while the other party sends retransmissions.

RVSIP_TRANSC_STATE_CLIENT_INVITE_PROXY_2XX_RESPONSE_RCVD

Upon receipt of an INVITE 2xx final response by a Proxy Client INVITE transaction, the *transaction* assumes the CLIENT_INVITE_PROXY_2XX_RESPONSE_RCVD state. When entering this state, the *transaction* timer is set to proxy2xxRcvdTimer. When this timer expires, the *transaction* is terminated. This state is used only for a Proxy implementation.

CLIENT INVITE TRANSACTION STATE MACHINE

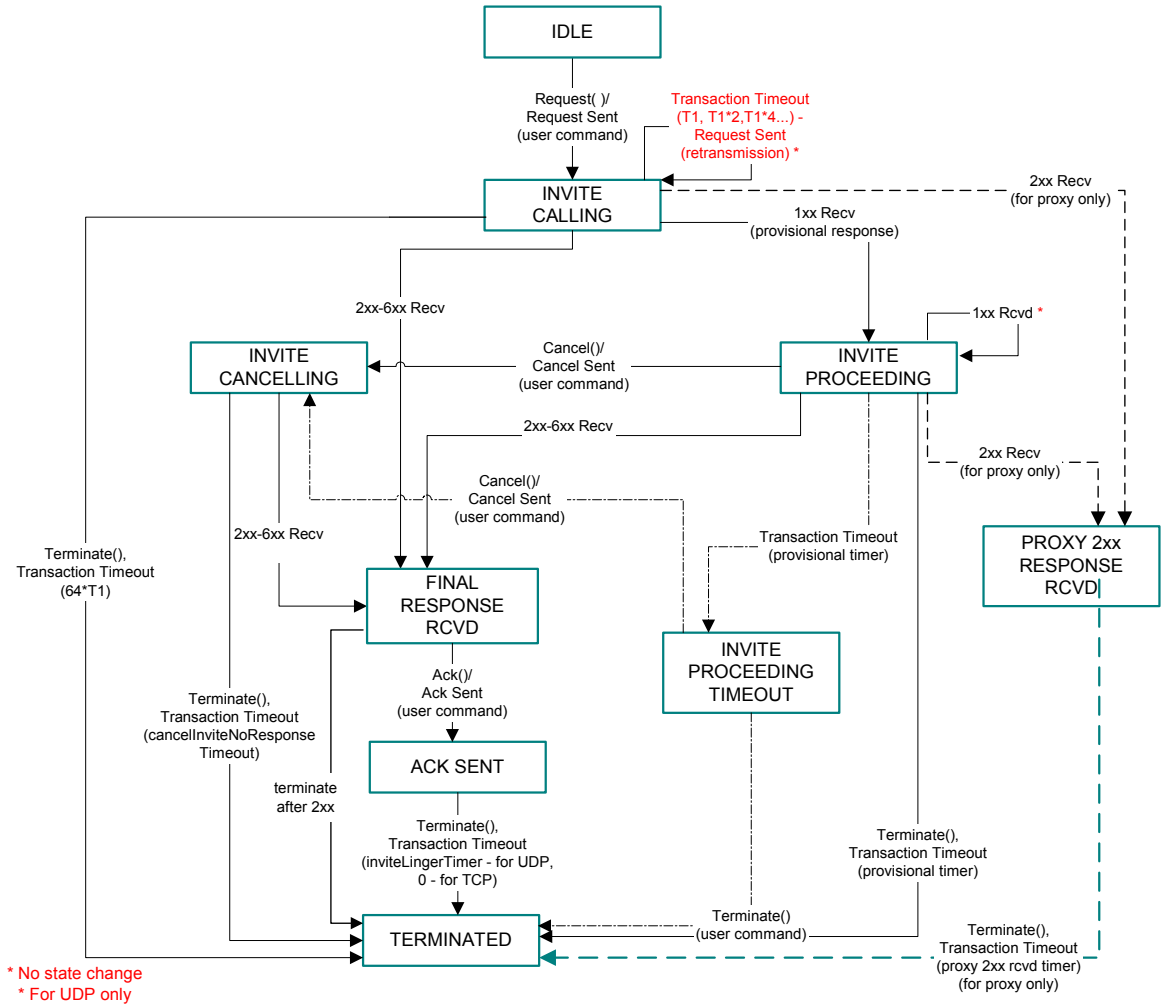


Figure 7-3 *Client INVITE Transaction State Machine*

SERVER INVITE TRANSACTION

A *transaction* may assume any of the following states in the Server INVITE State machine:

RVSIP_TRANSC_STATE_SERVER_INVITE_REQUEST_RCVD

Upon receipt of an INVITE request (that is not a retransmission) by a server INVITE transaction, the *transaction* assumes the SERVER_INVITE_REQUEST_RECEIVED state. In this state, it is up to you to respond to the request using the Transaction API. You may begin with sending provisional responses. You must end with sending a final response, or terminating the transaction.

RVSIP_TRANSC_STATE_SERVER_INVITE_FINAL_RESPONSE_SENT

When calling RvSipTransactionRespond() on an INVITE transaction, the *transaction* generates and sends a response message. The *transaction* will then assume the SERVER_ INVITE_FINAL_RESPONSE_SENT state. For a non-2xx response, the *transaction* retransmits the final response according to RFC 3261, and the value configured for T1 and T2. The retransmissions take place only if the transport is an unreliable transport.

If no ACK response is received when 64*T1 timer (Timer H according to RFC 3261) expires, the *transaction* is terminated automatically and assumes the TERMINATED state (unless the state has changed). If an ACK message is received, the *transaction* moves to the SERVER_INVITE_ACK_RCVD state. For a 2xx response, the *transaction* sets the inviteLingerTimer after sending the 2xx response. The *transaction* terminates when this timer expires.

RVSIP_TRANSC_STATE_SERVER_INVITE_ACK_RCVD

When entering this state (the *transaction* received an ACK message), a *transaction* timer is set to T4 if the transport is an unreliable transport. When T4 expires, the *transaction* is terminated and assumes the TERMINATED state. If the transport in a reliable transport, no timer is set and the *transaction* is terminated and assumes the TERMINATED state immediately when it reaches this state. (Timer I as defined in RFC 3261).

RVSIP_TRANSC_STATE_SERVER_INVITE_REL_PROV_RESPONSE_SENT

After calling the RvSipTransactionRespondReliable() function in the SERVER_INVITE_REQUEST_RCVD state, a reliable provisional response will be sent and the *transaction* will assume the SERVER_INVITE_REL_PROV_RESPONSE_SENT state. The *transaction* retransmits the Reliable Provisional Response according to RFC 3261 and the configuration of T1. When 64*T1 timer (Timer H according to RFC 3261)

expires, the *transaction* automatically sends a 500 response. Receipt of the PRACK and responding to it moves the *transaction* to the SERVER_INVITE_PRACK_COMPLETED state.

RVSIP_TRANSC_STATE_SERVER_INVITE_PRACK_COMPLETED

While in the SERVER_INVITE_REL_PROV_RESPONSE_SENT state, the INVITE server *transaction* waits for a PRACK process to be completed. The PRACK process is handled by a separate general server transaction. When the process is completed (PRACK request is received and responded to), the INVITE server *transaction* assumes the SERVER_INVITE_PRACK-COMPLETED state. You can continue by sending one or more provisional responses and must finish with a final response.

RVSIP_TRANSC_STATE_SERVER_INVITE_PROXY_2XX_RESPONSE_SENT

When proxying an INVITE 2xx response with the RvSipTransactionRespondMsg() function, the *transaction* sends a response message. The *transaction* will then assume the SERVER_PROXY_INVITE_FINAL_RESPONSE_SENT state. When entering this state, the *transaction* timer is set to proxy2xxSentTimer.When this timer expires, the *transaction* is terminated. This state is used only for a Proxy implementation.

Note The ACK on a 2xx response to INVITE is not part of the Invite transaction.

SERVER INVITE TRANSACTION STATE MACHINE

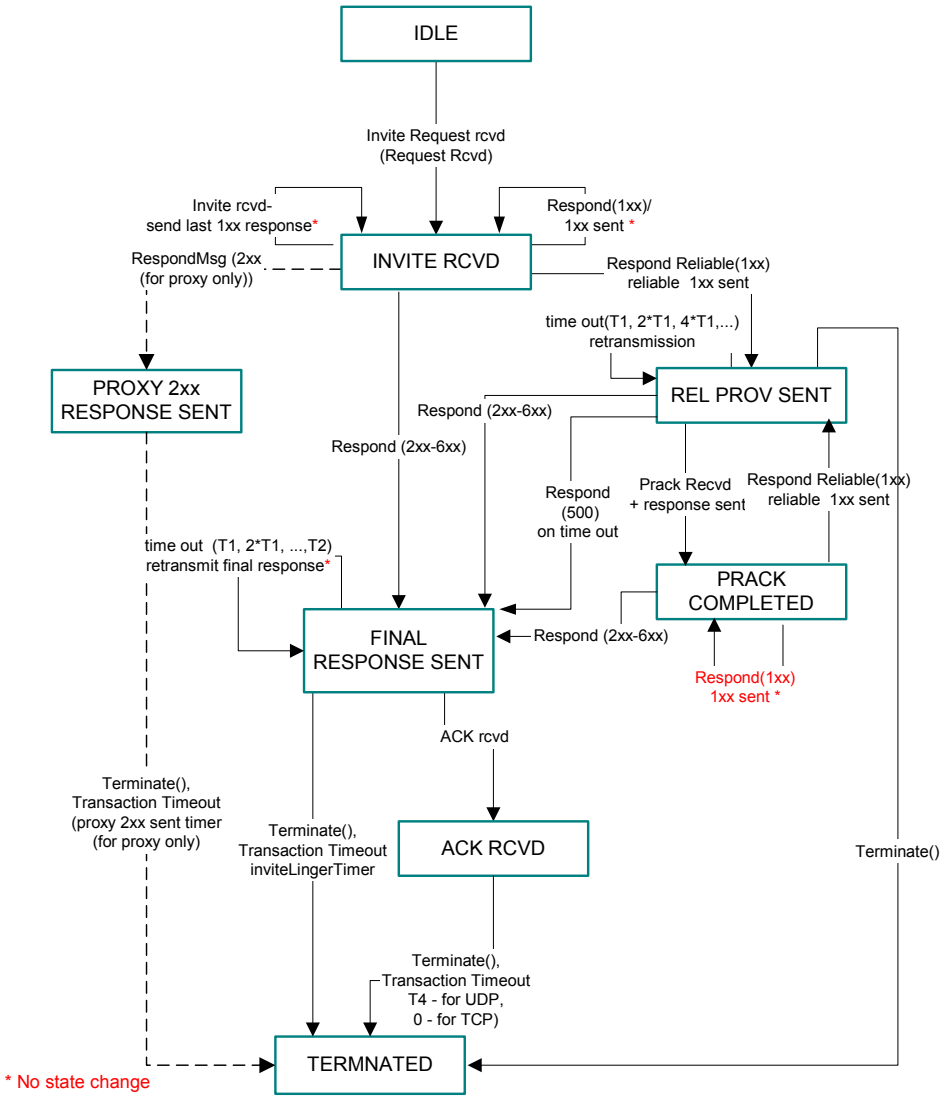


Figure 7-4 *Server INVITE Transaction State Machine*

CLIENT CANCEL TRANSACTION

A *transaction* may assume any of the following states in the Client CANCEL state machine:

RVSIP_TRANSC_STATE_CLIENT_CANCEL_SENT

When calling the RvSipTransactionCancel() function, a new Client CANCEL *transaction* is created and a CANCEL request is sent. The client CANCEL *transaction* assumes the CLIENT_CANCEL_SENT state. The *transaction* retransmits the CANCEL message according to the rules defined in RFC 3261, and the values configured for T1, T2 and generalRequestTimeoutTimer (timer F). The retransmissions take place only if the transport is an unreliable transport. If no response is received when generalRequestTimeoutTimer is expired, the *transaction* is terminated and assumes the TERMINATED state automatically (in any transport). Receipt of a 1xx response message moves the *transaction* to the CLIENT_CANCEL_PROCEEDING state. Receipt of a final response (2xx-6xx) moves the *transaction* to the CLIENT_CANCEL_FINAL_RESPONSE_RCVD state.

RVSIP_TRANSC_STATE_CLIENT_CANCEL_PROCEEDING

Upon receipt of a provisional response by a client CANCEL transaction, the *transaction* assumes the CLIENT_CANCEL_PROCEEDING state. The *transaction* will continue to retransmit the request message until generalRequestTimeoutTimer expires, using a consistent interval of T2 seconds as defined in RFC 3261 (Only if the transport is unreliable). Receipt of a final response will move the *transaction* to the CLIENT_CANCEL_FINAL_RESPONSE_RECEIVED state.

RVSIP_TRANSC_STATE_CLIENT_CANCEL_FINAL_RESPONSE_RCVD

When entering this state, a *transaction* timer is set to T4 if the transport is an unreliable transport. When this timer expires, the *transaction* will terminate and assume the TERMINATED state. If the transport is a reliable transport, the *transaction* will terminate and assume the TERMINATED state immediately when it reaches this state. (Timer J as defined in RFC 3261).

CLIENT CANCEL TRANSACTION STATE MACHINE

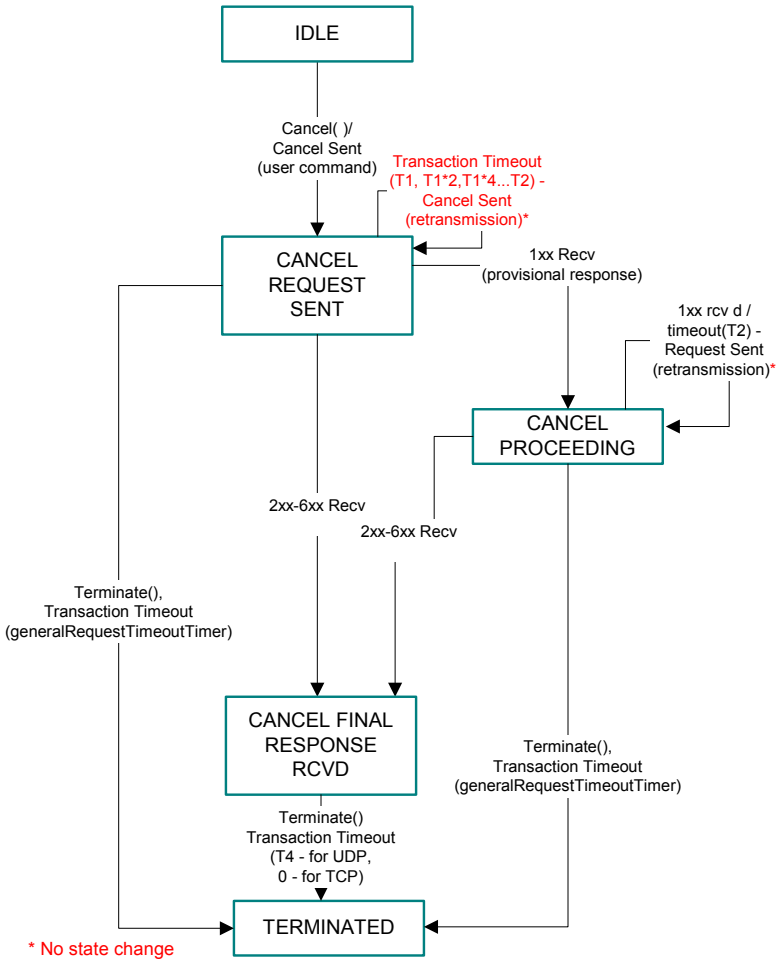


Figure 7-5 *Client CANCEL Transaction State Machine*

SERVER CANCEL TRANSACTION

A *transaction* may assume any of the following states in the Server CANCEL state machine:

RVSIP_TRANSC_STATE_SERVER_CANCEL_REQUEST_RCVD

Upon receipt of a CANCEL request that is not a retransmission by a server transaction, the *transaction* assumes the SERVER_CANCEL_REQUEST_RECEIVED state. In this state, it is up to you to respond to the request with the Transaction API functions. You may begin with sending provisional responses. You must end with sending a final response, or terminating the transaction. This state is used only if the SIP Stack is configured as a proxy. Otherwise, the CANCEL is responded to automatically.

RVSIP_TRANSC_STATE_SERVER_CANCEL_FINAL_RESPONSE_SENT

If your application is not a Proxy implementation, when a CANCEL request is received, it is automatically handled by the Transaction layer. The *transaction* will initiate the response to the CANCEL by itself and the *transaction* will assume the SERVER_CANCEL_FINAL_RESPONSE_SENT state. If your application is a Proxy implementation it is your responsibility to send a final response for the CANCEL request. When you send a final response, the *transaction* will move to this state.

In any application (Proxy or not) when entering this state, a *transaction* timer is set to generalLingerTimer if the transport is an unreliable transport. When this timer expires, the *transaction* is terminated and assumes the TERMINATED state. If the transport is a reliable transport the *transaction* is terminated and assumes the TERMINATED state immediately when it reaches this state. (Timer J as defined in RFC 3261).

SERVER CANCEL TRANSACTION STATE MACHINE

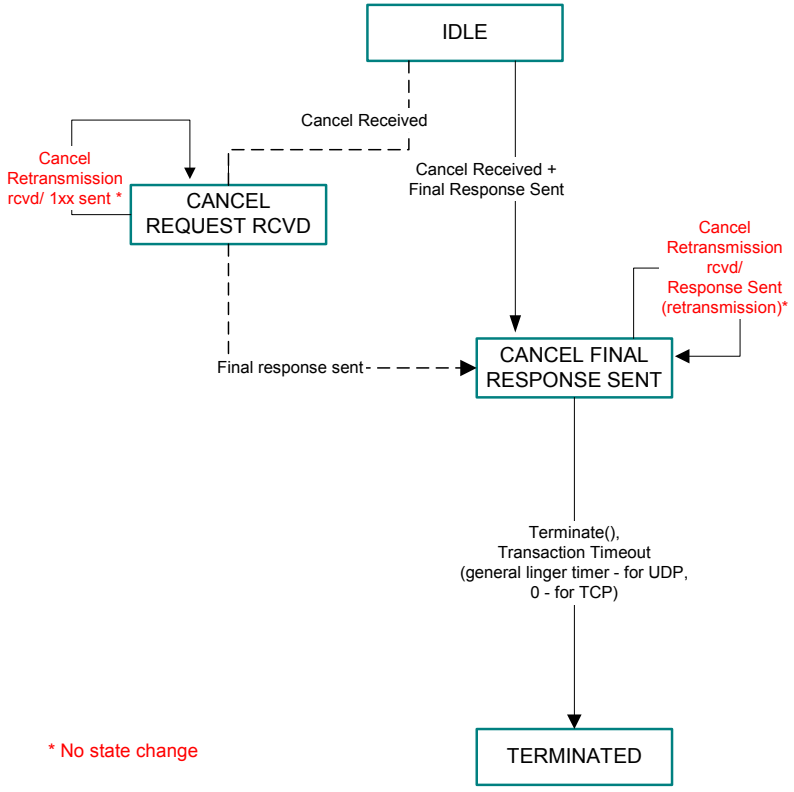


Figure 7-6 *Server CANCEL Transaction State Machine*

TRANSACTION ADVANCED STATES

Transaction states that are assumed only for specific configurations are referred to as “advanced states”. The *transaction* advanced state is Message Send Failure.

MESSAGE SEND FAILURE

The *transaction* assumes this state only when the SIP Stack is compiled with the RV_DNS_ENHANCED_FEATURES_SUPPORT compilation flag. Both Client Invite *transactions* and Client General *transactions* can assume this state.

RVSIP_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE

The *transaction* assumes this state when the sending of a request fails due to timeout (timer B or F expired and no response was received), network error or 503 final response. In this state it is your responsibility to decide whether to terminate the *transaction* or call the continue-DNS function that opens a new *transaction* and sends the message to the next address in the DNS list. If you decide to continue-DNS then it is the SIP Stack responsibility to terminate the failed transaction. This state can be reached from the following states:

- RVSIP_TRANSC_STATE_CLIENT_GEN_REQUEST_SENT
- RVSIP_TRANSC_STATE_CLIENT_GEN_PROCEEDING (when a 503 response is received)
- RVSIP_TRANSC_STATE_CLIENT_INVITE_CALLING
- RVSIP_TRANSC_STATE_CLIENT_INVITE_PROCEEDING (when a 503 response is received)
- RVSIP_TRANSC_STATE_CLIENT_INVITE_ACK_SENT.

For more information, see the Working with DNS chapter.

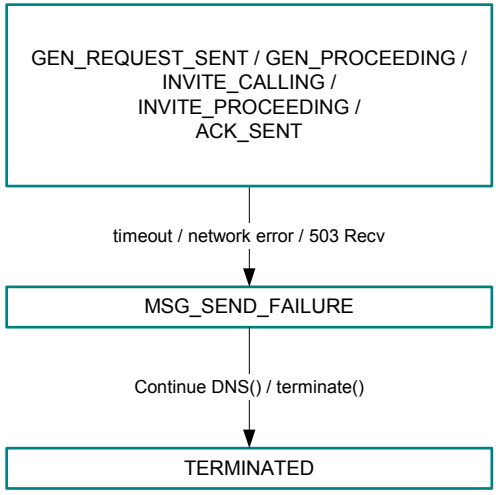


Figure 7-7 *Message Send Failure State Machine*

TRANSACTION MANAGER API

The *TransactionMgr* controls the SIP Stack collection of *transactions*. You use the Transaction Manager API to register application callbacks with the SIP Stack and to create new *transactions*.

TRANSACTION MANAGER CONTROL

The following API functions are provided for *TransactionMgr* control:

RvSipTranscMgrCreateTransaction()

You use this function to create a new *transaction*.

RvSipTransactionMgrSetEvHandlers()

You use this function to set event handler (callback function) pointers that are related to a *transaction* in the SIP Stack. The Transaction API declares prototypes for all *transaction* callback functions. For example:

```
/*=========================================================================*/
typedef void (RVCALLCONV * RvSipTransactionStateChangedEv)(
                IN RvSipTranscHandle                  hTransc,
                IN RvSipTranscOwnerHandle             hTranscOwner,
                IN RvSipTransactionState              eState,
                IN RvSipTransactionStateChangeReason  eReason);
/*=========================================================================*/
```

You can implement any callback you find necessary according to the callback function prototypes. All *transaction* callback functions are included in a structure called RvSipTransactionEvHandlers. This event handler structure is where you should set your callback function pointers and is given as a parameter to RvSipTransactionMgrSetEvHandlers(). The *transaction* notifies you when the event occurs using the callback functions you implemented.

Although you can set the event handlers at any time, it is customary to set them immediately after RvSipStackConstruct() so that the application immediately receives all necessary notifications.

RvSipTransactionMgrSetMgrEvHandlers()

The *TransactionMgr* informs the application about events that are not related to a specific transaction. These events are gathered in the RvSipTransactionMgrEvHandlers structure. You use the RvSipTransactionMgrSetMgrEvHandlers() function to set the pointers of your event handlers in the SIP Stack.

RvSipTranscMgrCreateServerTransactionFromMsg()

Creates a new Server *transaction* from a request message. This function will be used by stateless proxy applications. A stateless proxy does not open a *transaction* for incoming requests. However, according to RFC 3261, if a stateless proxy wishes to reject a request, it needs to handle this request in a stateful manner. For this, the stateless proxy needs to instruct the SIP Stack to open a server *transaction* from the request message. If the message method is INVITE, the new *transaction* will assume the SERVER_INVITE_REQUEST_RCVD state. If the message method is CANCEL, the new *transaction* will assume the SERVER_CANCEL_REQUEST_RCVD state. Otherwise the new *transaction* will assume the SERVER_GEN_REQUEST_RCVD state.

Transaction Manager Events

The *TransactionMgr* informs the application of events that are not related to a specific transaction. The following *TransactionMgr* events are available:

RvSipTranscMgrOutOfContextMsgRcvdEv()

This event is called when the *TransactionMgr* receives a message that does not match any existing transaction. The callback is called in the following cases:

- For a response message that does not match any client transaction.
- When ACK for a 2xx response is received.
- When CANCEL is received and the cancelled *transaction* is not found.

When the application acts as a user agent (UA) it will be notified only of ACK, 1xx and 2xx responses for INVITE. When the application is a proxy, it will get all the above notifications. The proxy needs decide whether to ignore the message or to proxy it to the destination in a stateless manner using the Transmitter API.

RvSipTranscMgrNewRequestRcvdEv()

This event is called when the *TransactionMgr* receives a new request that is not a retransmission and not an ACK request. The application should instruct the Stack whether or not it should create a new *transaction* for the request. If the application decides not to create a new *transaction* for the request, RvSipTranscMgrOutOfContextMsgRcvdEv() will be called. If you do not implement this callback, a new *transaction* will be created by default. This callback is usually used by stateless proxies.

USING TRANSACTIONS

The following sections present various ways of using *transactions* accompanied by code samples. These sections include:

- Registering Application Callbacks
- Exchanging Handles with the Application
- Sending a Request
- Using the Outbound Message Mechanism
- Transaction Merging Support

REGISTERING APPLICATION CALLBACKS

To register an application callback, you must first define the callback according to the prototype. The following code demonstrates an implementation of the RvSipTransactionCreatedEv() and RvSipTransactionStateChangedEv() callback functions.

Sample Code

The following sample code demonstrates an implementation of the RvSipTransactionCreatedEv() callback function. In this sample, the application indicates that it wishes to handle only the OPTIONS request. This means that other requests will be handled automatically by the SIP Stack.

```
/*=========================================================================*/
static void RVCALLCONV AppTransactionCreatedEvHandler(
                        IN RvSipTranscHandle          hTransc,
                        IN void                       *context,
                        OUT RvSipTranscOwnerHandle    *phAppTransc,
                        OUT RvBool                    *b_handleTransc)
{
        RvChar strMethod[20];
        *phAppTransc = NULL;
        *b_handleTransc = RV_FALSE;
        RvSipTransactionGetMethodStr(hTransc,20,strMethod);
        if(strcmp(strMethod,"OPTIONS") == 0)
        {
                *b_handleTransc = RV_TRUE;
        }
}
/*=========================================================================*/
```

Sample Code

The following sample code demonstrates an implementation of the RvSipTransactionStateChangedEv() callback function. Since, in the above sample code, the application requested to handle only the OPTIONS requests, it will be notified of the state of the OPTIONS *transactions* only. (Therefore there is no need to query the *transaction* method again.) In this sample, the application responds with 200 to the request.

```
/*=========================================================================*/
void RVCALLCONV AppTransactionStateChangedEvHandler(
                        IN RvSipTranscHandle hTransc,
                        IN RvSipTranscOwnerHandle hAppTransc,
                        IN RvSipTransactionState eState,
                        IN RvSipTransactionStateChangeReason eReason)
{
        RvStatus rv;
        switch(eState)
        {
        case RVSIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD:
                rv = RvSipTransactionRespond(hTransc,200,NULL);
                if(rv!= RV_OK)
                {
                        printf("Failed to respond to the request");
                }
                break;
        default:
                break;
        }
}
/*=========================================================================*/
```

The following steps describe how to register your application callbacks.

To register application callbacks

1. Declare a RvSipTransactionEvHandlers structure.
2. Initialize all the structure members to zero using memset().
3. Set the application defined callback in RvSipTransactionEvHandlers.
4. Call RvSipTransactionMgrSetEvHandlers().

Sample Code

The following code demonstrates an application implementation of callback registration.

```
/*=========================================================================================*/
void SetTransactionEvHandlers(RvSipTranscMgrHandle hMgr)
{
        /*step 1*/
        RvSipTransactionEvHandlers appEvHandlers;
        /*step 2*/
        memset(&appEvHandlers,0,sizeof (RvSipTransactionEvHandlers));
        /*step 3*/
        appEvHandlers.pfnEvTransactionCreated = AppTransactionCreatedEvHandler;
        appEvHandlers.pfnEvStateChanged = AppTransactionStateChangedEvHandler;
        /*step 4*/
        RvSipTransactionMgrSetEvHandlers(hMgr,NULL,&appEvHandlers,
                                         sizeof(RvSipTransactionEvHandlers));
}
/*=========================================================================================*/
```

Exchanging Handles with the Application

If you wish to handle a transaction, you become the owner of the *transaction* and you can create your own handle to the *transaction*. This will prove useful when you have your own application *transactions* database.

You can provide the SIP Stack with your *transaction* handle, which it must supply when calling your application callbacks.

You can use the RvSipTranscMgrCreateTransaction() API function to exchange handles for a client *transaction* and the RvSipTransactionCreatedEv() callback to exchange handles for a server transaction.

Sending a Request

The following steps describe how to send a request.

To send a request

1. Declare a handle for the new transaction.
2. Call the RvSipTranscMgrCreateTransaction() function. This enables you to exchange handles with the SIP Stack.
3. Call the RvSipTransactionMake() function. This function sends the request to the remote party.

Sample Code

The following code demonstrates an implementation of a call request.

```
/*=========================================================================*/
RvStatus AppSendOptionsRequest(RvSipTranscMgrHandle hMgr)
{
        RvSipTranscHandle       hTransc; /*Handle to the transaction.*/
        RvStatus               rv;

       RvChar  *strFrom = "From:sip:user1@172.20.0.1:5060";
       RvChar  *strTo = "To:sip:user2@172.20.10.11:5060";
       RvChar  *strRequestUri = "sip:172.20.0.1:5060";
       RvChar  *strMethod = "OPTIONS";
       RvInt32 cseq = 5;
        /*---------------------------
         Creates a new transaction.
        ----------------------------*/
        rv =
         RvSipTranscMgrCreateTransaction(hMgr,NULL,&hTransc);
        if(rv != RV_OK)
        {
            printf("Failed to create a new transaction");
            return rv;
        }
        /*-----------------------------------------------------
         Sends the request by calling the "make" function.
        -----------------------------------------------------*/
        rv = RvSipTransactionMake(hTransc,
                                  strFrom,
                                  strTo,
                                  strRequestUri,
                                  cseq,
                                  strMethod);
        if(rv != RV_OK)
        {
             printf("Transaction Make failed");
             return rv;
        }
         return RV_OK;
}
/*=========================================================================*/
```

USING THE OUTBOUND MESSAGE MECHANISM

The following steps describes how to use the outbound message mechanism to add the headers listed below to the 200 response of an OPTIONS request:

- Allow: INVITE,ACK,CANCEL,BYE,OPTIONS
- Accept: application
- Accept-Encoding: gzip

To add the headers

1. Get the outbound message from the *transaction* using the RvSipTransactionGetOutboundMsg() function.
2. Use the message API functions to add the above headers.
3. Send the 200 response using the RvSipTransactionRespond() function.

Sample Code

The following code demonstrates the steps described above.

Note The different fields of the Allow header should be added as separate Allow headers. However, when the message is encoded, all the methods are gathered in one Allow header.

Note In this sample, the return value of the Stack API functions is not checked for simplicity. You must always check return values.

```
*/============================================================================*/
void RVCALLCONV AppTransactionStateChangedEvHandler(
                        IN RvSipTranscHandle                hTransc,
                        IN RvSipTranscOwnerHandle           hAppTransc,
                        IN RvSipTransactionState            eState,
                        IN RvSipTransactionStateChangeReason eReason)
{

        RvSipMsgHandle hMsg = NULL;
        RvSipOtherHeaderHandle hHeader;
        RvSipAllowHeaderHandle hAllow;

        switch(eState)
        {
        case RVSIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD:
```

```
                RvSipTransactionGetOutboundMsg(hTransc,&hMsg);

                /*Adds the 'Allow: INVITE,ACK,CANCEL,BYE,OPTIONS' header*/
                RvSipAllowHeaderConstructInMsg(hMsg,RV_FALSE,&hAllow);
                RvSipAllowHeaderSetMethodType(hAllow,RVSIP_METHOD_INVITE,NULL);

                RvSipAllowHeaderConstructInMsg(hMsg,RV_FALSE,&hAllow);
                RvSipAllowHeaderSetMethodType(hAllow,RVSIP_METHOD_ACK,NULL);

                RvSipAllowHeaderConstructInMsg(hMsg,RV_FALSE,&hAllow);
                RvSipAllowHeaderSetMethodType(hAllow,RVSIP_METHOD_CANCEL,NULL);

                RvSipAllowHeaderConstructInMsg(hMsg,RV_FALSE,&hAllow);
                RvSipAllowHeaderSetMethodType(hAllow,RVSIP_METHOD_BYE,NULL);

                RvSipAllowHeaderConstructInMsg(hMsg,RV_FALSE,&hAllow);
                RvSipAllowHeaderSetMethodType(hAllow,RVSIP_METHOD_OTHER,"OPTIONS");

                /*Adds the 'Accept: application/' header*/
                RvSipOtherHeaderConstructInMsg(hMsg,RV_FALSE,&hHeader);
                RvSipOtherHeaderSetName(hHeader,"Accept");
                RvSipOtherHeaderSetValue(hHeader,"application/");

                /*Adds the 'Accept-Encoding: gzip' header*/
                RvSipOtherHeaderConstructInMsg(hMsg,RV_FALSE,&hHeader);
                RvSipOtherHeaderSetName(hHeader,"Accept-Encoding");
                RvSipOtherHeaderSetValue(hHeader,"gzip");

                RvSipTransactionRespond(hTransc,200,NULL);
                break;
        default:
                break;
        }
}
/*===========================================================================*/
```

TRANSACTION MERGING SUPPORT

A proxy may fork a request to several different proxies. As a result, the UAS may receive several request messages, with the same request identifier (Call-ID, From tag, To tag and cseq), but with a different via-branch. Since the branch parameter in the Via header is different, a new *transaction* will be created for each request.

According to RFC 3261, in the case of several incoming *transactions* all caused by a single request message, the UAS should handle only one request, and reject all others with the 482 (Loop Detected) response. This procedure is called “Merging”.

Figure 7-8 illustrates the message flow of a merging scenario.

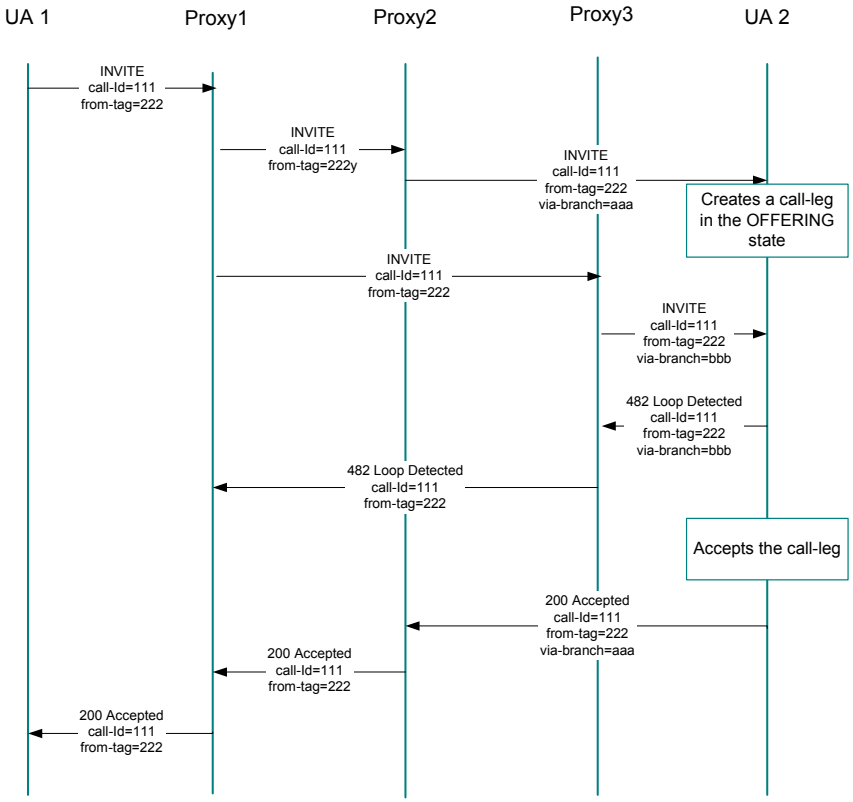


Figure 7-8 *Merging Message Flow*

The following is a description of the message flow of a merging scenario:

1. Proxy1 forked the INVITE request to Proxy2 and Proxy3. Both Proxy2 and Proxy3 forward the request to UA2. Therefore,

UA2 receives two INVITE requests with same request identifiers, but with different Via headers.

2. UA2 handles the first INVITE request as usual—it creates a *call-leg* in the OFFERING state, and waits for the application decision of whether to accept or reject this request.
3. UA2 discovers that the second INVITE request has the same request identifiers, and therefore rejects it with 482.
4. UA2 accepts the *call-leg*, and sends a 200 response on the first request message.
5. Proxy1, receiving both 482 and 200 responses, forwards the 200 response to UA1.

Disabling Merging Support

The SIP Stack supports merging behavior by default. However, applications may choose to disable the merging behavior. In this case, all INVITE requests will be handled in the same way—server *transactions* will be created and passed over to *call-leg* handling, even though they were all caused by the same initial INVITE request.

The following new configuration parameter is supplied for this purpose:

bDisableMerging

Defines whether or not to activate the merging support. If RV_FALSE, the merging behavior is enabled. If RV_TRUE, the merging behavior is disabled. The default value is RV_FALSE.

8

WORKING WITH REGISTER-CLIENTS

INTRODUCTION

The SIP REGISTER request allows a client to inform a Proxy or redirect server of the addresses at which the client can be reached.

The Register-Client API of the SIP Stack enables you to register with a Proxy or a SIP server, refresh registration when needed, and send authentication information to the server, as required.

The Register-Client API relates to the following two entities:

REGISTER-CLIENT

The register-client (*register-client*) enables your application to register with several alternative addresses, each with a unique expiration time, and to refresh old registrations. A register-client is a stateful object which can assume any state from a set defined in the Register-Client API. A register-client state represents the state of the registration process.

REGISTER-CLIENT MANAGER

The Register-Client Manager (*Register-ClientMgr*) manages the collection of all *register-clients* and is mainly used for creating new *register-clients*. In applications that register a single user to a particular registrar, all registrations from the UA should use the same Call-ID value. For this purpose, the *Register-ClientMgr* generates and holds the Call-ID that is shared by all *register-clients* and is unique to a reboot cycle. The *Register-ClientMgr* also manages the CSeq-Step counter, which increases every time the client sends a REGISTER request.

SINGLE-USER VERSUS MULTI-USER APPLICATIONS

By default, the *Register-ClientMgr* owns all the *register-clients*. The *Register-ClientMgr* is responsible for generating a Call-ID once and supplying it to all its *register-clients*. The *Register-ClientMgr* is also responsible for the CSeq-Step counter. Each *register-client* will receive the CSeq-Step current count from the *Register-ClientMgr* immediately before sending a new REGISTER request. The *Register-ClientMgr* will increase the CSeq-Step count each time a new REGISTER request is sent by one of its *register-clients*.

This mode of action is suitable for single user applications that register to a single registrar. When an application registers several users to different registrars, each of the *register-clients* needs to have its own Call-ID and to manage its own CSeq step. For this, the register client object needs to detach from its manager after its creation. The SIP Stack supplies API functions for detaching a *register-client* from the *Register-ClientMgr*. In this chapter, *register-clients* that detached from the *Register-ClientMgr* are referred to as stand-alone *register-clients*. You can either set a Call-ID for a stand-alone *register-client* or the *register-client* will generate one for you. A stand-alone *register-client* will also manage its own CSeq step counter and increase it for every outgoing REGISTER request.

For more information, see the RvSipRegClientDetachFromMgr() function in the *SIP Stack Reference Guide*.

WORKING WITH HANDLES

All *register-clients* and the *Register-ClientMgr* are identified using handles. You must supply these handles when using the Register-Client API.

RvSipRegClientMgrHandle defines the *Register-ClientMgr* handle. You receive this handle by calling RvSipStackGetRegClientMgrHandle().
RvSipRegClientHandle defines a *register-client* handle. You receive this handle from RvSipRegClientMgrCreateRegClient().

REGISTER-CLIENT API

The Register-Client API contains a set of functions and function callbacks that allow you to set or examine register-client parameters, control REGISTER requests and respond to network events.

REGISTER-CLIENT PARAMETERS

You can set or examine register-client parameters via Register-Client API Set and Get functions. The following parameters are available:

To Header and From Header

When creating a new register-client object, you must set the To and From headers of the object. After you send a REGISTER request from this object, you must not change the To and From header values.

Contact Headers List

The register-client object manages a list of Contact headers. The API allows you to add, remove and view Contact headers from the contact header list. When sending a REGISTER request, the register-client object adds all the Contact headers from the list to the outgoing REGISTER message.

Expires Header

The application can set an Expires header to the register-client object. When sending a REGISTER request, the register-client object adds this Expires header to the outgoing REGISTER message.

Registrar

Before registering to a Registrar, you must set the Registrar address in the registrar-client object. When the register-client receives a 3xx class response, the register-client will update the Registrar address with the first contact header that is found in the response. The application can view and change the Registrar address as necessary. You can update the Registrar address prior to sending a REGISTER request.

State

The register-client state parameter indicates the state of the registration process. You can access the state parameter only with a Get function. The state parameter is not modifiable.

CSeq-Step

The CSeq-Step is either managed by the *register-client* (for stand-alone *register-clients*) or by the *Register-ClientMgr*. Stand-alone *register-clients* increase the CSeq-Step for every REGISTER request. Non-stand-alone *register-clients* receive the CSeq-Step from the *Register-ClientMgr* each time they wish to send a REGISTER Request. The *Register-ClientMgr* keeps a global CSeq and increases it each time a new REGISTER request is sent.

You can access the CSeq-Step parameter only with a Get function. The CSeq-Step parameter is not modifiable.

Call-ID

By default, the *register-client* gets its Call-ID from the *Register-ClientMgr*. This Call-ID is global and generated once when the SIP Stack initializes. Stand-alone *register-clients* use a different Call-ID. You can either set the Call-ID to a stand-alone *register-client* or the *register-client* will generate the Call-ID.

Outbound Address and Local Address

These are the addresses the *register-client* uses for sending requests. If you set the *register-client* outbound address, all requests will be sent to this address regardless of the message Request-URI. The local address defines the address from which the Request will be sent (the network card). This is also the address that will be placed in the top Via header of the Request message. If the local address is not set, the *register-client* uses a default local address taken from the SIP Stack configuration.

Outbound Message

The outbound message is a handle to a *message* that the *register-client* will use for the next outgoing REGISTER request. Before calling an API function that causes the request to be sent, the application can get the outbound *message* and add headers and a body. (Note that at this stage, the object is empty.) You cannot use the outbound message to set headers that are part of the *register-client transaction* key such as To, From, Call-ID, CSeq and Via headers.

Received Message

The last response message that was received by the register-client. You can get this response only in the context of the *register-client* state-changed callback function when the new state indicates that the *register-client* received a response.

Register-client Transaction Timers and Retransmission Count

The SIP Stack configuration determines the value of the timers and retransmission count for all the SIP Stack *transactions*. The application can use a set function to change the different timer values of the *register-client transactions*. The application can also control the number of retransmissions that the *transactions* perform.

Persistency Definition and Used Connection

When working with TCP, the application can instruct the *register-client* to try and send all outgoing register requests on one TCP connection. The application can also query the *register-client* about the connection used to send each request. For more information on persistency level and Persistent Connection API functions, see Persistent Connection Handling of the Working with the Transport Layer chapter.

New Header Handles

Some of the *register-client* fields are message parts. For example, the To header field is a Party header object and the Expires field is an Expires header object.

Before setting a parameter of this type in the register-client, you should request a new handle for the given parameter from the register-client. After initializing this parameter with your requested values, you can set the parameter back in the register-client.

REGISTER-CLIENT CONTROL

The following API functions provide *register-client* control:

RvSipRegClientDetachFromMgr

Detaches a *register-client* from the *Register-ClientMgr*. By default, all the *register-clients* created in a single SIP Stack instance represent a single User Agent and therefore use the same Call-ID and an increased CSeq- Step. The Call-ID and CSeq-Step are managed by the *Register-ClientMgr*. When implementing a multi-user application, each *register-client* should have a different Call-ID and should manage its own CSeq-step counting. Calling RvSipRegClientDetachFromMgr() on a *register-client* will cause the *register-client* to generate its own Call-ID and manage its own CSeq-Step counter.

RvSipRegClientMake()

After creating a register-client, you can use this function to set the To and From headers and the contact and Registrar addresses in the register-client, and send the REGISTER request. You can use this function if you have all the needed fields in textual format.

RvSipRegClientRegister()

After creating a *register-client* and setting the To, From, Expires and Contact headers and the Registrar address, you can use RvSipRegClientRegister() for generating and sending the required REGISTER message to the Registrar.

RvSipRegClientAuthenticate()

If you receive 401 or 407 responses which indicate that the REGISTER request was not authenticated by the Registrar, the *register-client* assumes the UNAUTHENTICATED state. You can use RvSipRegClientAuthenticate() in the UNAUTHENTICATED state to re-send the registration request with authentication information.

RVSipRegClientTerminate()

Terminates a *register-client* and frees all its resources. After calling this function, you can no longer reference the *register-client*. Note that the terminate function does not send any messages to the Registrar, rather simply destructs the object.

EVENTS

The Register-Client API supplies several events, in the form of callback functions, to which your application may listen and react. In order to listen to an event, your application should first define a special function called the event handler and then pass the event handler pointer to the *Register-ClientMgr*. When an event occurs, the *register-client* calls the event handler function using the pointer.

The following events are supplied with the Register-Client API:

RvSipRegClientStateChangedEv()

This event is probably the most useful of the events that the SIP *register-client* reports. Through this function, you receive notifications of SIP register-client state changes and the associated state change reason. You can then react to the

new state. For example, upon receipt of an UNAUTHENTICATED state notification, your application can use the RvSipRegClientAuthenticate() to re-send the registration request with authentication information.

RvSipRegClientMsgToSendEv()

The *register-client* calls this event whenever a *register-client* related message is ready to be sent. You can use this callback for changing or examining a message before it is sent.

RvSipRegClientMsgReceivedEv()

The *register-client* calls this event whenever a *register-client* related message has been received and is about to be processed. You can use this callback to examine incoming messages.

RvSipRegClientFinalDestResolvedEv()

Indicates that the *register-client* is about to send a message after the destination address was resolved. This callback supplies the final *message* and the *transaction* that is responsible for sending this message. Changes in the message at this stage will not effect the destination address. When this callback is called, the application can query the *transaction* about the destination address using the RvSipTransactionGetCurrentDestAddress() function. If the application wishes, it can update the sent-by part of the top-most Via header. The application must not update the branch parameter.

Note To change the destination address resolved from the *message*, the application must use the Transmitter API. The application should first get the *transmitter* from the *transaction* using the RvSipTransactionGetTransmitter() API function. In can then manipulate the DNS list current destination address of the *transmitter* before the message is sent.

Register-Client State Machine

The Register-Client state machine represents the state of the registration process. The RvSipRegClientStateChangedEv() callback reports *register-client* state changes and state change reasons. The state change reason indicates how the *register-client* reached the new state. Some of the *register-client* states are basic and common to most registration scenarios. Advanced *register-client* states depend on SIP Stack configuration.

The *register-client* associates with the basic states described below.

Basic Register-Client States

RVSIP_REG_CLIENT_STATE_IDLE

The IDLE state is the initial state of the Register-Client state machine. Upon *register-client* creation, the Register-Client assumes the IDLE state. The *register-client* remains in this state until RvSipRegClientRegister() is called, whereupon it moves to the REGISTERING state.

RVSIP_REG_CLIENT_STATE_REGISTERING

After calling RvSipRegClientRegister() and sending a REGISTER request, the *register-client* enters the REGISTERING state. The *register-client* remains in this state until it receives a final response from the Registrar. If a 2xx class response is received, the *register-client* assumes the REGISTERED state. If a 3xx class response is received, the *register-client* moves to the REDIRECTED state. If 401 or 407 responses are received, the *register-client* moves to the UNAUTHENTICATED state. If the REGISTER request is rejected with a 4xx, 5xx or 6xx class response—other than 401 and 407—the *register-client* assumes the FAILED state. If no final response is received before time-out, the *register-client* assumes the FAILED state.

RVSIP_REG_CLIENT_STATE_REDIRECTED

A *register-client* in the REGISTERING state can receive a 3xx class response. In this case, the *register-client* assumes the REDIRECTED state. The *register-client* has already updated the Registrar address according to the first Contact header received in the 3xx class response. You can update the Registrar address with any other preferred Registrar address. At this point, you can confirm the redirection of the REGISTER request by calling the RvSipRegClientRegister() function. You can also decide to terminate the *register-client* using the RvSipRegClientTerminate() function.

RVSIP_REG_CLIENT_STATE_UNAUTHENTICATED

A *register-client* in the REGISTERING state can receive a 401 or 407 response. In this case, the *register-client* assumes the UNAUTHENTICATED state. At this point, you may re-send the registration request with authentication information by calling RvSipRegClientAuthenticate(). You can also terminate the register-client using the RvSipRegClientTerminate() function.

RVSIP_REG_CLIENT_STATE_REGISTERED

This state indicates that the *register-client* successfully registered with the Registrar. The *register-client* reaches this state when a 2xx final response is received. The *register-client* is not terminated although the registration process has successfully terminated. You can use this *register-client* for refreshing the registration by re-using the RvSipRegClientRegister() function. To terminate a *register-client* in this state, use the RvSipRegClientTerminate() function.

RVSIP_REG_CLIENT_STATE_FAILED

When a *register-client* receives a response of class 4xx, 5xx or 6xx—except 401 or 407 responses— the *register-client* assumes the FAILED state. The application is responsible for terminating a *register-client* that reaches the FAILED state.

RVSIP_REG_CLIENT_STATE_TERMINATED

This state is the final register-client state. When a *register-client* is terminated, the *register-client* assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *register-client*.

BASIC STATE MACHINE

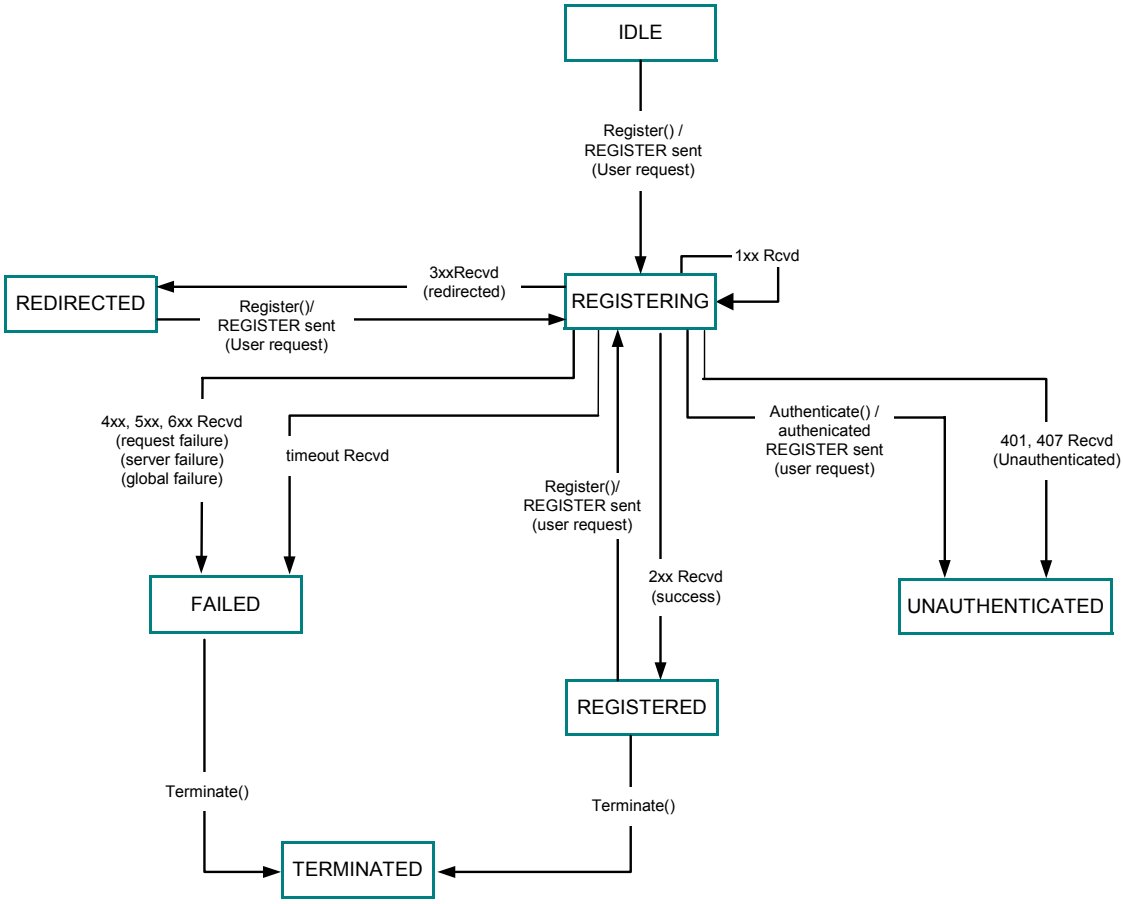


Figure 8-1 *Register-Client State Machine*

ADVANCED REGISTER-CLIENT STATES

The *register-client* associates with the advanced states described below.

RVSIP_REG_CLIENT_STATE_MSG_SEND_FAILURE

This state is assumed only if the SIP Stack works with the Enhanced DNS feature. The *register-client* moves to this state when it failed to send a request (the *register-client* received a network error, 503 response or time-out on the request). In this state your application can:

- Continue DNS—try to send the request to the next address in the *transaction* DNS list. For more information see the Working with DNS chapter.
- Give up—return to the previous state of the *register-client* and not send the request. For more information see the Working with DNS chapter.
- Terminate the register-client.

This state can be reached from the REGISTERING state only.

Message Send Failure State Machine

The following state machine is part of the Register-Client state machine and is shown here separately for simplicity. The Message Send Failure state machine shows how the RVSIP_REG_CLIENT_STATE_MSG_SEND_FAILURE state is reached.

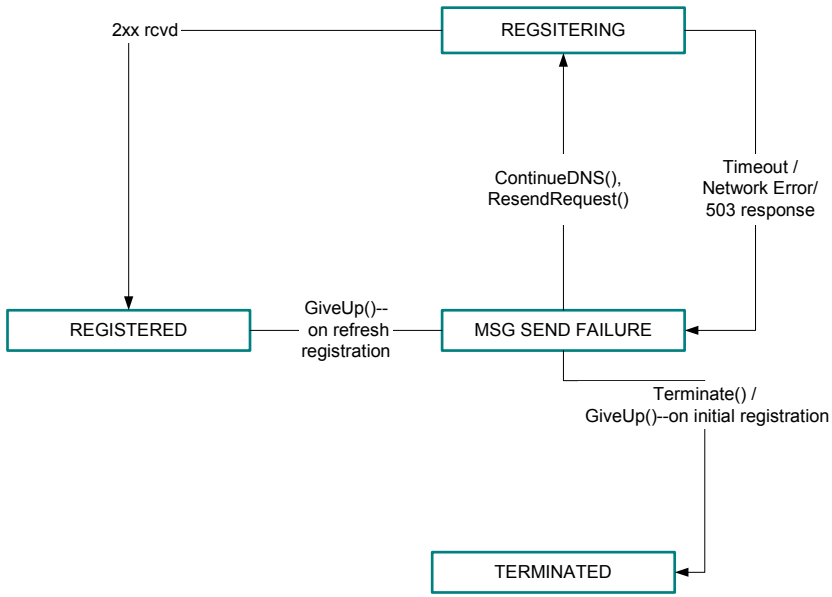


Figure 8-2 *Message Send Failure State Machine*

Register-Client Refresh Mechanism

When a REGISTER request receives a 2xx class response, the *register-client* assumes the REGISTERED state. This state indicates that the register-client is registered with the requested Registrar according to its To, From, Expires and Contact headers. The *register-client* is not terminated and remains in the REGISTERED state until the application causes a change in the state of the object.

The application can either terminate the *register-client* using the RvSipRegClientTerminate() function, or refresh the object registration using the RvSipRegClientRegister() function. Calling the RvSipRegClientRegister() function causes the *register-client* to use internal data to begin a new registration process.

A new REGISTER request is sent to the Registrar, containing the To, From, Expires and Contact headers associated with the register-client. You can use the refresh mechanism to change your registration information with the Registrar by updating the expiration time-outs and adding new Contact headers to the list. You can also use this mechanism to register with a different Registrar, by changing the Registrar address before calling RvSipRegClientRegister().

If your application is running on low resources and you prefer the register-client object to be terminated immediately after a successful registration, use the RvSipRegClientTerminate() function as soon as you have been notified of the REGISTERED state. This function frees the *register-client* but does not terminate the registration with the Registrar.

Register-Client Manager API

The *Register-ClientMgr* controls the SIP Stack collection of Register-Clients. You use the Register-Client Manager API to set application callbacks with the SIP Stack and to create new *register-clients*.

RvSipRegClientMgrCreateRegClient()

You can use RvSipRegClientMgrCreateRegClient() to create a new register-client object.

RvSipRegClientMgrSetEvHandlers()

You use RvSipRegClientMgrSetEvHandlers() to set your *register-client* associated event handler (callback function) pointers in the SIP Stack.

The Register-Client API declares prototypes for all Register-Client callback functions. For example:

```
/*=========================================================================*/
typedef void
(RVCALLCONV * RvSipRegClientStateChangedEv)(
       IN          RvSipRegClientHandle              hRegClient,
       IN          RvSipAppRegClientHandle           hAppRegClient,
       IN          RvSipRegClientState               eNewState,
       IN          RvSipRegClientStateChangeReasonm  eReason);

/*=========================================================================*/
```

You can implement any callbacks you find necessary, according to the callback function prototypes in the Register-Client API. All callback functions are included in a structure called RvSipRegClientEvHandler. This event handler structure is where you should set your callback function pointers and is given as a parameter to RvSipRegClientMgrSetEvHandlers(). The *register-client* notifies you when the event occurs using the callback functions you implemented.

If you do not wish to be notified of certain events, enter a NULL value for the callback functions that manage those events in RvSipRegClientEvHandler. Although you can set the event handlers at any time, it is customary to set them immediately after RvSipStackConstruct() so that the application immediately receives all necessary notifications.

Registering Application Callbacks

To register an application callback, you must first define the callback according to the prototype. The following code demonstrates an implementation of the RvSipEventStateChanged() callback function.

Sample Code

```
/*=========================================================================================*/
/*Implements the register-client state changed event handler. Prints the handle of each
 successfully registered register-client and terminates it.*/
void AppRegClientStateChangedEvHandler (
         IN  RvSipRegClientHandle            hRegClient,
         IN  RvSipAppRegClientHandle         hAppRegClient,
         IN  RvSipRegClientState             eNewState,
         IN  RvSipRegClientStateChangeReason eReason)
{

         if (eNewState == RVSIP_REG_CLIENT_STATE_REGISTERED)
         {
                printf("Register-Client %x was successfully
                       registered\n", hRegClient);
                RvSipRegClientTerminate(hRegClient);
         }
}
/*=========================================================================================*/
```

The following steps describe how to register your application callbacks:

To register application callbacks

1. Declare a RvSipRegClientEvHandlers structure.
2. Initialize all the structure members to zero using memset().
3. Set the application defined callbacks to the RvSipRegClientEvHandlers structure.
4. Call RvSipRegClientMgrSetEvHandlers() with the initialized structure.

Sample Code

The following code demonstrates an implementation of application callback registration.

```
/*=========================================================================================*/
void SetRegisterClientEvHandlers(
                        RvSipRegClientMgrHandle hMgr)
{
  /*Step 1*/
  RvSipRegClientEvHandlers appEvHandlers;
  /*Step 2*/
  memset(&appEvHandlers, 0,
                sizeof(RvSipRegClientEvHandlers));
  /*Step 3*/
  appEvHandlers.RvSipRegClientStateChangedEv = AppRegClientStateChangedEvHandler;
  /*Step 4*/
  RvSipRegClientMgrSetEvHandlers
         (hMgr,
         &appEvHandlers,
         sizeof(appEvHandlers));
}
/*=========================================================================================*/
```

Exchanging Handles with the Application

The SIP Stack enables you to create your own handle to a register-client. This will prove useful when you have your own application *register-client* database.

You can give your handle to the SIP Stack when calling RvSipRegClientMgrCreateRegClient() which it will then supply when calling your application callbacks.

Global Call-ID

The *Register-ClientMgr* is responsible for generating the Call-ID once and supplying it to all non-stand-alone *register-clients*. This Call-ID value can be set to the *Register-ClientMgr*. If you choose to set a different Call-ID header value to the manager—instead of using the generated Call-ID header value—you must do it only once, immediately after the SIP Stack is constructed. The ability to set the Call-ID allows a client to use the same Call-ID header value within more than one reboot cycle.

Initiating a Register-Client

The following steps describe a simple way to register to a registrar using the Register-Client API. The following code is an example of the implementation of these steps.

To initiate a Register-Client

1. Declare a handle to the new register-client.
2. Define the To, From, Registrar and Contact addresses as strings.
3. Call the RvSipRegClientMgrCreateRegClient() function.

   This enables you to exchange handles with the SIP Stack.
4. Call the RvSipRegClientMake() function, supplying the handle to the *register-client* and the string addresses defined in step 3.

   Calling this function sends a REGISTER request to the requested Registrar.

Note The RvSipRegClientMake() function lets you set only one contact header. If you want to set more than one contact, call the RvSipRegClientSetContactHeader() function before calling the make function.

Sample Code (next page)

```
/*===========================================================================*/
void AppSimpleRegister(
                      IN RvSipRegClientMgrHandle hRegClientMgr)
{
        RvStatus                rv;
        /*-----------------------------------------------------
        1. Declares a handle to the new register-client.
        -----------------------------------------------------*/
        RvSipRegClientHandle     hRegClient;

        /*-----------------------------------------------------
        2. Defines the To, From, Registrar, and Contact addresses
         as strings.
        -----------------------------------------------------*/
        RvChar  *strFrom       = "From:sip:172.20.3.38";
        RvChar  *strTo         = "To:sip:me@172.20.2.151";
        RvChar  *strContact    = "Contact:sip:172.20.3.38";
        RvChar  *strRegistrar  ="sip:172.20.2.151";

        /*-----------------------------------------------------
```

```
        3. Creates a new register-client.
        ----------------------------------------------------*/
        rv = RvSipRegClientMgrCreateRegClient(hRegClientMgr,
                                        NULL, &hRegClient);
        if (rv != RV_OK)
        {
                printf("Failed to create new register-client");
                return;
        }

        /*----------------------------------------------------
        4. Calls the "make" function to register with the
        register-client.
        ----------------------------------------------------*/
        rv = RvSipRegClientMake(hRegClient, strFrom, strTo,
                            strRegistrar, strContact);
        if (rv != RV_OK)
        {
                printf("Register request failed for register-
                       client");
                return;
        }
}
/*===========================================================================*/
```

9

WORKING WITH SIP MESSAGES

INTRODUCTION

The SIP Stack provides a flexible API for working with SIP messages and message parts, such as headers and addresses. The SIP Stack uses object-oriented methodology and treats messages and message parts as objects.

The Message API provides functions for working with the following objects:

- Message
- Address
- Body
- Body Part

Header objects:

- Allow header
- Allow-Events header
- Authentication header (Proxy-Authenticate and WWW-Authenticate headers)
- Authorization header (Authorization and Proxy-Authorization headers)
- Contact header
- Content Disposition header
- Content Type header
- CSeq header
- Date header
- Event header
- Expires header

- Min-SE header
- Other header
- Party header (To and From headers)
- RAck header
- Refer To header
- Referred By header
- Replaces header
- Retry-After header
- Route Hop header (Route, Record-Route, Service-Route and Path headers)
- RSeq header
- Session-Expires header
- Subscription-State header
- Via header

Working with Handles

Message API functions define handles for the different types of *messages*. For example:

- RvSipMsgHandle—defines the handle to a *message*.
- RvSipViaHeaderHandle—defines the handle to a Via header object.
- RvSipAddressHandle—defines the handle to an Address object.

When you construct an object, the Construct() function returns the handle to the newly created object. Whenever performing an operation on the object, you must supply the object handle to the function.

Message Manager Object

The *MessageMgr* manages the collection of *messages* and message parts that are not related to a specific *message*. It is used to construct stand-alone *messages* and message parts. For more information, see Creating a Stand-alone Header and Creating a New SIP Message.

You can get the *MessageMgr* handle by calling the RvSipStackGetMsgMgrHandle() function.

MESSAGE OBJECTS

A *message* holds all message parts, which include the message start line, one or more header fields and an optional message-body. The *message* holds most of its headers in a sequential list. The Call-ID, To, From, CSeq, Content-Type and Content-Length headers are held separately. These headers can only appear once in a SIP *message*. Figure 9-1 illustrates a SIP *message*.

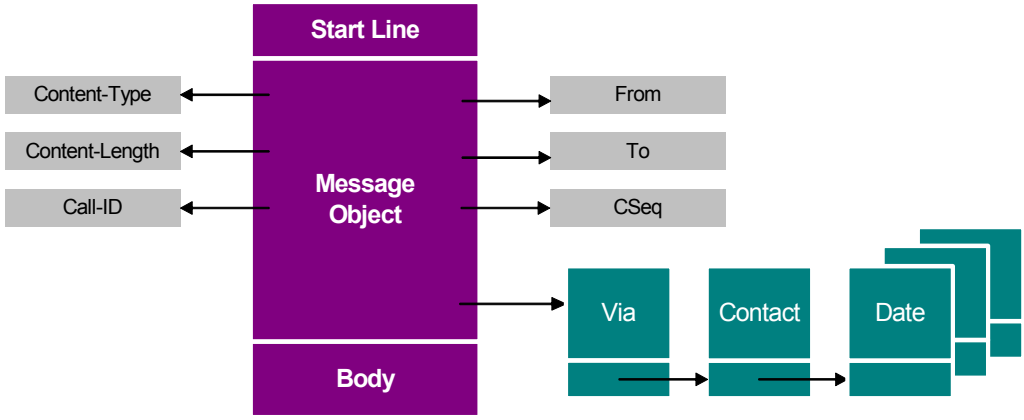


Figure 9-1 *Message Object*

HEADER OBJECTS

The SIP Stack has several types of header objects. Each header object represents a SIP header. The header object holds the header fields according to the header BNF definition. Figure 9-2 illustrates a To header object. (To and From SIP headers are both kept in a Party header object.)

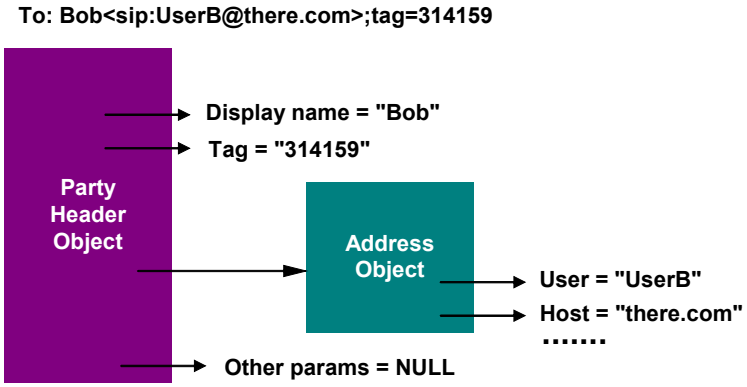


Figure 9-2 *Header Object*

SIP headers that do not have a dedicated object in the SIP Stack are held in an Other header object. The Other header object has two fields—header name and header value. Using the Other header object the SIP Stack can hold any type of SIP header.

Figure 9-3 illustrates an Accept-Encoding SIP header. Since this type of header does not have a dedicated object, it is held in an Other header object

.

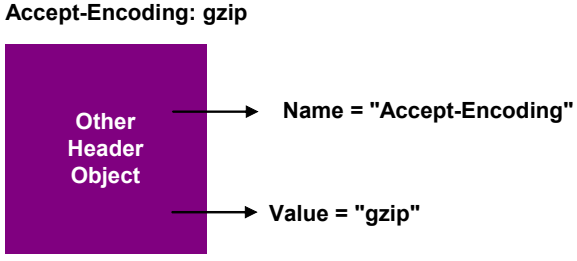


Figure 9-3 *Accept-Encoding SIP Header*

SIP Stack Message API

SIP Stack Message API functions can be categorized as follows:

Constructors

You use constructors to create new objects.

Get and Set Functions

Get and Set functions are the only way to access object fields. The following rules apply to Get and Set functions:

- **Setting parameters**

  The parameter you supply is always copied into the object.

- **Getting string parameters**

  You must supply a buffer. The Get function will copy the string into this buffer. You can use a special GetSize function to find out the required buffer size before performing the Get operation.

- **Getting non-string parameters**

  The actual value of the parameter is returned.

Encode and Parse Functions

SIP is a text-based protocol. RFC 3261 defines the syntax for each *message* and message part. You can use the encode functions to receive text strings that an object represents. The parse functions enable you to initialize an object from a formatted text string.

Copy Functions

Use the copy functions to create two similar objects. The copy functions copy all fields from one object to another.

Working with Headers

SIP Header API functions provide a set of access functions (including Get and Set) for reading and modifying various parameters of the headers.

Sample Code

The following code demonstrates how to get and set parameters in a Party header.

```
/*=========================================================================*/
void UsingHeaders(RvSipPartyHeaderHandle   ToHeader)
{
    int                length;
    RvChar*            strBuffer;
    RvSipAddressHandle hAddress;

    /*Gets and sets header parameters:*/

    /*Displays name.*/
    length = RvSipPartyHeaderGetStringLength(ToHeader, RVSIP_PARTY_DISPLAY_NAME);
    strBuffer = malloc(length);

    RvSipPartyHeaderGetDisplayName(ToHeader, strBuffer, length, &length);
    printf("got displayName %s", strBuffer);
    free(strBuffer);

    RvSipPartyHeaderSetDisplayName(ToHeader, "John");

    /*Address specification.*/
    hAddress = RvSipPartyHeaderGetAddrSpec(ToHeader);

    /*Gets and sets address parameters.*/
    length = RvSipAddrGetStringLength(hAddress, RVSIP_ADDRESS_HOST);

    strBuffer = malloc(length);
    RvSipAddrUrlGetHost(hAddress, strBuffer, length, &length);
    printf("The Host is %s", strBuffer);
    free(strBuffer);

    RvSipAddrUrlSetUser(hAddress, "UserAddressName");
}
/*=========================================================================*/
```

WORKING WITH SIP MESSAGES

The SIP Stack Message API provides a set of functions for accessing, encoding, parsing and adding new headers to *messages*.

READING AND MODIFYING SIP MESSAGES

The Message API provides a set of functions for reading and modifying various message parts such as the start-line field, headers and body.

You can access the To, From, CSeq, Call-ID, ContentLength, and ContentType headers directly since they are held separately in the *message* and can appear only once in a SIP *message*. All other headers are kept in a linked list. Some of these headers can appear more than once and you can get these headers from *messages* based on the header type, name or position in the list.

Sample Code

The following code demonstrates ways of manipulating a *message*:

```
/*=========================================================================================*/
RvStatus ProcessMessage(RvSipMsgHandle hMsg)
{
    RvSipMsgType msgType;
    RvSipPartyHeaderHandle fromHeader;
    RvSipViaHeaderHandle viaHeader;
    RvSipHeaderListElemHandle listElem;
    /*Checks if this is a request or response message.*/
    msgType = RvSipMsgGetMsgType(hMsg);
    if (msgType == RVSIP_MSG_REQUEST)
    {
        /*This is a request. Gets the method (request) type.*/
        RvSipMethodType methodType;
        methodType = RvSipMsgGetRequestMethod(hMsg);
        printf("the received message is a request, method type is %d",methodType);
    }
    else if (msgType == RVSIP_MSG_RESPONSE)
    {
        /*This is a response. Gets the response code.*/
        RvUint32 respCode;
        respCode = RvSipMsgGetStatusCode(hMsg);
        printf("the received message is a response, response code is %d", respCode);
    }
    else
    {
        /* msgType == RVSIP_MSG_UNDEFINED */
        printf("The received message is of undefined type.Fail");
```

```
        return RV_ERROR_UNKNOWN;
    }
    /*Gets the From header value--direct access.*/
    fromHeader = RvSipMsgGetFromHeader(hMsg);
    if(fromHeader != NULL)
    {
        /*Performs an action with the From Header.*/
    }
    /*Gets all the Via header values using the header list functions.*/
    viaHeader =
       (RvSipViaHeaderHandle)RvSipMsgGetHeaderByType(hMsg,
        RVSIP_HEADERTYPE_VIA,
        RVSIP_FIRST_HEADER,
        &listElem);
    while (viaHeader != NULL)
    {
        /*Performs an action with the Via header that was received.*/
        /*Gets the next one.*/
        viaHeader =
          (RvSipViaHeaderHandle)RvSipMsgGetHeaderByType(hMsg,
            RVSIP_HEADERTYPE_VIA,
            RVSIP_NEXT_HEADER,
            &listElem);
    }
    return RV_OK;
}
/*===========================================================================*/
```

ENCODING AND PARSING

The Message API provides a set of functions for encoding and parsing SIP *messages*.

ENCODING

To get the encoded format of a SIP *message* according to the specifications of RFC 3261, you should use the encoding functions. There are encode functions for *messages* and for each header of a *message*.

All the encode functions work in the same way, by getting the memory pool handle and constructing a new page for the encoded string. The function returns the handle to the page and the length of the encoded string. If you need the encoded string on a regular buffer, you should allocate the buffer and copy the encoded string from the page to the buffer.

For more information about the memory pool, see the Memory Pool chapter.

To encode a SIP message

1. Call the *RvSipMsgEncode()* function with the *message* handle and the pool handle. The *Encode()* function receives the pool handle and returns a handle to the new page containing the encoded string and the length of the encoded string.
2. Allocate a buffer of the same length as the retrieved length, +1 if you want to place '\0' at the end.
3. Call the *RPOOL_CopyToExternal()* function—with the offset parameter set to zero—to copy the encoded string from the memory pool page to the allocated buffer.

Note The encoded string in the memory page is not necessarily consecutive. Therefore, you must not use *strcpy*. Also, the string is not NULL-terminated so you should insert the NULL value manually if required.

Sample Code

The following code demonstrates how to encode a *message* and copy it to a new buffer and how to print the encoded *message* to the screen.

```
/*=========================================================================================*/
void EncodeMessage(RvSipMsgHandle hMessage, HRPOOL hPool)
{
  RvUint32  length = 0;
  HPAGE     hPage;
  RvStatus  status;
  RvChar    *msgBuf;

  status = RvSipMsgEncode(hMessage, hPool, &hPage, &length);
  if (status != RV_OK)
  {
            printf("RvSipMsgEncode failed. status is %d\n",status);
            return;
  }

  /*Allocate a consecutive buffer.*/
  msgBuf = malloc(length+1);

  /*Copies the encoded message to a consecutive buffer and sets '\ at the end of the
 string.*/
  status = RPOOL_CopyToExternal(hPool, hPage, 0, msgBuf, length);
  msgBuf[length] = '\0';
  if (status != RV_OK)
  {
                printf("RPOOL_CopyToExternal failed. status is %d\n",status);
  }
  else
  {
                /*print the message*/
                printf("%s",msgBuf);
  }

  /*Free resources.*/
  free(msgBuf);
  RPOOL_FreePage(hPool, hPage);
}
/*=========================================================================================*/
```

Parsing

To get a header object from a string encoded according to the specifications of RFC 3261, you should use the Parse() or ParseValue() function.

Sample Code

The following code demonstrates how to parse a textual Contact header into a Contact header object.

```
/*=========================================================================*/
void ContactHeaderParse(IN RvSipContactHeaderHandle hContact)
{
        RvStatus status;
        RvChar* strContact = "Contact: Carol Lee<sip:carol.lee@example.com>";

        status = RvSipContactHeaderParse(hContact,strContact);
        if(status != RV_OK)
        {
                printf("RvSipContactHeaderParse failed");
                return;
        }
}
/*=========================================================================*/
```

There are also general functions which are not specific to a header type for encoding and parsing headers. The names of these functions are RVSipHeaderXXX, where XXX = Encode, Parse, ParseValue.

Adding New Headers to a Message

To add new headers to a *message*, you should use the *xxxConstructInMsg()* functions.

If you want to add the new header to the header list, you should also declare whether you want it to be added to the top or bottom of the list.

Sample Code

The following code demonstrates the creation of new From and Contact headers.

```
/*=========================================================================================*/
RvStatus createHeadersInMessage
         (HRPOOL hPool, RvSipMsgHandle hMsg)
{
    RvStatus                  status;
    RvSipAddressHandle        uri;
    RvSipPartyHeaderHandle    fromHeader;
    RvSipContactHeaderHandle  contactHeader;

    /*Creates a From header in the message and creates and set a URL object in it.*/
    status = RvSipFromHeaderConstructInMsg(hMsg, &fromHeader);
    if(status!= RV_OK)
        return status;

    status = RvSipAddrConstructInPartyHeader(fromHeader, RVSIP_ADDRTYPE_URL, &uri);
    if(status!= RV_OK)
        return status;
    /*Sets the URL parameters by parsing the encoded URL or by setting parameters using
    the Set functions*/
    RvSipAddrParse(uri, "sip:john@acme.com");
    /*Creates a Contact header at the head of the message header list and sets the
     header parameters.*/
    status = RvSipContactHeaderConstructInMsg(hMsg, RV_TRUE, &contactHeader);
    if(status!= RV_OK)
        return status;

    /*Sets the Contact header parameters.*/
    RvSipContactHeaderSetDisplayName(contactHeader, "ContactName");

    /*Sets other parameters...*/
    return RV_OK;
}
/*=========================================================================================*/
```

STAND-ALONE HEADERS

A stand-alone header is a header object that is constructed independently of any particular *message*. You can set a stand-alone header in a *message*.

Creating a Stand-alone Header

When constructing a stand-alone header, you must provide the *MessageMgr* handle and a memory page. The *Construct()* function creates the header object on the given page and returns a handle to the new header object. For more information about the memory pool, see the Memory Pool chapter.

Sample Code

The following code demonstrates the creation of a stand-alone CSeq header.

```
/*=========================================================================*/
RvStatus CreateStandAlone(HRPOOL hPool,
              HPAGE hPage, RvSipMsgMgrHandle hMgr)
{
    RvStatus status;
    RvSipCSeqHeaderHandle hCSeq;
    /*Constructs a standalone CSeq header.*/
    status = RvSipCSeqHeaderConstruct(hMgr, hPool, hPage, &hCSeq);
    if(status!= RV_OK)
    {
        printf("RvSipCSeqHeaderConstruct failed.Status is %d", status);
        return status;
    }
    /*Sets the CSeq header parameters.*/
    RvSipCSeqHeaderSetStep(hCSeq, 12);
    RvSipCSeqHeaderSetMethodType(hCSeq, RVSIP_METHOD_INVITE, NULL);
    return RV_OK;
}
/*=========================================================================*/
```

Setting a Stand-alone Header in a Message

There are two types of functions you can use to insert stand-alone headers in a *message*. The function type you use depends on the headers you wish to insert. The function types you may use are as follows:

- **Set Functions**

  Use Set functions to insert To, From, CSeq, Content-length, Content-Type and Call-ID headers in a *message*. These headers are not kept in the header list, so you can set them directly in the *message* using the appropriate set function.

- **Push functions**

  Use Push functions to insert all headers that are kept in the header list of the *message*, such as Allow, Via, Contact and Other headers.

Sample Code

The following code demonstrates how to insert CSeq and Via headers into a *message*:

```
/*-----------------------------------------------------------*/
RvStatus SetStandAloneInMsg(RvSipMsgHandle hMsg,
                        RvSipCSeqHeaderHandle hCSeq,
                        RvSipViaHeaderHandle  hVia)
{
    RvSipHeaderListElemHandle listElem;

    /*Sets CSeq header in the message.*/
    RvSipMsgSetCSeqHeader(hMsg, hCSeq);

    /*Pushes the Via header in the message.*/
    RvSipMsgPushHeader(hMsg, RVSIP_LAST_HEADER,
                       (void*)hVia,
                        RVSIP_HEADERTYPE_VIA,
                        &listElem,
                        (void**)&hVia);

    return RV_OK;
}
/*-----------------------------------------------------------*/
```

Removing Headers from a Message

You can remove headers from a *message* as follows:

- You can remove headers to which you have direct access, such as To, From and CSeq, by calling the appropriate Set function with NULL. For example:

  `RvSipMsgSetFromHeader(hMsg,NULL)`

- You can remove headers kept in the header list, such as Via and Allow, by calling the list remove function. For example:

  `RvSipMsgRemoveHeaderAt(hMsg,hListElem)`

Creating a New SIP Message

You can create a new *message* by performing the following steps:

To create a new message

1. Create a *message* with the *Construct()* function.
2. Set the start-line values: request-line values, if this is a request, or the status-line values, if this is a response.
3. Add headers as required.
4. Add a message body as required.

Sample Code

The following code demonstrates how to create a new *message*.

```
/*-----------------------------------------------------------------------------*/
RvSipMsgHandle createMessage(
                        RvSipMsgMgrHandle hMgr,
                        HRPOOL            hPool)
{
    RvSipMsgHandle         hMsg;
    RvStatus               status;
    RvSipAddressHandle     uri;
    RvInt                  length;
    RvSipPartyHeaderHandle toHeader;

    /*Constructs a new message object.*/
    status = RvSipMsgConstruct(hMgr, hPool, &hMsg);
    if(status!= RV_OK)
        return NULL;

    /*Sets the status-code for the status-line.*/
    RvSipMsgSetStatusCode(hMsg, 200, RV_TRUE);

    /*Creates and sets the To header with a URL value.*/
    status = RvSipToHeaderConstructInMsg(hMsg, &toHeader);
    if(status!= RV_OK)
        return NULL;

    status = RvSipAddrConstructInPartyHeader(toHeader, RVSIP_ADDRTYPE_URL, &uri);
    if(status!= RV_OK)
```

```
            return NULL;

        /*Sets the URL parameters by parsing the encoded URL or setting parameters
         with the Set functions.*/
        RvSipAddrParse(uri, "sip:john@acme.com");

        /*Adds other headers as needed...*/

        /*Adds message body.*/
        RvSipMsgSetBody(hMsg, "This is the message body");

        return hMsg;
    }
    /*=========================================================================================*/
```

Using Compact Form

SIP provides a mechanism for representing header field names in a compact form. For example, the header, "`from: Bob <sip:bob@proxy.com>`" will become "`f: Bob <sip:bob@proxy.com>`" when compact form is used.

"Set" Compact Form Functionality

Table 9-1 represents the headers that can accept compact form and the API functions that should be used to set compact form to each header.

Table 9-1 *Setting Compact Form*

Header Name	Compact Form	Functions to Set Compact Form
To	t	RvSipPartyHeaderSetCompactForm()
From	f	RvSipPartyHeaderSetCompactForm()
Via	v	RvSipViaHeaderSetCompactForm()
Contact	m	RvSipContactHeaderSetCompactForm()
Allow-Events	u	RvSipAllowEventsHeaderSetCompactForm()
Content-Type	c	RvSipContentTypeHeaderSetCompactForm()
Event	o	RvSipEventHeaderSetCompactForm()
Refer-To	r	RvSipReferToHeaderSetCompactForm()
Referred-By	b	RvSipReferredByHeaderSetCompactForm()
Session-Expires	x	RvSipSessionExpiresHeaderSetCompactForm()
Call-ID	i	RvSipMsgSetCallIdCompactForm()
Content-Length	l	RvSipMsgSetContentLengthCompactForm()

As you can see in Table 9-1, the compact form for Call-ID and Content-Length is set by the *message* itself. The reason is that these two headers are not kept as objects but as string and integer.

"Get" Compact Form Functionality

The SIP Message API also provides a Get function for each of the above headers, which lets the application check if a header uses compact form. The function format is RvSipXXXHeaderGetComactForm(). An exception is the Call-ID and Content-Length headers were the format is

RvSipMsgGetXXXCompactForm().

Forcing Compact Form on the Entire Message

The application can instruct a *message* to force all of its headers to use compact form. This is done by calling the function RvSipMsgForceCompactForm(). When such a *message* is encoded, all headers that can take compact form will be encoded with compact form.

Headers that handled by the Stack as Other headers and are added to the *message* by the application will not use compact form. It is the responsibility of the application to set such header names with compact form. Other headers that are added to the *message* by the Stack, such as Supported header, will be encoded with compact form.

Note This flag effects only the encoding results of full *messages*. If you retrieve a specific header from the *message* and encode it, it will not be encoded with compact form if this specific header is not marked with compact form.

Note The Address header list and headers in multipart mime bodies are not effected by this flag and will not be encoded with compact form.

Handling Messages with Syntax Errors

When the SIP Stack receives an incoming SIP *message*, the *message* can contain headers with various syntax errors. The SIP Stack default behavior (which was the only possible behavior until version 3.0) is to automatically discard incoming *messages* with syntax errors as if the *message* was never received. version 3.0 introduces other possibilities to handle such *messages*. In this chapter, a *message* with syntax errors is referred to as a “bad syntax message”.

Bad Syntax Parameter

Version 3.0 introduces a new parameter that is added to each of the message parts, called StrBadSyntax. The Stack parses bad syntax *messages* as regular *messages*. The *message* still holds all message parts including start line, separate headers, header list, and message body. However, a message part that has a syntax error is held entirely as a string in the StrBadSyntax parameter and all other fields are set to NULL or UNDEFINED (-1).

Bad Syntax Header

The general grammar of any SIP header is “header-name: header-value”. When the SIP Stack parses a header, it first constructs the header object according to the header-name part and then parses the header-value into the object-specific fields.

If the header-value contains syntax errors, it is kept in the object as a single string in the StrBadSyntax parameter. All other header fields are set to NULL or UNDEFINED. Figure 9-4 illustrates how the *message* holds a To header object with a syntax error.

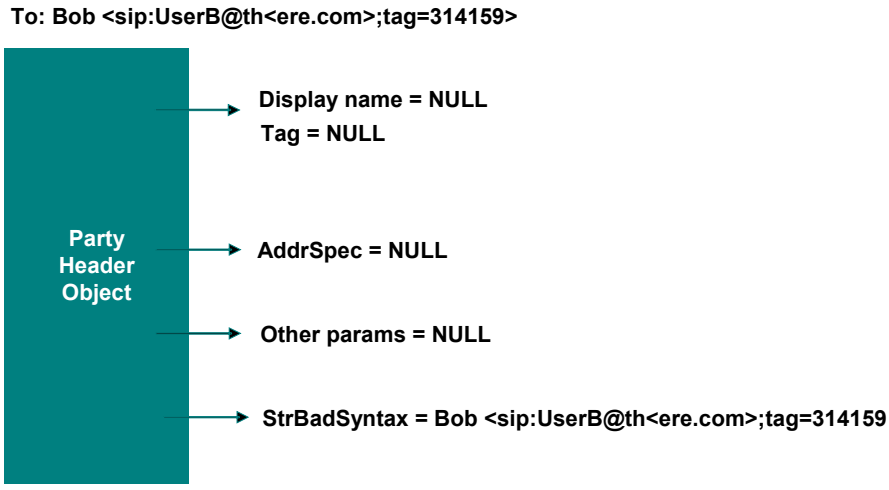


Figure 9-4 *Example of Syntax Error*

Bad Syntax Start Line

The start-line of a *message* can also include syntax errors. The *message* has specific fields to hold start line parameters of requests and responses.

When a *message* has a defected start line, the start line string is kept entirely in the StrBadSyntaxStartLine parameter and all other start line parameters remain empty. A SIP *message* with a bad syntax start line does not have a defined type and the *message* is not recognized as a request or response.

Handling Bad Syntax Messages

As explained above, the SIP Stack parses *messages* with syntax errors into regular *message* and header objects. The SIP Stack consults the application on how to handle incoming *messages* with syntax errors using callback functions. Whenever a *message* is received with a syntax error a suitable callback function is called.

In the callback functions, the application can query the *message* for its bad message parts and fix syntax errors. The application should then instruct the Stack on how to continue with the *message* processing with one of the following options:

- Discard the *message*
- Reject the *message*
- Continue *message* processing

The three options are defined in the RvSipTransportBsAction enumeration, where "Bs" stands for bad syntax. For more information, see the *SIP Stack Reference Guide*.

"Discard Message" Option

The application can instruct the Stack to discard bad syntax *messages* using the RVSIP_TRANSPORT_BS_ACTION_DISCARD_MSG option. Using this option will cause the Stack to ignore the *message* as if it was never received. This option can be used to maintain the behavior of version 2.2 applications. This is also the default behavior of the Stack.

"Reject Message" Option

The application can instruct the Stack to reject bad syntax *messages* using the RVSIP_TRANSPORT_BS_ACTION_REJECT_MSG option. This option is relevant only for defected request *messages*. Choosing this option causes the Stack to reject incoming bad syntax requests with 400 response code. The reject *message* reason phrase will include the syntax-error information. If you choose the reject *message* option on a bad syntax response, the Stack will discard this response *message*.

"Continue Message Processing" Option

The application can instruct the Stack to continue processing bad syntax *messages* using the RVSIP_TRANSPORT_BS_ACTION_CONTINUE_PROCESS option. Choosing this option instructs the Stack to continue the *message* processing even though the *message* contains bad syntax elements. If, in any of the processing stages, the stack finds that an essential header has syntax errors, requests will still be rejected with 400 response code and responses will be ignored.

If, for example, the Stack receives a bad syntax INVITE request and the error is in the Via header, the Stack will not be able to create a transaction, since the Via header is part of the *transaction* key. In this case, the Stack will reject the request with 400 response code.

However, if the Stack receives a bad syntax INVITE request and the syntax error is in a Date header, which has no logical meaning for the SIP Stack, the *message* will be handled as usual.

Figure 9-5 illustrates how the Stack handles a bad-syntax *message* across layers when the application chooses the "continue processing" option.

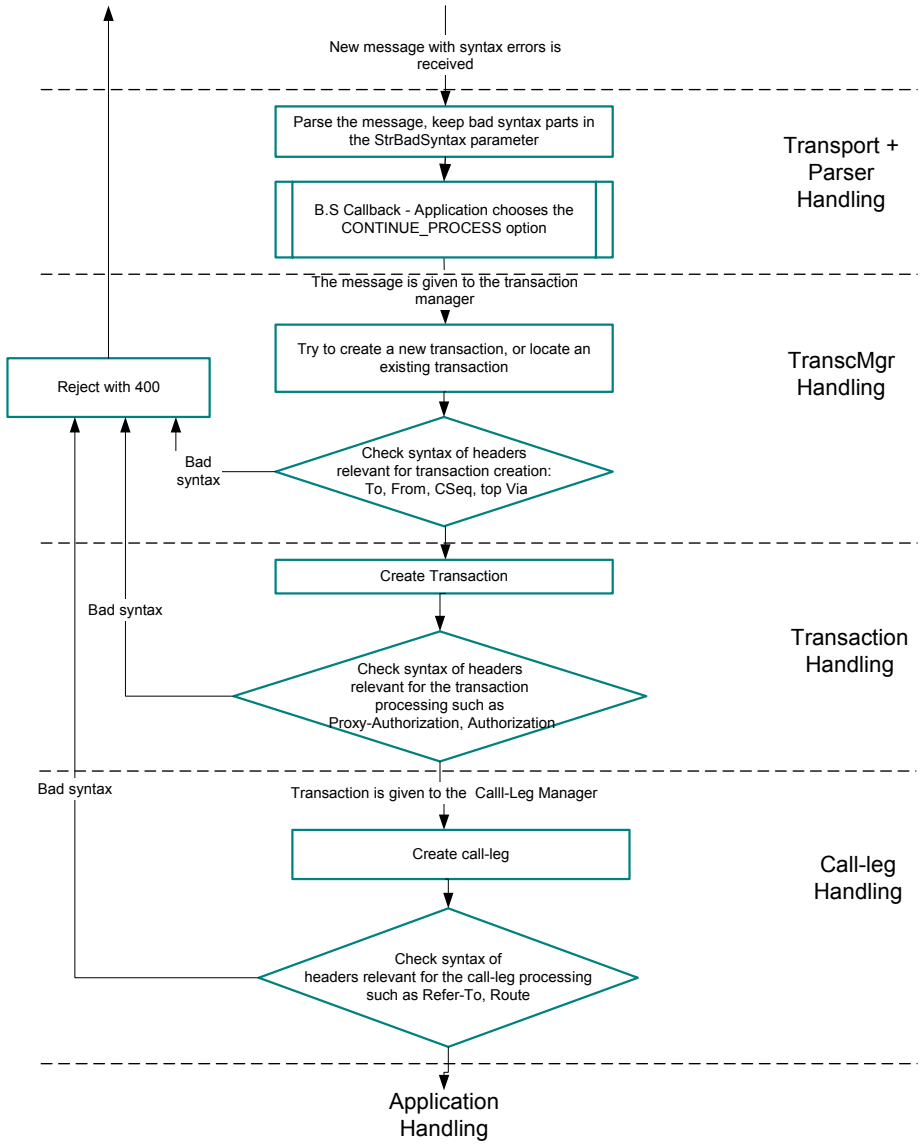


Figure 9-5 *Handling a Bad-syntax Message*

Fixing Bad Syntax Messages

There are cases where the application knows in advance that a certain header from the remote party will be received with a syntax error. This is usually the case when the application needs to work with another application that is not fully standard-compliant. The error can sometimes be crucial for SIP Stack *message* processing, such as an error in the top Via header.

The SIP Stack gives the application the option of fixing such syntax errors using a dedicated API. When the bad syntax callback is called, the application can query the *message* for the bad syntax element, fix the element with the relevant API function and then choose the "continue processing" option.

Bad Syntax Events (Callbacks)

The SIP Stack supplies two events, in the form of callback functions, for bad syntax *message* control. If these events are not implemented, the Stack uses its default behavior.

In order to receive an event, your application should define the event handler function and pass the event handler pointer to the *TransportMgr*. When a bad syntax *message* is received, the relevant event handler will be called using the supplied pointer.

The following events for bad syntax *message* handling are supplied with the Transport API:

RvSipTransportBadSyntaxMsgEv()

Notifies the application that a bad syntax *message* was received. This callback indicates that the syntax error is in one of the *message* headers. The callback supplies the application with the bad syntax *message*. The application can use the Message API and retrieve all the bad-syntax headers, fix, or remove them from the *message*. Using this callback the application should instruct the Stack on how to handle the *message*—discard, reject or continue processing.

If you do not implement this event, the Stack will discard bad syntax *message*s.

RvSipTransportBadSyntaxStartLineMsgEv()

Notifies the application that a bad syntax *message* was received. This callback indicates that the syntax error is in the *message* start line. The callback supplies the application with the bad syntax *message*. The application can use the

Message API and retrieve the bad-syntax start line and fix it. Using this callback, the application should instruct the Stack on how to handle the *message*—discard, reject or continue processing.

Note A *message* with a bad syntax start line is not recognized as a request or a response. Therefore, if you wish the Stack to continue the *message* processing, you must fix the start line using the RvSipMsgStartLineFix() function. If you choose not to fix the start line, the *message* will be ignored.

If you do not implement this event, the Stack will discard *messages* with a bad syntax start line.

Note If the *message* contains a defected start line and defected headers, both the above callback functions are called. The RvSipTransportBadSyntaxStartLineMsgEv() is called first. If the application fixes the start line and chooses the "continue *message* processing" option, RvSipTransportBadSyntaxMsgEv() is also called.

BAD SYNTAX API

The Message layer provides several API functions to get and fix bad syntax elements in a *message*. The following API functions are available:

RvSipMsgGetBadSyntaxHeader()

Retrieves headers with a syntax error from a *message* according to a given location. The *message* holds most headers in a sequential list. However, To, From, Call-ID, CSeq, Content-Length, and Content-Type headers are held separately.

This function scans all headers in the *message*, (the ones that are in the header list and the ones that are not) and retrieves only headers with syntax errors. (This function treats all headers as if they were located in one virtual list. The virtual list includes the headers that are in the header list and the headers that are not.)

You can use this function in the RvSipTransportBadSyntaxMsgEv() callback to check and fix bad syntax headers.

RvSipMsgGetStrBadSyntaxStartLine()

Gets the bad syntax start line from a *message*. When a message is received and the start line contains a syntax error, the start line is kept as a separate bad syntax string. This function retrieves this string.

You can use this function in the RvSipTransportBadSyntaxStartLineMsgEv() callback to check and fix the defected start-line.

RvSipMsgGetHeaderExt(), RvSipMsgGetHeaderByTypeExt(), RvSipMsgGetHeaderByNameExt()

These three function extend the functionality of functions that already exists in the SIP Stack API. RvSipMsgGetHeaderExt() extends the RvSipMsgGetHeader() function; RvSipMsgGetHeaderByTypeExt() extends the RvSipMsgGetHeaderByType function; and RvSipMsgGetHeaderByNameExt() extends the RvSipMsgGetHeaderByName.

These functions enable the application to specify whether it wishes to get only legal headers or both legal and illegal headers. (An illegal header is a header with a bad-syntax string.)

Note Using the regular function retrieves only legal headers.

RvSipMsgStartLineFix()

Fixes a start-line with bad-syntax information. When a *message* is received with a bad syntax start line, the start line string is kept as a separate "bad-syntax" string in the *message*. Use this function in the RvSipTransportBadSyntaxStartLineMsgEv() callback to fix a defected start-line.

When fixing a start line, you need to supply a string with a correct start line. The Stack will parse this string into the *message*. If parsing succeeds, the function will place all fields inside the *message* start-line parameters and will remove the bad syntax string. If parsing fails, the start line bad-syntax string remains untouched.

RvSipHeaderIsBadSyntax()

This function identifies if a given header is or is not a bad-syntax header.

Additional Bad Syntax Functions

A SIP header has the following grammar: header-name: header-value. When a header contains a syntax error, the header-value is kept as a separate bad-syntax string. The following functions are supplied for each of the SIP Stack headers. "XXX" in the function name represents the header name.

RvSipXXXHeaderGetStrBadSyntax()

Gets the bad-syntax string from the header object. Use this function in the RvSipTransportBadSyntaxMsgEv() callback implementation to see the bad-syntax header-value.

RvSipXXXHeaderSetStrBadSyntax()

Sets a bad syntax string to the header object and marks the header as a bad-syntax header.

RvSipXXXHeaderFix()

Fixes a bad syntax header. This function parses a given correct header-value string to the supplied header object. If parsing succeeds, this function places all fields inside the object and removes the bad syntax string. If parsing fails, the bad-syntax string in the header remains as it was.

SAMPLE CODE

The following sample code shows how to handle *messages* with syntax errors.

Sample One: RvSipTransportBadSyntaxMsgEv() callback implementation

This sample code demonstrates an implementation of the RvSipTransportBadSyntaxMsgEv() callback function. In this sample, the application tries to fix the defected headers. If it fails, the application instructs the Stack to reject the *message*. Otherwise, the application chooses the “continue processing” option.

```
/*-------------------------------------------------------------------------------*/
RvStatus RVCALLCONV AppTransportBadSyntaxMsgEv(
           IN    RvSipTransportMgrHandle    hTransportMgr,
           IN    RvSipAppTransportMgrHandle hAppTransportMgr,
           IN    RvSipMsgHandle             hMsgReceived,
           OUT   RvSipTransportBsAction     *peAction)
{
    RvStatus                   rv = RV_OK;
    void*                      pHeader = NULL;
    RvSipHeaderType            eHeaderType;
    RvSipHeaderListElemHandle  hElem;

    *peAction = RVSIP_TRANSPORT_BS_ACTION_CONTINUE_PROCESS;

    /*Loops and gets all headers with syntax errors.*/
    pHeader = RvSipMsgGetBadSyntaxHeader (hMsgReceived, RVSIP_FIRST_HEADER, &hElem,
                                          &eHeaderType);
    while (pHeader != NULL)
    {
        /*Tries to fix the bad syntax header.*/
        rv = AppFixHeader(pHeader, eHeaderType);
        if(rv != RV_OK)
        {
             printf("fixing header failed. reject message");
             *peAction = RVSIP_TRANSPORT_BS_ACTION_REJECT_MSG;
             return RV_OK;
        }
        pHeader = RvSipMsgGetBadSyntaxHeader (hMsgReceived, RVSIP_NEXT_HEADER, &hElem,
                                              &eHeaderType);
    }

    printf("all headers were fixed. Continue processing the message");
    return RV_OK;
}
/*-------------------------------------------------------------------------------*/
```

Sample Two: Fixing a bad syntax CSeq header

The following sample code demonstrates how to fix a bad syntax CSeq header. A CSeq header has the following format: CSeq: 3 UPDATE

In this sample, the application receives a CSeq without the method part. To fix the CSeq header, the application takes the method from the *message* request line and uses the RvSipCSeqHeaderFix() function to fix the CSeq header.

```
/*-----------------------------------------------------------------------------------------*/
RvStatus AppFixCSeqHeaderInMsg(RvSipMsgHandle hMsg)
{
    RvStatus              rv;
    RvSipCSeqHeaderHandle hCSeq;
    RvInt32               cseqBadSyntaxLen;
    RvChar                *cseqBadSyntaxStr;
    RvChar                startLineMethodStr[10];
    RvInt32               startLineMethodStrLen = 10;
    RvChar                *cseqCorrectHeaderValueStr;

    hCSeq = RvSipMsgGetCSeqHeader(hMsg);
    if(hCSeq == NULL)
    {
        printf("no cseq header in given message");
        return RV_ERROR_NOT_FOUND;
    }
    /*Checks the length of the bad-syntax string, and allocates a buffer with the correct
    size.*/
    cseqBadSyntaxLen = RvSipCSeqHeaderGetStringLength(hCSeq, RVSIP_CSEQ_BAD_SYNTAX);
    if(cseqBadSyntaxLen == 0)
    {
        printf("cseq header is correct. no need to fix it.");
        return RV_OK;
    }

    cseqBadSyntaxStr = malloc(cseqBadSyntaxLen + 1);

    /*Gets the cseq bad-syntax string. Assuming that the CSeq header was "CSeq: 3", and the
    method was missing, the bad-syntax string will be "3".*/
    rv = RvSipCSeqHeaderGetStrBadSyntax(hCSeq,
                                        cseqBadSyntaxStr,
                                        cseqBadSyntaxLen+1,
                                        &cseqBadSyntaxLen);
    if(rv != RV_OK)
    {
        printf("Failed to get bad-syntax string from cseq header");
        return RV_ERROR_UNKNOWN;
```

```
    }

    /*Gets the method string from the message request line.*/
    rv = RvSipMsgGetStrRequestMethod(hMsg,
                                     startLineMethodStr,
                                     startLineMethodStrLen,
                                     &startLineMethodStrLen);
    if(rv != RV_OK)
    {
        printf("Failed to get method string from message start-line");
        return RV_ERROR_UNKNOWN;
    }

    /*Creates the correct header value string by concatenating the bad-syntax string with the
    method.*/
    cseqCorrectHeaderValueStr = malloc(cseqBadSyntaxLen + startLineMethodStrLen +1);

    sprintf(cseqCorrectHeaderValueStr,"%s \0", cseqBadSyntaxStr);
    strcat(cseqCorrectHeaderValueStr,startLineMethodStr);

    /*Fixes the Cseq header by giving the correct header value
       (for example "3 UPDATE").*/
    rv = RvSipCSeqHeaderFix(hCSeq, cseqCorrectHeaderValueStr);
    if(rv != RV_OK)
    {
        printf("Failed to fix cseq header");
        return RV_ERROR_UNKNOWN;
    }

    printf("cseq header was fixed");
    return RV_OK;
}
/*===========================================================================================*/
```

WORKING WITH MULTIPART MIME

Multipart MIME is a generic mechanism that allows the encapsulation and transfer of arbitrary chunks of data, both text and binary, within text *messages*. The SIP Specification allows the usage of multipart MIME as SIP *message* bodies. Multipart MIME, defined in RFC 2046, is a necessary capability for some features, such as ISUP *message* tunneling through SIP, which is part of SIP-T.

The SIP Stack Message API provides a set of functions that enables you to Construct, Parse and encode multipart MIME bodies.

Message Multipart Body Structure

The SIP Stack provides two types of objects for working with multipart MIME bodies—the Body object and the Body Part objects. Both the Body and the Body Part objects are identified using handles.

Similar to the Message API, when you construct a Body or Body Part object, the Construct() function returns the handle to the newly created object. Whenever performing an operation on the object, you must supply the object handle to the function.

Body Object

The Body object is identified using the RvSipBodyHandle. Each SIP *message* can hold only one Body object. A Body object includes the following elements (each of them may be empty):

- Content-Type header—defines the type of the Body, such as multipart MIME.
- Body string—a buffer that holds the Body information (textual or binary).
- List of Body Parts—when the Body is of type multipart MIME, it can hold a list of Body Parts instead of a Body string.

Body Part Object

The Body Part object is identified using the RvSipBodyPartHandle. A Body Part object includes the following elements (each of them can be empty):

- Headers list
- Content Disposition header
- Body object.

Note The Body and Body Part objects are defined recursively within one another.

Figure 9-6 shows the structure and relationship between the Body and the Body Part objects.

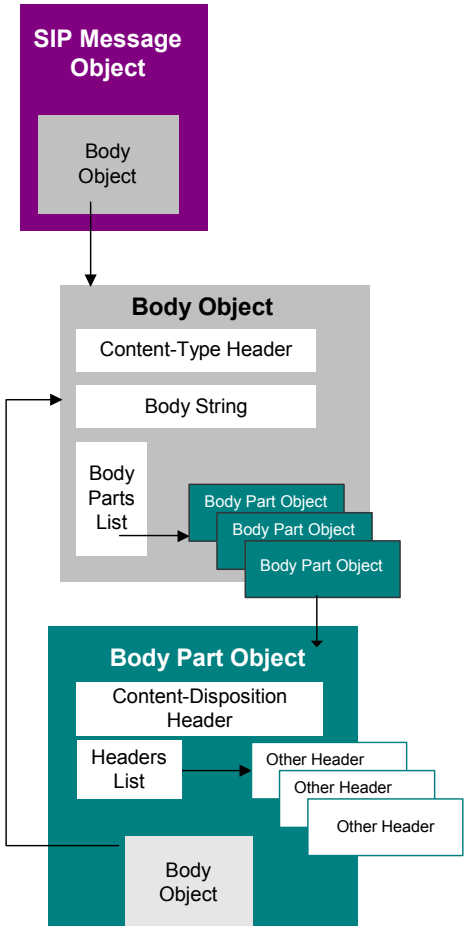


Figure 9-6 *Body and Body Part Objects*

Figure 9-7 is an example of a SIP *message* with a multipart MIME body. Each of the message parts is related to its corresponding object.

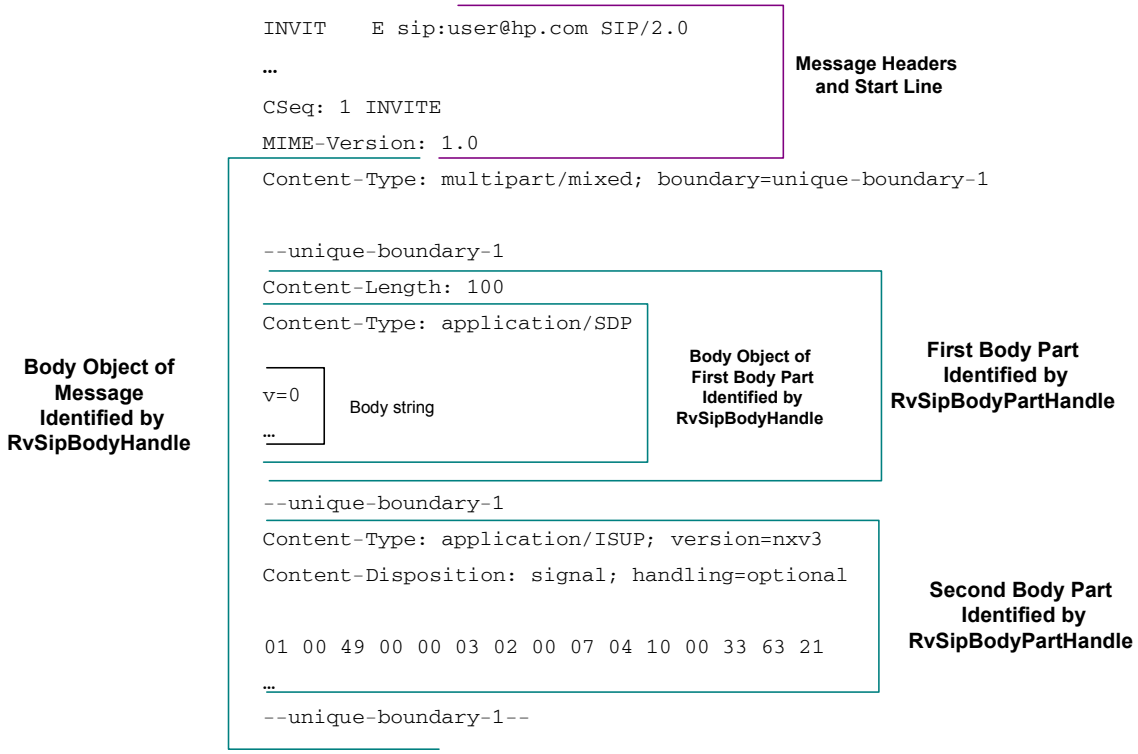


Figure 9-7 *Multipart SIP Message*

CONTENT-TYPE HEADER

The Body object contains a Content-Type header object identified by RvSipContentTypeHeaderHandle. The Content-Type header describes the media type of the Body. This applies both to a Body object within a SIP *message* and to a Body object within a Body Part object.

Sample Code

The following code demonstrates how to set a Content-Type header to a *message*.

```
/*=========================================================================================*/
void AppSetContentTypeToMsg(
            IN RvSipMsgHandle                hMsg,
            IN RvSipContentTypeHeaderHandle  hContentType)
{
        RvStatus               rv;
       RvSipBodyHandle          hBody;

         /*Gets the body object of the message.*/
      hBody = RvSipMsgGetBodyObject(hMsg);
      if (NULL == hBody)
      {
      /*Constructs a new body object if there is no body object for this message.*/
                rv = RvSipBodyConstructInMsg(hMsg, &hBody);
                if (RV_OK != rv)
                {
                             printf("Failed to construct body in message\n");
                             return;
                }
         }
         /*Sets the Content-Type header to the body object.*/
          rv = RvSipBodySetContentType(hBody, hContentType);
        if (RV_OK != rv)
         {
                        printf("Failed to set Content-Type in body\n");
         }
}
/*=========================================================================================*/
```

HEADERS IN THE BODY PART OBJECT

A Body Part object may contain the following MIME-part-headers, as described in RFC 2046:

- The Content-Type header of a Body Part is held within the Body object of the Body Part object, as described earlier. To modify or view the Content-Type header of a Body Part, get the Body object from the Body Part and use the functions, RvSipBodySetContentType() and RvSipBodyGetContentType().
- The Content-Disposition header of a Body Part is held within the Body Part object, represented by RvSipContentDispositionHeaderHandle. To modify and view

the Content-Disposition header of a Body Part, use the functions, RvSipBodyPartGetContentDisposition() and RvSipBodyPartSetContentDisposition().

- All headers other than Content-Type and Content-Disposition are held in a list within the Body Part object. RvSipOtherHeaderHandle represents each of these headers. This also refers to headers that have special objects defined for them in the SIP Stack, such as To, From, CSeq, etc. To add, remove, modify or view a header form the list use the functions, RvSipBodyPartPushHeader(), RvSipBodyPartGetHeader() and RvSipBodyPartRemoveHeader().

Parsing a Multipart Body

The SIP Stack allows you to parse a textual multipart Body into a list of Body Parts. In order to parse a multipart Body, the Body object must contain a Content-Type header with the multipart media type and a valid boundary. The SIP Stack parses the Body according to the boundary given in the Content-Type header.

After parsing a multipart Body, the Body object contains a list of Body Part objects.

Note After parsing the Body object, each of the Body Parts holds its headers as parsed and its Body as a string. If the Body Part contains a multipart Body, you have to specifically parse it (recursively).

You can view and modify the Body Part list of a Parsed Body using the functions, RvSipBodyGetBodyPart(), RvSipBodyPushBodyPart() and RvSipBodyRemoveBodyPart().

Sample Code

The following code demonstrates how to parse the Body of a *message* and afterwards view the Body Parts of which it consisted.

```
/*=========================================================================*/
void AppParseMultipartMixedBody(IN RvSipMsgHandle hMsg)
{
        RvSipBodyHandle             hBody;
        RvSipBodyPartHandle         hBodyPart;
        RvUint32                    length;
        RvChar                      *strBody;
        RvStatus                    rv;

        /*Gets the body object of the message.*/
        hBody = RvSipMsgGetBodyObject(hMsg);
        if (NULL == hBody)
        {
                return;
        }
        /*Gets the body string from the body object.*/
        length   = RvSipBodyGetBodyStrLength(hBody);
        strBody  = malloc((length)*sizeof(RvChar));
        rv = RvSipBodyGetBodyStr(hBody, strBody, length, &length);
        if (RV_OK != rv)
        {
                free(strBody);
                printf("Get body string failed");
                return;
        }
        /*Parse the body string.*/
        rv = RvSipBodyMultipartParse(hBody, strBody, length);
        if (RV_OK != rv)
        {
                free(strBody);
                printf("Parse multipart body failed");
                return;
        }
        free(strBody);

        /*Views the list of body parts.*/
        rv = RvSipBodyGetBodyPart(hBody, RVSIP_FIRST_ELEMENT, NULL, &hBodyPart);
```

```
        if (RV_OK != rv)
        {
                printf("Failed to get body part");
                return;
        }
        while (NULL != hBodyPart)
        {
                /*Do something with the body part ... */
                /*Gets the next body part in the list.*/
                rv = RvSipBodyGetBodyPart(hBody, RVSIP_NEXT_ELEMENT, hBodyPart,
                                          &hBodyPart);
                 if (RV_OK != rv)
                 {
                        printf("Failed to get body part");
                        return;
                }
        }
}
/*===========================================================================*/
```

CREATING A MULTIPART BODY

The SIP Stack allows you to build a multipart Body from several Body Part objects. You use the function, RvSipBodyPushBodyPart(), to add a new Body Part to the list of Body Parts in the Body object.

Sample Code

The following code demonstrates how to create a Body object from Body Parts that are given as strings.

```
/*=========================================================================================*/
void AppCreateMultipartBodyFromBodyParts(
               IN      RvChar           *strBodyPart1,
               IN      RvChar           *strBodyPart2,
               INOUT   RvSipBodyHandle   hBody)
{
      RvSipBodyPartHandle             hBodyPart;
      RvSipContentTypeHeaderHandle    hContentType;
      RvStatus                        rv;

      /*Constructs the Content-Type header within the given body object. Note that the
      received Body was constructed outside of this function.*/
      rv = RvSipContentTypeHeaderConstructInBody (hBody, &hContentType);
      if (RV_OK != rv)
      {
           printf("Failed to construct Content-Type header");
           return;
      }
      /*Sets the Content-Type to be of type multipart. A unique boundary will be generated by
      the Stack when you encode the Body object.*/
      RvSipContentTypeHeaderSetMediaType(hContentType, RVSIP_MEDIATYPE_MULTIPART, NULL);
      RvSipContentTypeHeaderSetMediaSubType(hContentType, RVSIP_MEDIASUBTYPE_ALTERNATIVE,
                                                                              NULL);

      /*Constructs a new Body Part within the given body object. The body part is constructed
      at the end of the body parts list.*/
      rv = RvSipBodyPartConstructInBody(hBody, RV_FALSE, &hBodyPart);
      if (RV_OK != rv)
      {
           printf("Failed to construct body part");
           return;
      }
       /*Parses a body part string.*/
      rv = RvSipBodyPartParse(hBodyPart, strBodyPart1,
                              strlen(strBodyPart1));
      if (RV_OK != rv)
      {
            printf("Failed to parse body part");
            return;
      }
```

```
      /*Constructs the new body part within the given body object.*/
      rv = RvSipBodyPartConstructInBody(hBody, RV_FALSE, &hBodyPart);
      if (RV_OK != rv)
      {
           printf("Failed to construct body part");
           return;
      }
      /*Parses the body part string.*/
      rv = RvSipBodyPartParse(hBodyPart, strBodyPart2, strlen(strBodyPart2));
      if (RV_OK != rv)
      {
           printf("Failed to parse body part");
           return;
      }
}
/*=========================================================================================*/
```

ENCODING A MULTIPART BODY

After building a Body object from multiple Body Part objects, you can encode the Body into a textual Body using the function, RvSipBodyEncode().

In order to encode Body of type Multipart, the Body object should contain a multipart Content-Type with a valid boundary parameter. Note the following:

- If the Body object you wish to encode does not contain a Content-Type header, the SIP Stack will add a Content-Type header and will initialize it to multipart/mixed media.
- If the Content-Type header does not contain a boundary parameter the SIP Stack will generate a unique boundary according to the rules defined in RFC 2046 and will set it to the Content-Type header.

BODY STRING

A Body object can be represented as a string or as a list of Body Parts

A multipart Body is represented as a string only when the Body object is not parsed, such as in the messageRecvd callback. You can get the Body string using the RvSipBodyGetBodyStrLength() and RvSipBodyGetBodyStr() functions, as shown in the following code. To set a Body string to a Body object, use the function, RvSipBodySetBodyStr().

After you parse a multipart Body object, or after you set a Body Part to the Body object, the Body string can no longer be retrieved. A Body with a list of Body Parts does not contain a string. In this case when calling the function, RvSipBodyGetBodyStrLength(), the returned length will be 0.

To check if a Body object is parsed, use the RvSipBodyGetBodyPart() function with location RVSIP_FIRST_ELEMENT. A non-NULL value will indicate a parsed Body.

A Body object that is not of type Multipart can be represented by a string only. You can get the Body string using the functions, RvSipBodyGetBodyStrLength() and RvSipBodyGetBodyStr(). You can set the Body string using the function, RvSipBodySetBodyStr().

10

AUTHENTICATION

INTRODUCTION

The authentication mechanism enables a User Agent Client (UAC) to prove authenticity to servers or proxies which require authentication. The SIP Stack supports SIP authentication using the HTTP Digest Scheme as described in RFC 3261 and RFC 2617. The SIP Stack authenticator object (*authenticator*) is responsible for applying the authentication mechanism in both the client and server authentication process. Client authentication is done above the User Agent layer; server authentication is done above the User Agent layer or the Transaction layer.

SHARED SECRET

A basic concept of the authentication mechanism is the shared secret between the User Agent Client (UAC) and the server or proxy. Prior to establishing SIP communication, the UAC should obtain a user name and password that the server acknowledges. The password is a shared secret between the UAC and the server.

DIGEST AUTHENTICATION WITH MD5

The Digest Authentication method implemented by the SIP Stack uses the MD5 algorithm in the authentication process. MD5 is a one-way hash function that operates on a given string and produces a fixed-length hash value.

AUTHENTICATION PROCESS

A SIP server verifies UAC authenticity using the HTTP Digest Scheme as follows:

1. A server responds to the originator of an incoming request with a 401 Unauthorized response. A Proxy server responds with a 407 Proxy Authentication Required response.

   These responses each include a special Authentication header with information required by the UAS in the Authentication process. The information included in the Authenticator header is called “challenge”.
2. The UAC uses the Authentication header parameters along with the user name and password to generate a hash value using the MD5 algorithm.
3. The hash value and other parameters are inserted into a special Authorization header.
4. The client re-sends the request with the Authorization header. The information included in the Authorization header is called “credentials”.
5. The server uses the credentials to verify the authenticity of the originator of the request.

AUTHENTICATOR OBJECT

The *authenticator* is responsible for applying the authentication mechanism by building the Authorization header on the client side, and by verifying the incoming Authorization header on the server side.

Call-legs, *register-clients* and *subscriptions* use the *authenticator* in order to authenticate outgoing requests. For example, a *call-leg* that sends an INVITE request and receives a 407 response assumes the UNAUTHENTICATED state. Your application can then call the RvSipCallLegAuthenticate() function which uses the *authenticator* and issues an INVITE with the Authorization header.

Call-leg, subscription and server *transactions* use the *authenticator* to authenticate incoming requests. When the object state indicates that a request was received, your application can trigger the *authenticator* to verify the authenticity of the originator.

Working with the Authenticator

To work with the authenticator, you should first get the *authenticator* handle from the SIP Stack. RvSipAuthenticatorHandle defines the handle to the authenticator. You can receive this handle by calling RvSipStackGetAuthenticatorHandle(). You use this handle to set your application callbacks, and to activate other *authenticator* functions. The *authenticator* is responsible for both client and server authentication.

Client Authentication Implementation

The *authenticator* performs the client authentication process independently but still requires your application to do the following:

- Supply the user name and password.
- Apply the MD5 hash function on a given string according to the requirements of the authenticator.
- Apply the MD5 hash result calculated on the message body for qop=auth-int authentication.

The application handles all of these tasks by implementing the callback functions that the *authenticator* will call whenever one of these tasks needs to be performed.

Client Authenticator Callbacks

The Authenticator API supplies several callback functions that your application must implement for the authentication process to succeed. You should implement these callbacks and set the callback pointers to the authenticator. The *authenticator* calls these callbacks during the authentication process as required. The following callbacks are supplied with the Authenticator API:

RvSipAuthenticatorMD5ExEv()

Notifies the application that use of the MD5 algorithm is required. This event supplies the application with the input string for the algorithm and the application returns the output string of the MD5 algorithm.

RvSipAuthenticatorGetSharedSecretEv()

Notifies the application that the shared user name and password are required. The event also supplies the *Realm* parameter that the UAS sent.

RvSipAuthenticatorMD5EntityBodyEv()

Notifies the application that it should supply the hash result on the message body (MD5(entity-body)). The body hash value is needed when the required quality of protection (qop) was set to “auth-int”.

Note This callback supplies the *message* as a parameter. However, it is called before the msgToSend() callback of any SIP Stack object. If your code adds the message body in the msgToSend callback, the body will not be available when this callback is called. If you wish the *message* to include the body in this callback, you must use the outbound message mechanism to add the body.

RvSipAuthenticatorUnsupportedChallengeEv()

Notifies the application about a challenge whose credentials cannot be prepared by the SIP Stack. An example of such a challenge is one with a non-MRD algorithm. When the *authenticator* encounters such a challenge, it gives the application a chance to build the credentials. The RvSipAuthenticatorUnsupportedChallengeEv() is called and the application should calculate the credentials by itself, build the Authorization header in the message—using RvSipAuthorizationHeaderConstructInMsg()—and set the calculated credential into it.

RvSipAuthenticatorNonceCountUsageEv()

Notifies the application about the value of the nonceCount parameter that the SIP Stack is going to use when calculating credentials. The application can change this value to fit more precise management of the nonceCount. The SIP Stack does not check the uniqueness of the used NONCE for a specific realm when the same realm is used by different objects.

RvSipAuthenticatorAuthorizationReadyEv()

Notifies the application that an Authorization header was built, and is ready to be sent in the outgoing message. In this callback the application can set additional information in the header.

SAMPLE CODE

The following code demonstrates the implementation and registration of the client authentication callbacks.

IMPLEMENTING THE MD5 CALLBACK FUNCTION

The input string to the MD5 algorithm is placed on an rpool page. When calling this callback, the *authenticator* supplies the RPOOL_Ptr structure which holds the memory pool, page and offset of the input string.

To implement the MD5 callback function

1. Allocate a consecutive buffer.
2. Use RPOOL_CopyToExternal() to copy the string to the buffer.
3. Give this buffer as an input to the MD5 algorithm.
4. Use RPOOL_AppendFromExternalToPage to insert the MD5 output into the page that is supplied in the pRpoolMD5Output parameter.

Sample Code

The following code demonstrates how to implement the MD5 callback function:

```
/*===========================================================================*/
void RVCALLCONV AuthenticationMD5Ev(
                    IN RvSipAuthenticatorHandle      hAuthenticator,
                    IN RvSipAppAuthenticatorHandle   hAppAuthenticator,
                    IN RPOOL_Ptr                     *pRpoolMD5Input,
                    IN RvUint32                      length,
                    OUT RPOOL_Ptr                    *pRpoolMD5Output)
{
    RvChar       *strInput;
    RvChar       strResponse[33];
    RvUint8      digest[20];
    MD5_CTX      mdc;

   /*Allocates the consecutive buffer.*/
    strInput = (RvChar*)malloc(length);

    /*Gets the string out of the page.*/
    RPOOL_CopyToExternal(pRpoolMD5Input->hPool,
                         pRpoolMD5Input->hPage,
                         pRpoolMD5Input->offset,
                         (void*) strInput,
                         length);
```

```
    /*Implements the MD5 algorithm.*/
    MD5Init(&mdc);
    MD5Update(&mdc, strInput,strlen(strInput));
    MD5Final(digest, &mdc);
    /*Changes the digest into a string format.*/
    MD5toString(digest, strResponse);
     /*Inserts the MD5 output to the page that is supplied in pRpoolMD5Output.*/
    RPOOL_AppendFromExternalToPage(pRpoolMD5Output->hPool,
                                   pRpoolMD5Output->hPage,
                                  (void*)strResponse,
                                  strlen(strResponse) + 1,
                                  &(pRpoolMD5Output->offset));
}
/*-------------------------------------------------------------------------*/
```

IMPLEMENTING THE SHARED SECRET CALLBACK FUNCTION

The following code demonstrates how to set the user name and password defined by the Realm parameter in the *authenticator* using RPOOL_AppendFromExternalToPage(). You can also use pRpoolRealm to search for the relevant shared secret which includes the user name and password.

Sample Code

The following code shows you how to implement the Shared Secret *authenticator* callback function:

```
/*-------------------------------------------------------------------------*/
RvChar       * strAuthUserName = "bob";
RvChar       * strAuthPassword   =  "bsg23hp";

void RVCALLCONV AuthenticationSharedSecretEv(
            IN RvSipAuthenticatrHandle       hAuthenticator,
            IN RvSipAppAuthenticatorHandle   hAppAuthenticator,
            IN void*                         hObject,
            IN void                          peObjectType,
            IN RPOOL_Ptr                     *pRpoolRealm,
            OUT RPOOL_Ptr                    *pRpoolUserName,
            OUT RPOOL_Ptr                    *pRpoolPassword)
{
      /*Appends the username to the given page.*/
      RPOOL_AppendFromExternalToPage(
```

```
                        pRpoolUserName->hPool,
                        pRpoolUserName->hPage,
                        (void*) strAuthUserName,
                        strlen(strAuthUserName) + 1,
                        &(pRpoolUserName->offset));

                /*Appends the password to the given page.*/
                RPOOL_AppendFromExternalToPage(
                        pRpoolPassword->hPool,
                        pRpoolPassword->hPage,
                        (void*) strAuthPassword,
                        strlen(strAuthPassword) + 1,
                        &(pRpoolPassword->offset));
        }

        /*-------------------------------------------------------------------------*/
```

SETTING APPLICATION CALLBACKS

Set your callback functions pointers in the *authenticator* using RvSipAuthenticatorSetEvHandlers().

The *authenticator* callback functions are gathered together in a structure called RvSipAuthenticatorEvHandlers. This structure is where you should set your callback function pointers and is given as a parameter to RvSipAuthenticatorSetEvHandlers().

Sample Code

The following code demonstrates how to set callback functions.

```
/*-----------------------------------------------------------------------------------------*/
static void SetEventHandlers()
{
  RvStatus                     status;
  RvSipAuthenticatorEvHandlers authEvHandlers;

  memset(&authEvHandlers,0,sizeof(RvSipAuthenticatorEvHandlers));

  /*Sets application callbacks in the structure.*/
  authEvHandlers.pfnMD5AuthenticationExHandler = AppAuthenticationMD5Ev;
  authEvHandlers.pfnGetSharedSecretAuthenticationHandler = AppAuthenticationGetSharedSecretEv;
  status = RvSipAuthenticatorSetEvHandlers(
                      g_hAuthenticatorMgr,
```

```
                        &authEvHandlers,
                        sizeof(RvSipAuthenticatorEvHandlers));
  return status;
}
/*===========================================================================================*/
```

AUTHENTICATION OBJECT CONTROL

Each WWW-Authenticate/Proxy-Authenticate header that is received in a 401/407 response message is saved in the relevant SIP Stack object (such as reg-client and *call-leg*) for the entire lifetime of the object. The SIP Stack will insert an equivalent Authorization header into every request that this SIP Stack object sends. An authentication object holds the information of the received Authentication header. The authentication object is represented by the RvSipAuthObjHandle handle.

The application can go over the list of authentication objects within a SIP Stack object (such as *call-leg*, *register-client*) and manipulate the authentication object list with the following API functions:

- RvSipXXXAuthObjGet()—gets all authentication objects from the relevant SIP Stack object one after another (such as RvSipRegClientAuthObjGet()).
- RvSipXXXAuthObjRemove()—removes an authentication object from the *authentication-object* list (such as RvSipRegClientAuthObjRemove()).
- RvSipXXXGetCurrProcessedAuthObj()—gets the currently processed authentication object in the context of the RvSipAuthenticatorGetSharedSecretEv() callback. The application can then get and set additional information from/to the received authentication object.

The application can query the authentication object parameters using the following APIs:

- RvSipAuthObjGetAuthenticationHeader()—gets the Authentication header from the *authentication-object*.
- RvSipAuthObjSetUserInfo()—sets the user information (user name and password) in the *authentication-object* before resending a request. In such cases, the RvSipAuthenticatorGetSharedSecretEv() callback will not be called for this object, since the SIP Stack already knows the required information.
- RvSipAuthObjSetAppContext()—sets an application context pointer in the *authentication-object*, and gets it afterwards.

Sample Code

The following code sample demonstrates how to get the authentication objects from a *call-leg*, and how to set the shared secret in each object.

```
/*=========================================================================*/
#define USERNAME "John"
#define PASSWORD "1234"
void AppSetSharedSecretInAllAuthObjects(
                              RvSipAuthenticatorHandle hAuthenticator,
                              RvSipCallLegHandle       hCallLeg)
{
    RvStatus rv;
    RvSipAuthObjHandle hAuthObj = NULL;

    /* Gets the first auth-obj from the call-leg. */
    rv = RvSipCallLegAuthObjGet(hCallLeg, RVSIP_FIRST_ELEMENT,
                               NULL, &hAuthObj);
    if(rv != RV_OK || hAuthObj == NULL)
    {
        printf("\nClient - no auth-obj in call-leg\n");
        return;
    }

    while (rv == RV_OK && hAuthObj != NULL)
    {
        /* Sets the shared secret in the authentication object. */
        rv = RvSipAuthObjSetUserInfo(hAuthenticator, hAuthObj,
                                     USERNAME, PASSWORD);
        /* Gets the next auth-obj from the call-leg. */
        RvSipCallLegAuthObjGet(hCallLeg, RVSIP_NEXT_ELEMENT,
                               hAuthObj, &hAuthObj);
    }
}
/*=========================================================================*/
```

Authenticating a Message in Advance

The *authenticator* uses the Authentication header from the 407 response to authenticate an outgoing request. The *authenticator* provides several API functions that allow you to authenticate requests in advance. To authenticate requests in advance, you should supply the *authenticator* with an Authentication header with which to work.

You can construct an Authentication header and set the fields using the Message API. You should then set the header to the *authenticator*, using RvSipAuthenticatorSetProxyAuthInfo(). The *authenticator* uses the header for building the Proxy-Authorization header that is inserted into the initial request.

The following API functions supply in-advance authentication control:

RvSipAuthenticatorSetProxyAuthInfo()

Sets a Proxy-Authorization header to the *authenticator*. This header will be used to authenticate requests in advance. You can use this function in cases where all outgoing requests are passed through the same Proxy. For example, when working with an outbound Proxy. In these cases, the UAC might know all the information needed for the authentication process in advance and may wish to send initial requests with authorization information. The Proxy can then authenticate the UAC immediately without sending a 407 response.

The *authenticator* uses the header for building the Proxy-Authorization header that is inserted into the initial request.

RvSipAuthenticatorSetNonceCount()

Sets the initial value of the nonce-count parameter that will be used when creating the Proxy-Authorization header that is placed in outgoing requests. This nonce count is used with the challenge supplied by calling the RvSipAuthenticatorSetProxyAuthInfo() function for authenticating *messages* in advance. The nonce-count value is incremented by the *authenticator* after each header calculation, according to RFC 2617.

RvSipAuthenticatorAddProxyAuthorizationToMsg()

Adds a Proxy-Authorization header to the supplied *message*. You can use this function only if you set a Proxy-Authenticate header to the *authenticator* using the RvSipAuthenticatorSetProxyAuthInfo() function. The *authenticator* uses the challenge found in the Proxy-Authenticate header to build the correct Proxy-Authorization() header. You should use this function if you want to add credentials to outgoing requests that were sent by stand-alone *transactions*. This function should be called from the RvSipTransactionMsgToSendEv() callback of the transaction. For other SIP Stack objects, the process of adding the authorization header is automatic.

Sample Code

The following code demonstrates how to set the Proxy-Authenticate header for authenticating *messages* in advance.

```
/*=========================================================================*/
RvStatus SetProxyAuthenticateHeader (
       RvSipAuthenticatorHandle     hAuthenticator,
       HRPOOL          hPool,
       HPAGE           hPage)
{
        RvStatus status;
      RvSipAuthenticationHeaderHandle hProxyAuth;
        /*-----------------------------------------
         Constructs the Proxy-Authenticate header.
        ----------------------------------------*/
        status = RvSipAuthenticationHeaderConstruct(hPool,
                        hPage, &hProxyAuth);
        if (status != RV_OK)
        {
            return status;
        }
        /*-----------------------------------------------------
         Sets parameters to the header. (You can set more parameters).
        -----------------------------------------------------*/
        /*Sets the nonce.*/
        RvSipAuthenticationHeaderSetNonce(hProxyAuth,
               "\"1234abab\"");
        /*Sets the realm.*/
        RvSipAuthenticationHeaderSetRealm(hProxyAuth,
               "\"hp.com\"");
        /*Sets the opaque.*/
        RvSipAuthenticationHeaderSetOpaque(hProxyAuth,
                "\"abcdef1234\"");
                /*Set authentication scheme.*/
        RvSipAuthenticationHeaderSetAuthScheme
                              (hProxyAuth,
                                 RVSIP_AUTH_SCHEME_DIGEST,NULL);

        /* Sets the algorithm.*/
        RvSipAuthenticationHeaderSetAuthAlgorithm
                              (hProxyAuth,
```

```
                                    RVSIP_AUTH_ALGORITHM_MD5,NULL);
        /*Sets the QOP options.*/
          RvSipAuthenticationHeaderSetQopOption(hProxyAuth,
                    RVSIP_AUTH_QOP_AUTH_ONLY);
          /*-------------------------------------
          Sets the Proxy-Authenticate header.
          -------------------------------------*/
          status =
              RvSipAuthenticatorSetProxyAuthInfo(hAuthenticator,
                                          hProxyAuth,
                                          "bob",
                                          "bsg23hp");
          return status;
}
/*=========================================================================*/
```

CLIENT AUTHENTICATION FOR STAND-ALONE TRANSACTION

A stand-alone client *transaction* can receive a 401 or 407 response. The response will include a challenge, and the application may wish to send a new request with the required credentials. To do this, the application needs to create a new transaction. The application should then use the RvSipAuthenticatorPrepareAuthorizationHeader() function to add the credentials to the new outgoing request.

The application should supply this function with the outbound message together with the Authentication header taken from the 401 or 407 response. The function will add the required Authorization header to the message.

Note It is the responsibility of the application to add to the new request all the applicative information that was added to the original request.

CLIENT AUTHENTICATION WITH MULTIPLE PROXIES

In many cases, a request message sent by a client has to traverse several proxies that require authentication. Each of these proxies will require different credentials in order to forward the message to the next hop. Each of the SIP Stack objects (*register-clients* and *call-legs*) maintains a list of all the challenges returned from the different proxies in the message path. When calling the RvSipXXXAuthenticat() API function, suitable credentials will be added for each of these challenges.

SERVER AUTHENTICATION IMPLEMENTATION

Call-leg, subscription and server *transactions* can authenticate incoming requests, and respond with 401/407 when the authentication process fails. The originator credentials are located in an Authorization header. An incoming request message may contain more than one Authorization header. The server authentication process is actually a loop that searches for Authorization header in the received request message, tries to verify the credentials of the header, and if verification fails, searches for the next Authorization header. The server authentication procedure is completed if there are no more Authorization headers in the message, or if one Authorization header with verified credentials was found.

APPLYING THE SERVER AUTHENTICATION MECHANISM

In order to apply the server authentication mechanism, you must follow the steps below.

To apply the server authentication mechanism

1. Set the *enableServerAuth* configuration parameter to RV_TRUE.
2. Implement the MD5 callback and set it to the *authenticator* using the Authenticator API. (For more information, see the above section, Client Authentication Implementation.)
3. Use the Call-leg, Subscription and Server Transaction API functions and callback functions to trigger and advance the authentication process. These functions scan and verify the Authorization headers.

Note The application is responsible for the progress of the loop, and may do it in an asynchronic or synchronic way.

When the SIP Stack receives a new request message, the server assumes a suitable state that indicates that the request was received. For example the OFFERING state of a *call-leg* indicates that an INVITE request was received. At this point, if the application wishes to verify the authenticity of the originator, it may begin the server authentication procedure.

Note The server authentication procedure can be implemented above the Call-leg, Subscription or Transaction layers. In the following description, the RvSipXXX prefix is added to the API and callback functions. This prefix can be replaced by RvSipCallLeg, RvSipSubs or RvSipTransaction.

SERVER AUTHENTICATION PROCESS

The server authentication procedure steps are as follows:

1. To start the authentication process after a request was received, the application should call the function, RvSipXXXAuthBegin().
2. This triggers the SIP Stack to locate the first Authorization header in the incoming request message.
3. If the SIP Stack fails to find an Authorization header, it continues to step 7. Otherwise, the SIP Stack calls the RvSipXXXAuthCredentialsFoundEv() callback and supplies the application with the retrieved header. This header includes the sender userName, realm, and other credentials parameters. The SIP Stack also indicates whether it is capable of verifying the credentials.[1]
4. The application should look for the user in its database and retrieve the user password, if such a user exists in the database.
5. The application should then instruct the SIP Stack on how to proceed the authentication process. The application can choose one of the following options:
   - If the user was found in the database, the application may call the RvSipXXXAuthProceed(USE_PW) function, giving the user password. This instructs the SIP Stack to try to authenticate the user with the supplied password.

1. If, for example, the SIP Stack does not support the credentials algorithm, it will indicate that it is incapable of verifying the credentials.

- If the user was not found in the database, the application may call the RvSipXXXAuthProceed(SKIP_HEADER) function in order to continue with the loop, finding the next Authorization header (back to step 3).
- If the SIP Stack does not support the credentials, the application can try to verify the credentials by itself. By calling the RvSipXXXAuthProceed(SUCCESS) function, the application indicates that the credentials were verified by it. The SIP Stack will stop the loop, and RvSipXXXAuthCompletedEv() event will be called, with Success indication. (If it was not verified, the application may use the SKIP_HEADER option).
- If the application wishes to stop the authentication process, it can call the RvSipXXXAuthProceed(Failure) function. This will stop the loop and RvSipXXXAuthCompletedEv() event will be called, with Failure indication.

6. Calling the RvSipXXXAuthProceed (USE_PW) function will instruct the SIP Stack to verify the authenticity of the originator, using the given password.
   - If authentication succeeds, the server authentication procedure is completed. The SIP Stack will call the RvSipXXXAuthCompletedEv() callback with Success indication.
   - If authentication fails, the SIP Stack will continue the loop, searching for the next Authorization header (back to step 3).
7. If there are no more Authorization headers, RvSipXXXAuthCompletedEv will be called, with Failure indication
8. When the RvSipCallLegAuthCompletedEv is called, the application should decide how to respond to the request. If the completed status is failure, the application should respond with RvSipXXXRespondUnauthenticated. If the completed status is success, the application should respond as usual, according to the request type, and object state.

Figure 10-1 describes the server authentication process flow:

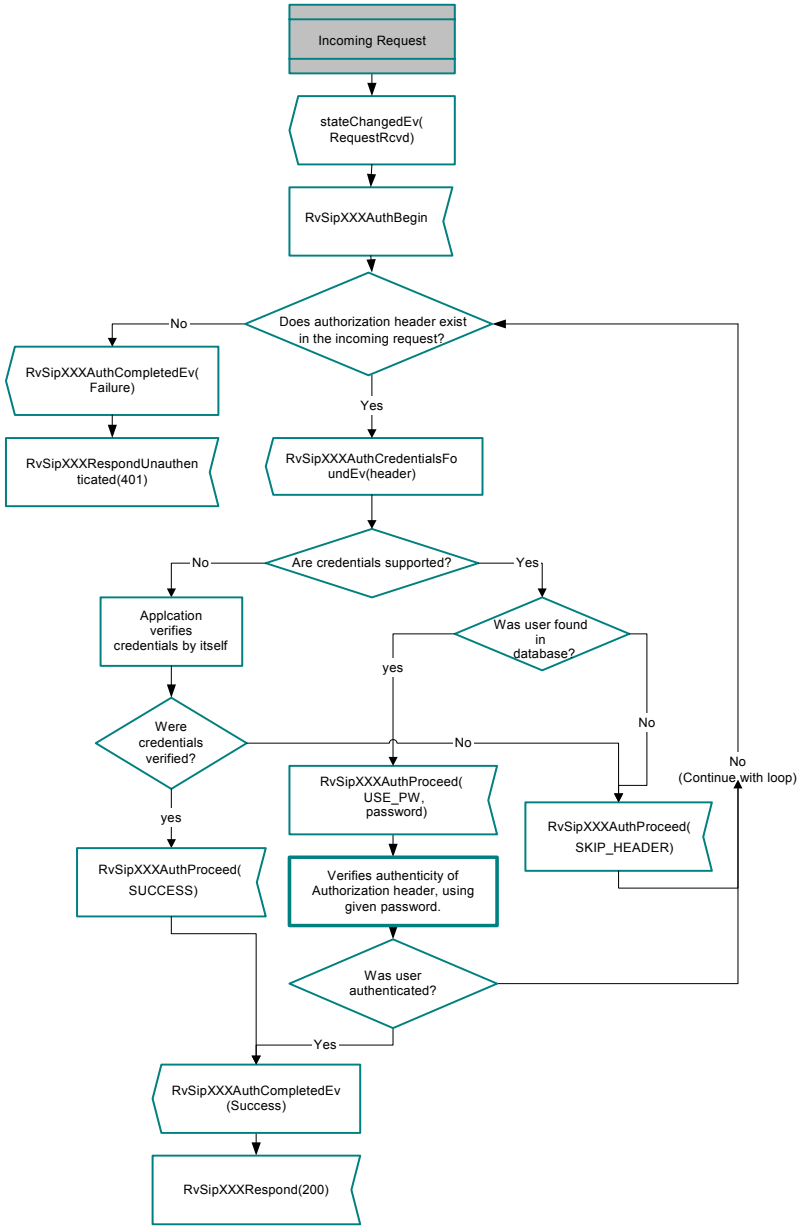


Figure 10-1 *Server Authentication Process Flow*

Server Authentication Functions and Callbacks

Server authentication may be implemented above the Call-leg, Subscription or Transaction layers (for proxies). Because of this, there are three sets of functions, one for each layer.

Call-leg Server Authentication Functions

The following functions are used for Call-leg server authentication:

RvSipCallLegAuthBegin()

Begins the server authentication process. This function may be called in any of the following states which indicate that a new request was received:

- RVSIP_CALL_LEG_STATE_OFFERING—authenticates an INVITE request
- RVSIP_CALL_LEG_MODIFY_STATE_REINVITE_RCVD —authenticates re-INVITE request
- RVSIP_CALL_LEG_REFER_STATE_REFER_RCVD— authenticates a REFER request
- Inside the RvSipCallLegTrancRequestRcvdEv callback function—to authenticate a general request

RvSipCallLegAuthProceed()

Proceeds with the server authentication process. This function should be called after the RvSipCallLegCredentialsFoundEv callback was called (see below). In this function you may choose to proceed with one of four actions:

- Instruct the SIP Stack to authenticate the credentials, suppling it with a suitable password.
- Skip the given credentials and instruct the SIP Stack to search for the next credentials. (Use this option if you are not familiar with the realm or username found in the given credentials.)
- Finish the authentication process with Success indication.
- Finish the authentication process with Failure indication.

RvSipCallLegRespondUnauthenticated()

Responds with a 401 or 407 message. This function gets an already-built header (Authentication header or Other header), and sets it in the response message.

RvSipCallLegRespondUnauthenticatedDigest()

Responds with a 401 or 407 message. You supply this function with the challenge parameters. The SIP Stack will use these parameters to create an Authentication header. This header will be placed in the response message.

.Call-leg Server Authentication Callbacks

RvSipCallLegAuthCredentialsFoundEv()

This callback notifies the application that credentials were found in the request message. The callback supplies the application with the Authorization header that contains these credentials. At this stage the application should use the RvSipCallLegAuthProceed() function

This callback also indicates whether the SIP Stack is capable of verifying the credentials that were found.Whenever the SIP Stack does not support the credentials (for example, if the algorithm is not MD5) the application may verify the credentials by itself or instructing the SIP Stack to continue to the next header.

RvSipCallLegAuthCompletedEv()

Notifies the application that the server authentication process was completed, and indicates whether or not the sender was authenticated. According to the authentication results, the application should decide whether to accept the request or reject it using the RvSipCallLegRespondUnauthenticatedDigest() or RvSipCallLegRespondUnauthenticated() functions.

Transaction Server Authentication Functions

The server authentication functions of the Transaction layer are similar to the functions of the Call-leg layer.

RvSipTransactionAuthBegin()

Can be used in the following states:

- RVSIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD
- RVSIP_TRANSC_STATE_SERVER_INVITE_REQUEST_ RCVD

RvSipTransactionAuthProceed()

RvSipTransactionRespondUnauthenticated()

RvSipTransactionRespondUnauthenticatedDigest()

Transaction Server Authentication Callbacks

The server authentication callback functions of the Transaction layer are similar to the callback functions of the Call-leg layer. The following functions are supplied:

RvSipTransactionAuthCredentialsFoundEv()

RvSipTransactionAuthCompletedEv()

Subscription Server Authentication

Server authentication in the Subscription layer is identical to the server authentication in the Call-leg and Transaction layers. For more information, see the Event Notification chapters in the *SIP Stack Reference Guide.*

Figure 10-2 shows the event flow of a successful server authentication procedure.

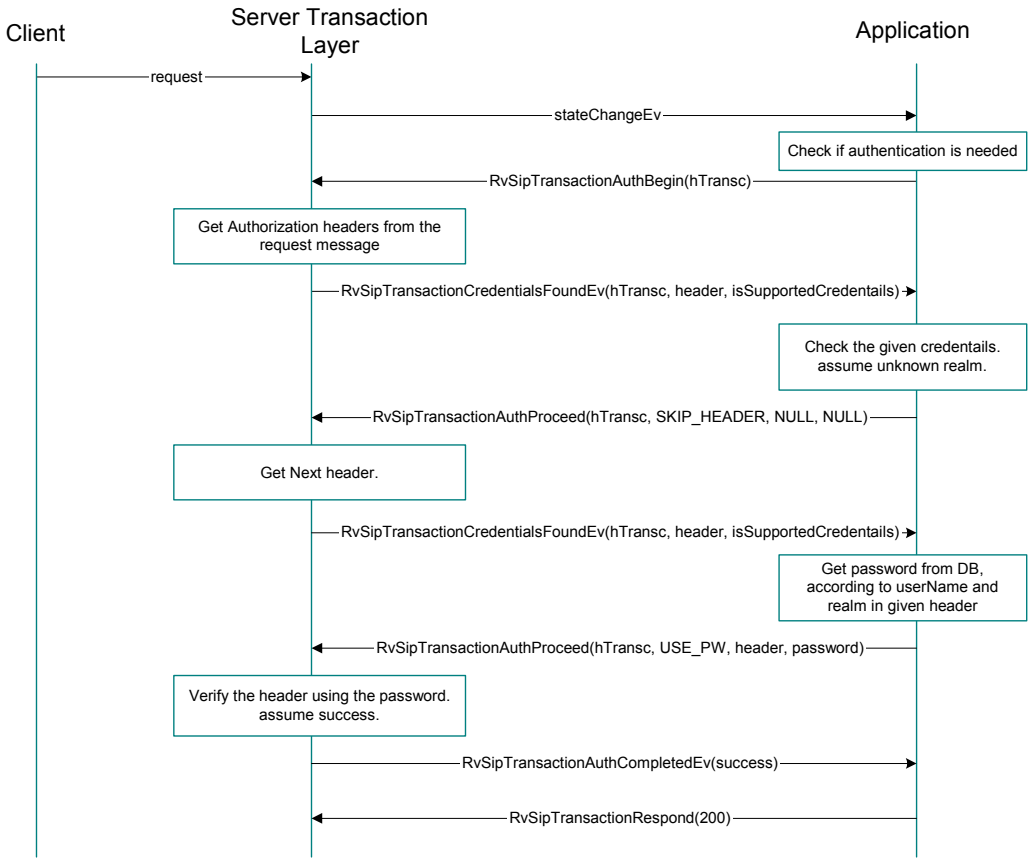


Figure 10-2 *Server Authentication Event Flow*

Sample Code

The following code demonstrates the implementation of the RvSipCallLegAuthCredentialsFoundEv callback function.

```
/*=========================================================================================*/
RvChar* g_strSrvAuthRealm = "HP Realm";

void AppAuthCredentialsFoundEv(
        IN RvSipCallLegHandle             hCallLeg,
        IN RvSipAppCallLegHandle          hAppCallLeg,
        IN RvSipTranscHandle              hTransc,
        IN RvSipAuthorizationHeaderHandle hAuthorization,
        IN RvBool                         bCredentialsSupported)
{
    RvStatus rv;
    RvChar    strName[50], strRealm[50], strNonce[50];
    RvChar    strPW[50];
    RvInt32   actualLen;
    RvBool    toSkip = RV_FALSE;

    if(bCredentialsSupported == RV_FALSE)
    {
        printf("Unknown Algorithm or scheme. skip header.");
        toSkip = RV_TRUE;
    }
    else
    {
      /*-------------------------------------------------------------------------------------
      Gets the realm, username and nonce parameters from the given credentials and checks if
      they fit.
      -------------------------------------------------------------------------------------*/
        rv = RvSipAuthorizationHeaderGetCredentialIdentifier(
                                    hAuthorization,
                                    strRealm, 50,
                                    strName, 50,
                                    strNonce, 50,
                                    &actualLen,
                                    &actualLen,
                                    &actualLen);
        if(rv != RV_OK)
        {
            printf("Couldn't get credentials Identifiers from authorization header. skip
                                                                                header.");
            toSkip = RV_TRUE;
        }
```

```
    /*-------------------------------------------------------------------------------
    g_strSrvAuthNonce and g_strSrvAuthRealm are global parameters, holding the
    application's nonce and realm values.
     -------------------------------------------------------------------------------*/
    if(strcmp(strNonce, g_strSrvAuthNonce)!=0)
    {
        printf("incorrect nonce %s. call %x, continue with loop\n", strNonce, hCallLeg);
        toSkip = RV_TRUE;
    }
    if(strcmp(strRealm, g_strSrvAuthRealm)!=0)
    {
        printf("unknown realm %s. call %x, continue with loop\n", strRealm, hCallLeg);
        toSkip = RV_TRUE;
    }

    /*-------------------------------------------------------------------------------
       Searches the application's users database to find if this user exists. If the user
       exists, takes the password.
       -------------------------------------------------------------------------------*/
    else if(GetUserFromSrvAuthDB(strName, strPW) != RV_OK)
    {
        printf("unknown username %s. call %x, continue with loop", strName, hCallLeg);
        toSkip = RV_TRUE;
    }
}

/*-------------------------------------------------------------------------------
  Proceeds with the loop using the "toSkip" decision that was made.
   -------------------------------------------------------------------------------*/
if(toSkip == RV_FALSE) /*Finds the correct user and password*/
{
    rv = RvSipCallLegAuthProceed(
                    hCallLeg,
                    hTransc,
                   RVSIP_TRANSC_AUTH_ACTION_USE_PASSWORD,
                   hAuthorization,
                    strPW);
}
else
{
    rv = RvSipCallLegAuthProceed(
```

```
                                hCallLeg,
                                hTransc,
                                RVSIP_TRANSC_AUTH_ACTION_SKIP,
                                NULL,
                                NULL);
    }
    if(rv != RV_OK)
    {
        printf("RvSipCallLegAuthProceed fail. Status is %d.", rv);
    }
}
/*===========================================================================================*/
```

Sample Code

The following code demonstrates the implementation of the RvSipCallLegAuthCompletedEv callback function.

```
/*===========================================================================================*/
void RVCALLCONV AppAuthCompletedEv(
               IN    RvSipCallLegHandle       hCallLeg,
               IN    RvSipAppCallLegHandle    hAppCallLeg,
               IN    RvSipTranscHandle        hTransc,
               IN    RvBool                   bAuthSucceed)
{
    RvStatus rv;
    RvSipCallLegState eState;

    if(bAuthSucceed == RV_FALSE)
    {
        /*-------------------------------------------------------------------------------------
        The authentication procedure was completed with failure. Sends a 401 Unauthorized
        response.
        -------------------------------------------------------------------------------------*/
        rv = RvSipCallLegRespondUnauthenticatedDigest(
                      hCallLeg, NULL, 401, NULL,
                      g_strSrvAuthRealm, NULL,
                      g_strSrvAuthNonce, NULL,
                      RV_TRUE,
                      RVSIP_AUTH_ALGORITHM_UNDEFINED, NULL,
                      RVSIP_AUTH_QOP_UNDEFINED, NULL);
        if(rv != RV_OK)
        {
```

```
            printf("Error in RespondUnauthenticatedDigest");
        }
    }
    else
    {
        /*-----------------------------------------------------------------------------
          The authentication procedure was completed with success. Accepts the Invite request.
          -----------------------------------------------------------------------------*/
        if(hTransc != NULL)
        {
            rv = RvSipCallLegTranscResponse(hCallLeg, hTransc, 200);
        }
        else
        {
            rv = RvSipCallLegAccept(hCallLeg);
        }
        if(rv != RV_OK)
        {
            printf("Error in accepting the request");
        }
    }
}
/*=============================================================================*/
```

AUTHENTICATOR FUNCTIONS FOR SERVER USAGE

Server applications, such as stateless proxy, may want to use the *authenticator* functions directly to verify credentials, and not to use them through the Transaction functions. Such applications should use the following *authenticator* functions:

RvSipAuthenticatorCredentialsSupported()

Checks whether or not the SIP Stack is capable of authenticating the given Authenticator header. Supporting requirements are MD5 algorithm, Digest scheme, and qop = auth. Another requirement is that the RvSipAuthenticatorMD5Ev callback function is implemented.

RvSipAuthenticatorVerifyCredentialsExt()

Gets an Authorization header and password, and generates a hash value using the MD5 algorithm. Checks if the generated value is equal to the value in the Authorization header. If it is equal, the credentials are verified.

Sample Code

The following code demonstrates the usage of the *authenticator* functions for server authentication.

```
/*=========================================================================================*/
RvStatus AuthenticateCredentials(
        IN      Transaction*                    pTransc,
        IN      RvSipAuthorizationHeaderHandle hAuthorization,
        IN      RvChar                          *password,
        OUT     RvBool*                        isCorrect)
{
    RvStatus  stat;
    RvBool    bIsSupported;

    /*-----------------------------------------------------------------------------------------
      Validity Checking. Will the authenticator be able to verify this Authorization header?
      -----------------------------------------------------------------------------------------*/
    stat = RvSipAuthenticatorCredentialsSupported(
                          pTransc->pMgr->hAuthenticator,
                          hAuthorization,
                          &bIsSupported);
    if(bIsSupported != RV_TRUE)
    {
        printf("given credentials are invalid. cannot authenticate it\n");
        return RV_ERROR_UNKNOWN;
    }

    /*-----------------------------------------------------------------------------------------
     Authorization header verification. The authenticator will use the given header and
     password.
     -----------------------------------------------------------------------------------------*/
    stat = RvSipAuthenticatorVerifyCredentials(
                            pTransc->pMgr->hAuthenticator,
                            hAuthorization,
                            password,
                            makeStrMethod(pTransc),
                            isCorrect);
    if(*isCorrect == RV_TRUE)
    {
        printf("credentials were verified successfully! user authenticity was proved");
    }
```

```
    else
    {
        printf("credentials were not verified! n");
    }
    return stat;
}
/*=============================================================================================*/
```

11

WORKING WITH TRANSMITTERS

INTRODUCTION

The SIP Stack uses transmitter objects (*transmitters*) for message sending. Each *transaction* holds a single *transmitter* and uses it to send SIP *messages* and message retransmissions. The *transaction* creates its *transmitter* upon initialization and terminates it only upon destruction. The *transmitter* is responsible for all message sending activities, including address resolution, Via header handling, and the actual message sending.

The application can also directly use *transmitters*. The application can create *transmitters* and use them to send SIP *messages* that are not related to *transactions*. Each *transmitter* can send a single SIP *message* (Request or Response). Applications will need to use *transmitters* in several situations. For example, a proxy application should use *transmitters* to proxy responses that were not mapped to any *transaction* or the ACK on the 2xx response to INVITE. A *transmitter* can also be used to send non-SIP *messages*. The application can use the *transmitter* to send any buffer it wishes to its chosen destination.

This chapter focuses on how the application uses *transmitters* to send SIP *messages*. The section, Sending Buffers with Transmitter Objects, explains how to use the *transmitter* to send a non-SIP message.

TRANSMITTER ENTITIES

The Transmitter API relates to the following two entities:

- Transmitter (*transmitter*)
- Transmitter Manager (*TransmitterMgr*)

TRANSMITTER

A *transmitter* is capable of sending a single SIP message, request or response. The application should supply the *message* to the *transmitter* and the *transmitter* will be responsible for all message sending activities, including address resolution and DNS, *connection* handling, Via header handling and the actual message sending.

A *transmitter* is a stateful object that can assume any state from a set defined in the Transmitter API. The Transmitter state machine indicates the progress and result of the message sending process.

TRANSMITTER MANAGER

The *TransmitterMgr* manages the collection of *transmitters* and is mainly used for creating new *transmitters*.

WORKING WITH HANDLES

All *transmitters* and the *TransmitterMgr* are identified using handles. You must supply these handles when using the Transmitter API.

RvSipTransmitterMgrHandle defines the *TransmitterMgr* handle. You receive this handle by calling RvSipStackGetTransmitterMgrHandle().

RvSipTransmitterHandle defines a *transmitter* handle. You receive the Transmitter handle when creating a *transmitter* with RvSipTransmitterMgrCreateTransmitter().

TRANSMITTER API

The Transmitter API contains a set of functions and function callbacks that allow you to set or examine *transmitter* parameters and to control *transmitter* functionality.

TRANSMITTER PARAMETERS

You can set or examine *transmitter* parameters via *transmitter* Set and Get API functions. The following parameters are available:

Local Address and Current Local Address

The Local Address defines the address from where the message will be sent (the network card). This is also the address that will be placed in the top Via header of a Request message. If the local address is not set, the *transmitter* will use a default local address according to the SIP Stack configuration. A local address can be configured for each address and transport type. The Current Local Address is the address that is actually used for sending the message.

Outbound Address

An address of an outbound proxy that the *transmitter* should use. The outbound address is used only if the *transmitter* is sending a Request message. In this case, the *transmitter* will use the outbound address as a remote address and the Request-URI will be ignored. The outbound address of the *transmitter* is ignored if the request contains a Route header, or if the RvSipTransmitterSetIgnoreOutboundProxyFlag() function was called.

Destination Address

The destination address is the final address (IP, port and transport) that the *transmitter* will use for sending the message. The destination address is calculated in the address resolution process, which takes into account the existence of outbound address, Route headers and other *transmitter* parameters. The application can get the destination address only in the FINAL_DEST_RESOLVED state of the *transmitter* that indicates that the address resolution process has completed. The application can set the destination address and, in this case, this address will be used and the *transmitter* will not perform address resolution.

Persistency Definition and Used Connection

When working with TCP or TLS, the application can instruct the *transmitter* to try to locate a suitable open *connection* in the *connection* hash before constructing a new *connection*. The application can also supply the *transmitter* with a *connection* that the *transmitter* may use. For more information on persistency level and Persistent Connection API functions, see Persistent Connection Handling of the Working with the Transport Layer chapter.

Current State

Represents the state of the *transmitter* in the message sending process. You can only access the state parameter with a Get function and it is not modifiable.

DNS List

A *transmitter* always holds a DNS list object. The DNS list object holds the answers from the DNS server and is updated in the address resolution process. The application can manipulate the list using the DNS List API. For more information on the DNS List API see the Working with DNS chapter.

Via Branch

A branch parameter that will be added to the top Via header of an outgoing Request message. The application can set the branch parameter to the *transmitter* before sending a Request message. When the Request is sent, if the message already has a branch, it will be replaced with the branch set to the transmitter. If no branch was set to the *transmitter* and the top Via header of the message does not include a branch, the *transmitter* will generate a new branch and set it in the top Via header. This parameter has a Set function only.

Ignore Outbound Proxy Flag

Instructs the *transmitter* to ignore its outbound proxy when sending a request. When the message includes one or more Route headers, the *transmitter* will always ignore its outbound proxy. If there are no Route headers and an outbound proxy is configured, it will be used as the remote address of the message.

In some cases, the application will want the *transmitter* to ignore the outbound proxy even if it is configured to use one. An example is the case of strict routing when a single Route header becomes the message request URI, and therefore the *message* does not include any Route headers but the *transmitter* still needs to ignore its outbound proxy.

Use First Route Flag

Instructs the *transmitter* to use the first Route header as the remote address and not the Request-URI. The message should be sent to the first route header and not to the Request-URI when the message is sent to a loose route proxy.

Fix Via Flag

Indicates that the *transmitter* should update the “sent-by” parameter of the top Via header before sending the message. The “sent-by” parameter should be updated according to the local address from which the request will be sent. This address is determined according to the remote address transport and address types.

In many cases, the application does not know the remote party IP, transport and address types in advance, and therefore cannot know which local address will be used. In this case, the application might want the *transmitter* to update the top Via automatically and therefore should call the RvSipTransmitterSetFixViaFlag() function.

The default value of the Fix Via parameter is RV_FALSE and it should remain RV_FALSE if the application updates the top Via by itself.

Remark: Regardless of the value of this parameter, the *transmitter* will update the *transport* and *rport* parameters of the top Via header.

Keep Msg Flag

Indicates that the *transmitter* should not destruct the message immediately after encoding is completed. Before the *transmitter* sends a message, it first encodes the *message* to a buffer and then sends the buffer to the remote party. After encoding is completed, the *transmitter* destructs the *message*. In case of send failure, the *transmitter* moves to the MSG_SEND_FAILURE state. In this state, the application can re-send the *message* to the next address in the DNS list using RvSipTransmitterSendMessage(). The application can instruct the *transmitter* not to destruct the message after encoding by setting the Keep Msg Flag to RV_TRUE and, in this case, it can supply NULL as a message handle to the RvSipTransmitterSendMessage() function. The message will be destructed only upon termination of the transmitter.

Transmitter Control

The following API functions provide *transmitter* control:

RvSipTransmitterSendMessage()

Sends a message to the remote party. The application should supply the *message* that it wishes the *transmitter* to send. To send the message, the *transmitter* has to resolve the destination address. The *transmitter* first moves to the RESOLVING_ADDR state and starts the address resolution process. The *transmitter* calculates the remote address of the message according to RFC 3261 and RFC 3263 and takes into account the existence of outbound proxy, Route headers and loose routing rules.

Once address resolution is completed, the *transmitter* moves to the FINAL_DEST_RESOLVED state. This state is the last chance for the application to modify the Via header. The *transmitter* will then move to the READY_FOR_SENDING state and will try to send the message. If the message is sent successfully, the *transmitter* will move to the MSG_SENT state. If the *transmitter* fails to send the message, it will move to the MSG_SEND_FAILURE state.

The RvSipTransmitterSendMessage() function can be called in two states—the IDLE state for initial sending, and the MSG_SEND_FAILURE state for sending the message to the next address in the DNS list in case the previous address failed.

Remarks:

- The DNS procedure is a-synchronic, and therefore the send function may return with success before the message was actually sent.
- If you wish the *transmitter* to fix the top Via header of the *message* according to the remote party address and transport types, you should first call the RvSipTransmitterSetFixViaFlag() function. Otherwise the *transmitter* will only fix the via transport and will add the *rport* parameter in case it was configured to the SIP Stack. The sent-by parameter will remain untouched.
- The *transmitter* copies the message supplied by the application to the SIP Stack memory. The application is responsible for destructing the message it supplies to this function.

RvSipTransmitterHoldSending()

Holds all sending activities of the *transmitter* and moves the *transmitter* to the ON_HOLD state. After address resolution is completed and before the message is sent, the *transmitter* moves to the FINAL_DEST_RESOLVED state. In this state, the application can hold the message sending by calling RvSipTransmitterHoldSending(). If the application wishes, it can change the remote address at this point using the RvSipTransmitterSetDestAddress() function and manipulate the rest of the DNS list using the Transport layer API. If the application wishes the *transmitter* to use the next element in the list, it can use the RvSipTransmitterSetDestAddress() function to reset the current destination address. The *transmitter* will then repeat the address resolution process before sending the message. It is the responsibility of the application to resume the sending of the message using RvSipTransmitterResumeSending().

RvSipTransmitterResumeSending()

Resumes the sending activities of the transmitter. This function can be called only in the ON_HOLD state of the transmitter. When this function is called, the *transmitter* first checks that a destination address exists. If one exists, the *transmitter* moves to the READY_FOR_SENDING state and then sends the message to this address. If there is no a destination address (the user reset the address by calling RvSipTransmitterSetDestAddress() with NULL values), the *transmitter* returns to the RESOLVING_ADDR state and to the address resolution process.

RvSipTransmitterTerminate()

Terminates a *transmitter* and frees all transmitter-allocated resources. The *transmitter* will assume the TERMINATED state.

Remark: The application is responsible for terminating the *transmitter*. The SIP Stack will never terminate *transmitters* that were created by the application. In case of an a-synchronic failure, the *transmitter* will move to the MSG_SEND_FAILURE with a reason that indicates the nature of this failure. It is up to the application to terminate the *transmitter* at this point.

EVENTS

The Transmitter API supplies several events in the form of callback functions, to which your application may listen and react. To listen to an event, your application should pass the event handler pointer to the *transmitter* when it is created. When an event occurs, the *transmitter* calls the event handler function using the pointer.

The following events are supplied with the Transmitter API:

RvSipTransmitterStateChangedEv()

Notifies the application of a *transmitter* state change. Each phase of the message sending process of the *transmitter* is represented by a state and the application is notified through the RvSipTransmitterStateChangedEv() callback. For each state change, the new state is supplied with the reason for the new state, and the *message* when valid. An additional parameter, *pExtraInfo*, will hold specific state information.

Most of the states are informative only. The final message sending states (MSG_SENT and MSG_SEND_FAILURE) indicate that the application should now terminate the *transmitter*.

Remark: Currently the *pExtraInfo* parameter is used only in the NEW_CONN_IN_USE state.

RvSipTransmitterOtherURLAddressFoundEv()

Notifies the application that a message needs to be sent and the destination address is a URL type that is currently not supported by the SIP Stack (for example a tel URL). The application must convert the URL to a SIP URL for the message to be sent.

TRANSMITTER STATE MACHINE

The Transmitter state machine illustrated in Figure 11-1 represents the state of the *transmitter* in the message sending activity.

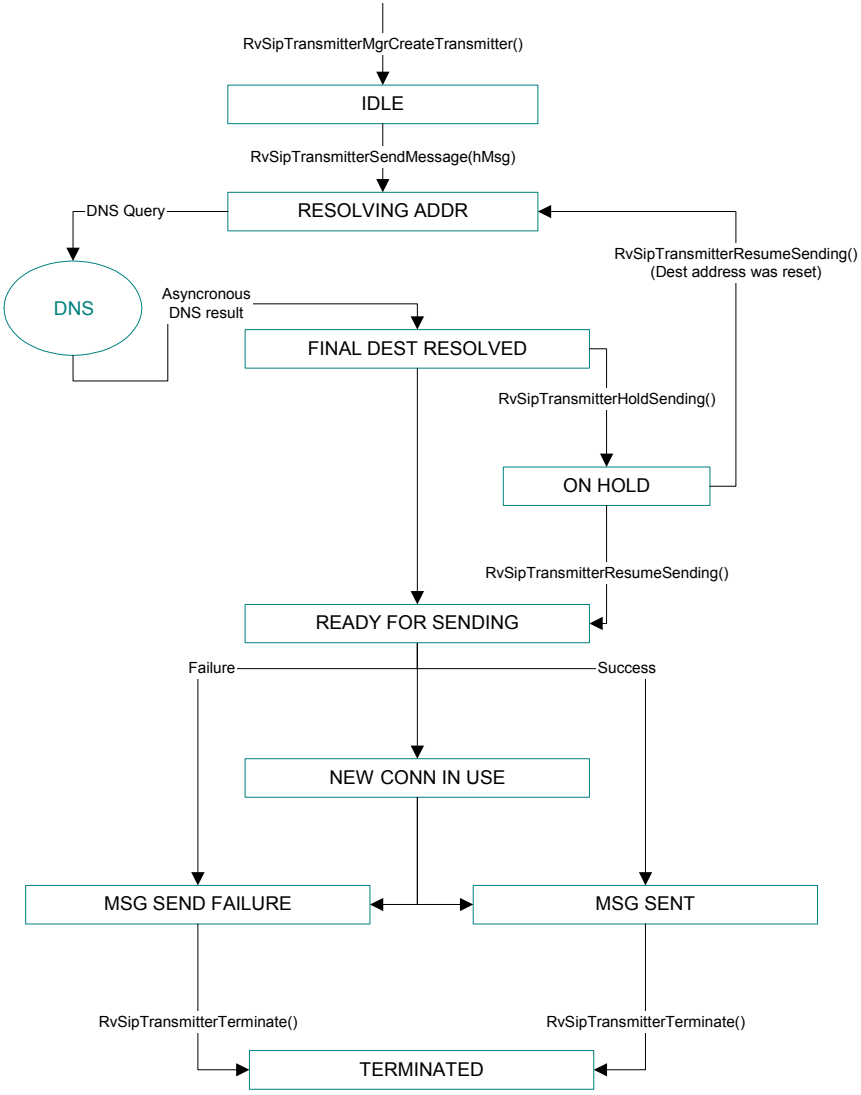


Figure 11-1 *Transmitter State Machine*

The RvSipTransmitterStateChangedEv() callback reports *transmitter* state changes and state change reasons. The state change reason indicates the reason for the new state of the transmitter. The *transmitter* associates with the following states:

RVSIP_TRANSMITTER_STATE_UNDEFINED

Indicates that the *transmitter* was not yet initialized.

RVSIP_TRANSMITTER_STATE_IDLE

The initial state of the transmitter. Upon *transmitter* creation, the *transmitter* assumes the IDLE state. It remains in this state until RvSipTransmitterSendMessage() is called. The *transmitter* then moves to the RESOLVING_ADDR state and starts the address resolution process.

RVSIP_TRANSMITTER_STATE_RESOLVING_ADDR

Indicates that the *transmitter* is about to start the address resolution process. After RvSipTransmitterSendMessage() is called, the *transmitter* copies the supplied *message* and moves to the RESOLVING_ADDR state. The *transmitter* then uses RFC 3261 rules to get the remote URI from the *message*, and the procedures of RFC 3263 to resolve the address by querying the DNS server. When the DNS procedure is completed and the destination address is determined, the *transmitter* moves to the FINAL_DEST_RESOLVED state.

RVSIP_TRANSMITTER_STATE_FINAL_DEST_RESOLVED

Indicates that the *transmitter* has completed the address resolution process and that it has a destination address to use. The application can use the RvSipTransmitterGetDestAddress() function to get the destination address. The application can change the destination address using the RvSipTransmitterSetDestAddress() function. If the application did not set the Fix Via flag, the FINAL_DEST_RESOLVED state is the last chance for the application to fix the Via by itself before the message is sent.

In this state, the application can hold the message sending activity by calling the RvSipTransmitterHoldSending() function. If so, the *transmitter* will move to the ON_HOLD state. Otherwise, the message will be sent when the state changed callback returns.

RVSIP_TRANSMITTER_STATE_ON_HOLD

The *transmitter* assumes the ON_HOLD state if the RvSipTransmitterHoldSending() function is called in the FINAL_DEST_RESOLVED state. The application should call this function if it wishes to manipulate the *transmitter* destination address and DNS list before the message is sent. In this state, the application can use the RvSipTransmitterSetDestAddress() function to change the current destination address or it can supply NULL values to this function and reset the destination address. The application can also manipulate the DNS list that contains the rest of the DNS results. Resetting the destination address will cause the *transmitter* to return to its DNS list for further address resolution.

To continue the sending activities, the application must call the RvSipTransmitterResumeSending() function. If the *transmitter* has a valid destination address, it will continue with the message sending and proceed to the READY_FOR_SENDING state. Otherwise the *transmitter* will return to the RESOLVING_ADDR state.

RVSIP_TRANSMITTER_STATE_READY_FOR_SENDING

Indicates that the *transmitter* is ready to send the message to the remote party. The message in this state has its final format, and the application should no longer change the message content or remote address. This state is informative only.

RVSIP_TRANSMITTER_STATE_NEW_CONN_IN_USE

Indicates that the *transmitter* is about to use a new *connection*. The *connection* can be a totally new *connection* created by the *transmitter* or a *connection* that the *transmitter* found in the *connection* hash. In both cases the pExtraInfo will hold a pointer to a structure of type RvSipTransmitterNewConnInUseStateInfo that includes the *connection* handle.

Two reasons are associated with this state:

- RVSIP_TRANSMITTER_REASON_NEW_CONN_CREATED —indicates that the *transmitter* created a new *connection*.
- RVSIP_TRANSMITTER_REASON_CONN_FOUND_IN_HASH—indicates that the *transmitter* found an existing *connection* in the *connection* hash and that the *transmitter* is about to use the *connection*.

RVSIP_TRANSMITTER_STATE_MSG_SENT

Indicates that the message was sent successfully to the remote party. This state does not supply the *message*. The message is most likely destructed when this state is reached. This state is also an indication that the application should now terminate the *transmitter* that has completed all its tasks and cannot be used any longer.

RVSIP_TRANSMITTER_STATE_MSG_SEND_FAILURE

Indicates that an error occurred and the *transmitter* will not be able to send the message. The state change reason indicates the type of failure. The following reasons are associated with this state:

- RVSIP_TRANSMITTER_REASON_UNDEFINED—a general error caused the failure.
- RVSIP_TRANSMITTER_REASON_NETWORK_ERROR—a network error occurred while trying to send the request or during the DNS procedure.
- RVSIP_TRANSMITTER_REASON_CONNECTION_ERROR—An error occurred on the *connection* that was used to send the message.
- RVSIP_TRANSMITTER_REASON_OUT_OF_RESOURCES- The message cannot be sent because of a lack of resources.

In the message send failure, the application can do one of two things:

- Terminate the *transmitter* using the RvSipTransmitterTerminate() function.
- Send the message again to the next address in the DNS list by calling the RvSipTransmitterSendMessage() function. This option is available only if the enhanced DNS feature is enabled.

Remark: The MSG_SEND_FAILURE state is the way the *transmitter* notifies the application of errors that are caused by incoming events, such as *connection* events or results received from the DNS server. If the error happens in the context of the API function call, the API function will return an error. In this case, the *transmitter* will not move to the MSG_SEND_FAILURE state. However, network errors will always cause the *transmitter* to assume the MSG_SEND_FAILURE state.

RVSIP_TRANSMITTER_STATE_TERMINATED

This is the final state of the transmitter. When a *transmitter* is terminated, the *transmitter* assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *transmitter*.

SENDING BUFFERS WITH TRANSMITTER OBJECTS

A *transmitter* can be used for sending a non-SIP *message* to a specified destination. For example, in some keep-alive mechanisms, the application is required to send a buffer containing CRLF only to detect whether a *connection* has failed. The RvSipTransmitterSendBuffer() function should be used for this purpose.

To send a buffer

1. Create a buffer with the required content.
2. Create a new *transmitter* by calling RvSipTransmitterMgrCreateTransmitter().
3. Set the required destination address to the *transmitter* by calling RvSipTransmitterSetDestAddress(). The buffer will be sent to this address.
4. Call RvSipTransmitterSendBuffer() and supply the required buffer and buffer size.

Note There is no address resolution in the process of buffer sending. The application must supply the *transmitter* with a real IP, transport and port before calling RvSipTransmitterSendBuffer(). Otherwise this function will fail.

The Transmitter state machine is slightly changed when the *transmitter* is used for sending a buffer and several states are omitted. Figure 11-2 shows the Transmitter state machine for buffer sending.

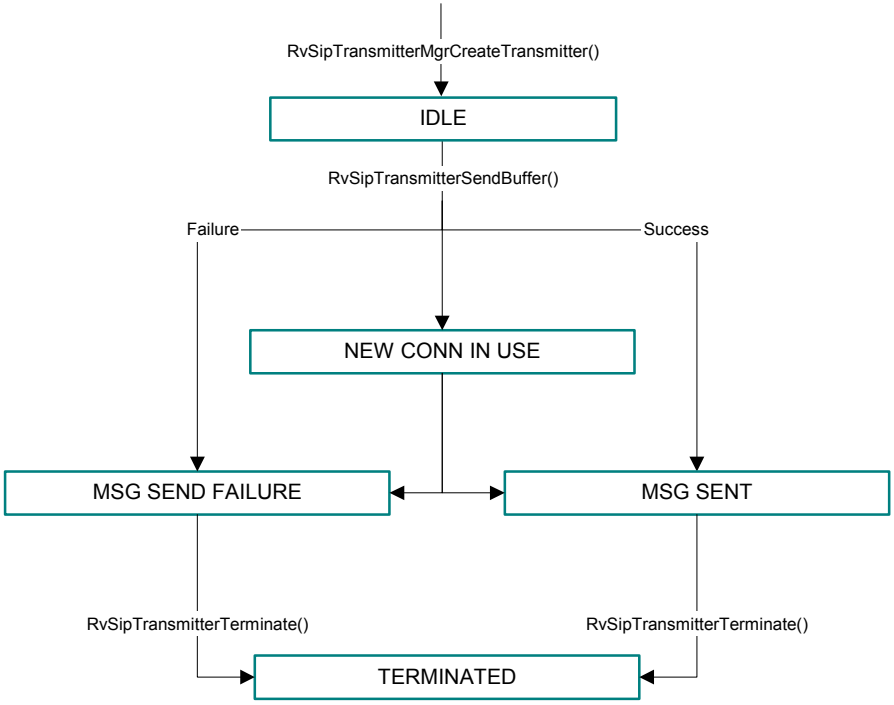


Figure 11-2 *Transmitter State Machine when Sending a Buffer*

TRANSMITTER MANAGER API

The *TransmitterMgr* controls the SIP Stack collection of *transmitters*. You use the Transmitter Manager API to create new *transmitters*. The following Transmitter Manager API function is provided:

RvSipTransmitterMgrCreateTransmitter()

Creates a new *transmitter* and exchanges handles with the application. The new *transmitter* assumes the IDLE state and can be used for sending only one SIP message. When creating a *transmitter* you should supply the *transmitter* event handlers to be notified of *transmitter* events. Each *transmitter* can hold a different set of event handlers. The application is notified about *transmitter* events with the specific event handlers that were supplied.

Sample Code

The following code samples describe how to create a *transmitter* and how to send a message using a *transmitter*.

To create a transmitter

1. Define the event handler functions.
2. Declare a handle for the new transmitter.
3. Set the event handlers in the event handler functions.
4. Call the RvSipTransmitterMgrCreateTransmitter() function and supply the event handlers.

Sample Code

The following code demonstrates an implementation of the *transmitter* state changed callback (step 1). In this sample the application terminates the *transmitter* after the message was sent or if the *transmitter* reaches the MSG_SEND_FAILURE state (next page).

```
/*=========================================================*/
void RVCALLCONV AppTrxStateChangedEv(
             IN RvSipTransmitterHandle          hTrx,
             IN RvSipAppTransmitterHandle       hAppTrx,
             IN RvSipTransmitterState           eState,
             IN RvSipTransmitterReason          eReason,
             IN RvSipMsgHandle                  hMsg,
             IN void*                           pExtraInfo)
{
    switch (eState)
    {
    case RVSIP_TRANSMITTER_STATE_MSG_SENT:
    case RVSIP_TRANSMITTER_STATE_MSG_SEND_FAILURE:
       RvSipTransmitterTerminate(hTrx);
       break;
    default:
       break;
    }
}
/*=========================================================*/
```

Sample Code

The following sample demonstrates how to create a new *transmitter* (steps 2-4).

```
/*=========================================================================*/
RvSipTransmitterHandle AppTrxCreate(RvSipTransmitterMgrHandle hTrxMgr)
{
      RvStatus                      rv      = RV_OK;
      RvSipTransmitterHandle        hTrx    = NULL;

      /*Initializing the event handler structure*/
      RvSipTransmitterEvHandlers    trxEvHandlers;

      memset(&trxEvHandlers,0,sizeof(RvSipTransmitterEvHandlers));
      trxEvHandlers.pfnStateChangedEvHandler = AppTrxStateChangedEv;

      /*Creating a new transmitter - supplying the event structure*/
      rv = RvSipTransmitterMgrCreateTransmitter(
                              hTrxMgr,
                              NULL,
                              &trxEvHandlers,
                              sizeof(RvSipTransmitterEvHandlers),
                              &hTrx);
      if (RV_OK != rv)
    {
            printf("Failed to create a new transmitter\n");
            return NULL;
    }
      return hTrx;
}
/*=========================================================================*/
```

Sending Messages with a Transmitter

The following steps describe how to send a message using a *transmitter*.

1. Create a *transmitter*.
2. Set *transmitter* parameters (optional).
3. Send the message using RvSipTransmitterSendMessage().

Sample Code

The following sample demonstrates how to send a message with a *transmitter*.

```
/*-----------------------------------------------------------------*/
static RvStatus AppTrxSendMsg(RvSipTransmitterMgrHandle hTrxMgr,
                              RvSipMsgHandle         hMsgToSend)
{
    RvStatus                rv;
    RvSipTransmitterHandle  hTrx;

    /*Create a transmitter object*/
    hTrx = AppTrxCreate(hTrxMgr);
    if(hTrx == NULL)
    {
        return RV_ERROR_UNKNOWN;
    }
    /*Set transmitter parameters*/
    RvSipTransmitterSetFixViaFlag(hTrx,RV_TRUE);

    /*Send the message*/
    rv = RvSipTransmitterSendMessage(hTrx,hMsgToSend,RV_FALSE);
    if(rv != RV_OK)
    {
        printf("Failed to send the message\n");
        RvSipTransmitterTerminate(hTrx);
    }
    return rv;
}
/*-----------------------------------------------------------------*/
```

12

EVENT NOTIFICATION

INTRODUCTION

The Event Notification feature, as defined in RFC 3265, provides an extensible framework by which SIP nodes can request notification from remote nodes indicating that certain events have occurred.

The concept behind the Event Notification feature is that entities in the network can subscribe to several resources and/or call states. Each resource and/or call state updates the entity subscribed to it with the states of the specific event.

DEFINITIONS

Notifier

A notifier is an application agent which generates NOTIFY requests for the purpose of notifying subscribers of the state of a resource. Notifiers typically accept SUBSCRIBE requests to create subscriptions.

Subscriber

A subscriber is an application agent which receives NOTIFY requests from notifiers—these NOTIFY requests contain information about the state of a resource in which the subscriber is interested. Subscribers typically generate SUBSCRIBE requests and send them to notifiers to create subscriptions.

Subscription

A subscription is a set of application states associated with a dialog. A subscription includes a pointer to the associated dialog and the event of the subscription. By definition, *subscriptions* exist in both a subscriber and a notifier.

TYPICAL MESSAGE FLOW

Figure 12-1 demonstrates a typical flow of *messages* in a subscription.

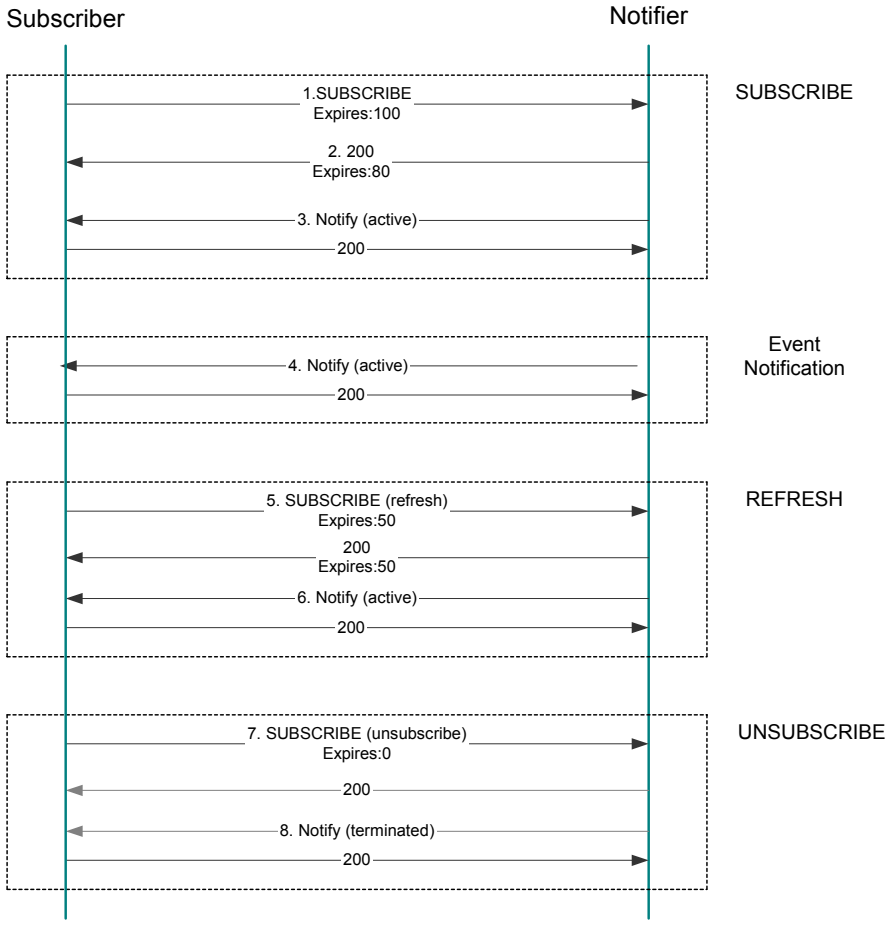


Figure 12-1 *Message Flow*

1. In order to subscribe to a resource or call sate, the subscriber sends a SUBSCRIBE request to the notifier. The SUBSCRIBE

request must include an Expires header that determines the duration of the subscription.

2. The Notifier acknowledges the subscription with a 200 response. It can shorten the subscription expiration by setting a lower value to the Expires header.
3. Upon successfully accepting the subscription, the notifier sends a NOTIFY message immediately to communicate the current resource state to the subscriber, or just as a confirmation of the subscription.
4. NOTIFY messages are sent to inform subscribers of changes in state. (This is actually the purpose of the subscription).
5. In order to increase the subscription duration, the subscriber sends a refresh SUBSCRIBE request, with a new expiration value.
6. Immediately the notifier sends a NOTIFY message. (See step 2).
7. When the subscriber wishes to unsubscribe, it initiates a SUBSCIRBE request with an expiration value of zero.
8. The Notifier accepts the UNSUBSCRIBE request with a 200 response.
9. The Notifier sends a NOTIFY request indicating that the subscription was terminated.

SIP Stack Event Notification Feature

The SIP Stack Event Notification Feature is responsible for creating, maintaining and terminating *subscription*. *Subscriptions* are responsible for sending and receiving notifications about events. A *subscription* can be established by sending a SUBSCRIBE request. However, additional ways exist for establishing *subscriptions*. For more information, see Out-of-Band Subscription.

Each *subscription* can hold several *notifications*. A *notification* is always created using a NOTIFY message. A *subscription* uses an Event header to declare the event type that should get notifications about its state changes. This Event header is also used to match incoming notifications to their *subscription*, and therefore this header must be present in all request messages of this *subscription*.

Since the application is aware of the resource state it is under, its responsibility is to initiate the NOTFIY messages in all situations. The SIP Stack does not send Notify messages automatically. For example, after accepting a SUBSCRIBE request, the application must immediately send a NOTIFY request.

Subscription Entities

The Subscription API of the SIP Stack relates to the following entities:

- Subscription Manager (*SubscriptionMgr*)
- Subscription (*subscription*)
- Notification (*notification*)

Subscription Manager

The *SubscriptionMgr* manages the collection of all *subscriptions* and notifications. The *SubscriptionMgr* is mainly used for creating new *subscriptions*.

Subscription

A *subscription* represents a SIP Subscription as defined in RFC 3265. A *subscription* is associated with a dialog. (Every *subscription* must be related to a dialog. A dialog, however, may hold several *subscriptions*.) A *subscription* can be created inside a *call-leg*, using the *call-leg* dialog, or it can be created independently with its own dialog.

A *subscription* is uniquely identified by the Call-ID, From and To headers which identify the dialog, and by the Event header which identifies the *subscription* in this dialog. Your application can create a *subscription*, refresh it, send notifications and terminate it using the Subscription API.

A *subscription* is a stateful object that can assume any state from a set defined in the Subscription API. A *subscription* state represents the state of the *subscription* setup between two SIP User Agents.

Notification

A *notification* manages the NOTIFY request and NOTIFY response. Each *subscription* holds a list of notifications. The *notifications* hold the notify *transaction* and inform of the *notification* status.

A *notification* exists as long as the notify *transaction* exists. After sending or receiving a response for the NOTIFY request, the *notification* is destructed, and removed from *subscription* list.

Your application can initiate notification, send NOTIFY requests and respond to incoming NOTIFY requests using the Subscription Notify API.

Figure 12-2 shows the relationship of the objects. (Note that hCallLeg in the *subscription* represents a dialog.)

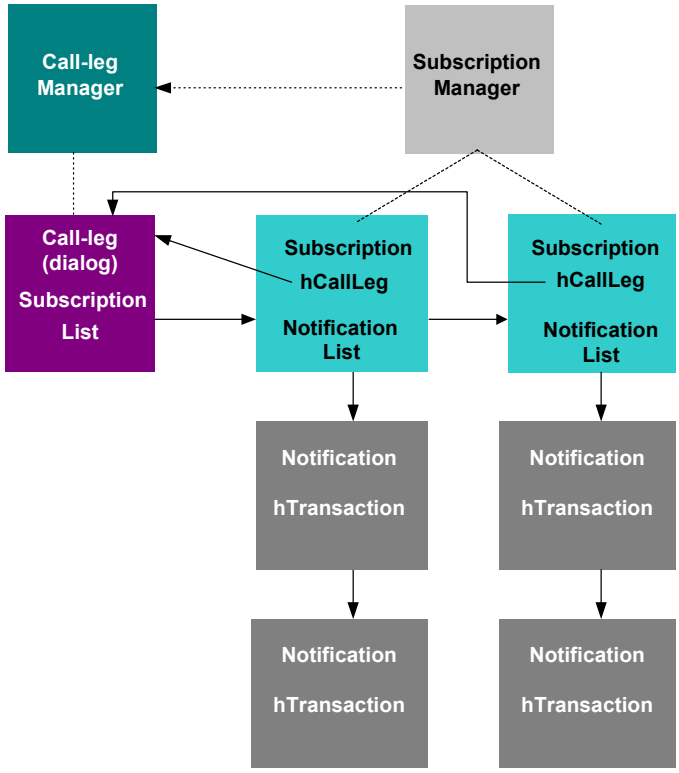


Figure 12-2 *Relationship of Objects*

WORKING WITH HANDLES

All *subscriptions* and the *SubscriptionMgr* are identified using handles. You must supply these handles when using the Subscription API.

- RvSipSubsMgrHandle defines the *SubscriptionMgr* handle. You receive this handle by calling RvSipStackGetSubsMgrHandle().
- RvSipSubsHandle defines a *subscription* handle. For a subscriber *subscription*, you receive the *subscription* handle from RvSipSubsMgrCreateSubscription(). For a notifier *subscription*, you receive the *subscription* handle from the RvSipSubsCreatedEv() callback.

- RvSipNotifyHandle defines a *notification* handle. For an outgoing notification, you receive the *notification* handle from RvSipSubsNotiferCreate(). For an incoming notification, you receive the *notification* handle from the RvSipSubsNotifyCreatedEv() callback.

Subscription Manager API

The *SubscriptionMgr* controls the SIP Stack collection of *subscriptions*. You use the Subscription Manager API to register application callbacks with the SIP Stack and to create new *subscriptions*.

RvSipSubsMgrCreateSubscription()

Creates a new *subscription*.

RvSipSubsMgrSetEvHandlers()

Sets your event handler (callback function) pointers in the SIP Stack.

Subscription Manager Parameters

You can set or examine *subscription* parameters via *SubscriptionMgr* Set and Get API functions. The following parameters are available:

Stack Instance

The handle of the SIP Stack instance to which this *SubscriptionMgr* belongs.

AppMgrHandle

The handle to the application *SubscriptionMgr*.

Subscription API

The Subscription API contains a set of functions and function callbacks that allow you to set or examine *subscription* parameters, control *subscription*, send notifications, and respond to network events.

Subscription Parameters

You can set or examine *subscription* parameters via Subscription Set and Get API functions. The following parameters are available:

Event Header

Uniquely identifies a *subscription* in a dialog. When creating a subscriber *subscription*, you must set the Event header of the *subscription*.

Expires Header

Defines the lifetime of the *subscription*. Before sending an initial SUBSCRIBE request, you may set the Expires header value for the *subscription*. The expires value must be smaller than RVSIP_SUBS_EXPIRES_VAL_LIMIT value (4000000). This value will be set in the SUBSCRIBE request as the requested expiration time. The notifier, however, may shorten the expiration value by setting an Expires header with a lower value in the 2xx response. The subscriber can update the expiration value of the *subscription* later, by sending a refresh request. In this case too, the notifier can shorten the requested expiration value in the 200 response. The notifier can also shorten the *subscription* expiration value by sending a NOTIFY request with the Expires parameter in the Subscription-state header. You can get the last expiration value that was set for the *subscription* at all stages of the *subscription*. You can only set the expiration value before sending the initial SUBSCRIBE request. Note that for all cases (initial SUBSCRIBE, refresh, and an expires parameter in a NOTIFY request) the expires value must be smaller than RVSIP_SUBS_EXPIRES_VAL_LIMIT value (4000000).

Requested Expires

The expires value that was within the SUBSCRIBE request that was received from the subscriber.

Remaining Time

The remaining time of a *subscription*, in seconds.

Dialog Parameters (To, From, CallId, Cseq etc.)

A *subscription* created inside a *call-leg* takes the *call-leg* dialog parameters. An independent *subscription* has its own dialog, therefore To and From parameters should be set in the dialog. You can get the dialog handle from the *subscription* (in a RvSipCallLegHandle format) and use it to examine other dialog parameters.

Note The To and From parameters can only be set by using the *subscription* initialization function and not the dialog handle. The dialog handle can be used for the other dialog parameters.

As in a *call-leg*, you can either set the Call-ID of the dialog, or the SIP Stack will generate the Call-ID. The dialog automatically handles the CSeq, which is increased by one for every outgoing request.

Note You must not use the Call-leg Control API with the dialog handle if the *subscription* was created independently (not related to a connected *call-leg*).

Received Message

The last SUBSCRIBE message (Request or Response) that was received by the *subscription*.

You can get this message only in the context of the *subscription* state changed callback function when the new state indicates that the *subscription* received a new message.

Outbound Message

The SUBSCRIBE message (Request or Response) that the *subscription* is going to send.

You can call this function before you call the Subscription Control API functions that send a message (such as RvSipSubsSubscribe()). You should use this function to add headers to the message before it is sent.

Note The outbound message you receive is not complete. In some cases it might even be empty.

Application Subscription Handle

The handle to the application *subscriptions*.

Current Local Address

The local address that is used by a *subscription* transaction.

Stack Instance

The handle to the Stack instance to which this *subscription* belongs.

State

Indicates the state of the *subscription* setup between two SIP User Agents. You can only access the *state* parameter with a Get function and it is not modifiable.

Subscription Type

Indicates whether the *subscription* represents the subscriber or notifier side of the *subscription*. You can only access the type parameter with a Get function and it is not modifiable.

Event Package Type

The type of event package of a *subscription*.

Reject Status Code On Creation

You can set the "reject status code on creation" parameter to automatically reject the request that created this *subscription*. If you set this status code, the *subscription* will be destructed automatically when the RvSipSubsCreatedEv() callback returns. You will not get any further callbacks that relate to this *subscription* (you will not get the RvSipSubsMsgToSendEv() for the reject response message or the TERMINATED state for the *subscriptions*). This parameter should not be used for rejecting a request in a normal scenario. For this you should use the RvSipSubsRespondReject() function. You should use this function only if your application is incapable of handling this new *subscription* at all, for example, if the application is out of resources.

Remark: When this function is used to reject a request, you cannot use the outbound message mechanism to add information to the outgoing response message. If you wish to change the response message, you must use the regular reject function in the RVSIP_SUBS_STATE_SUBS_RCVD state.

Subscription Control

The following API functions provide *subscription* control:

RvSipSubsInit()

After creating a *subscription*, you can use this function to set the To, From and Event headers, and the Expires header value in the *subscription*. If the *subscription* was created within an existing *call-leg*, you should not set the To and From parameters.

RvSipSubsInitStr()

After creating a *subscription*, you can use this function to set the To, From and Event headers value in a string format, and the Expires header value in the *subscription*. If the *subscription* was created within an existing *call-leg*, you should not set the To and From parameters.

RvSipSubsDialogInit()

After creating a *subscription*, you can use this function to set the To, From, Local and Remote Contact parameters of the *subscription*'s dialog. This function is relevant for an independent *subscription* (with no *call-leg*).

RvSipSubsDialogInitStr()

After creating a *subscription*, you can use this function to set the To, From, Local and Remote Contact parameters of the *subscription*'s dialog in string format. This function is relevant for an independent *subscription* (with no *call-leg*).

RvSipSubsSubscribe()

After creating and initializing a *subscription*, you can use RvSipSubsSubscribe() for generating and sending the required SUBSCRIBE message to establish the *subscription*.

RvSipSubsRefresh()

After the *subscription* was established successfully, it assumes the SUBS_ACTIVE state. In this state the subscriber may refresh the timer of the *subscription* by sending another SUBSCRIBE request on the same dialog as the existing *subscription*. For this purpose use RvSipSubsRefresh(). It generates and sends a refresh SUBSCRIBE message with a new Expires header, and updates the Subscription state machine to progress to the SUBS_REFRESHING state.

RvSipSubsUnsubscribe()

After the *subscription* was established successfully, it assumes the SUBS_ACTIVE state. In every state from now on, the subscriber may ask the notifier to terminate the *subscription* by sending another SUBSCRIBE request on the same dialog as the existing *subscription* with the Expires header equal to 0. (Such a subscribe request is called "unsubscribe"). For this purpose use

RvSipSubsUnsubscribe(). It generates and sends a SUBSCRIBE message with an “Expires:0” header, and updates the Subscription state machine to progress to the SUBS_UNSUBSCRIBING state.

Note After receiving a successful UNSUBSCRIBE request, your application should wait for a final NOTIFY message.

RvSipSubsRespondAccept()

When the notifier receives a SUBSCRIBE request, it can accept the request with this function.

You can use RvSipSubsRespondAccept() in the SUBS_RCVD state to accept an incoming *subscription*. You can also use this function to accept a Refresh UNSUBSCRIBE request received by an established *subscription*. The user must send a NOTIFY request with a “Subscription-State:active” header after calling RvSipSubsRespondAccept().

RvSipSubsRespondPending()

When the notifier receives an initial SUBSCRIBE request, it can send a 202 pending response with this function.

You can use RvSipSubsRespondPending() in the SUBS_RCVD state to send a 202 response of an unauthorized incoming *subscription*. (The 202 response indicates that the request has been received and understood, but does not necessarily imply that the *subscription* has been authorized yet). The user must send a NOTIFY request with a “Subscription-State:pending” header in it after calling RvSipSubsRespondPending().

Note You can **not** send a 202 response to a Refresh UNSUBSCRIBE request; it is only applicable to the first SUBSCRIBE request.

RvSipSubsRespondReject()

You can use this function in the SUBS_RCVD state to reject an incoming *subscription*. You can also use this function to reject a REFRESH and UNSUBSCRIBE request received by an established *subscription*.

RvSipSubsTerminate()

Causes an immediate shut-down of the *subscription*, without sending any messages (UNSUBSCRIBE or NOTIFY). The *subscription* will assume the TERMINATED state. Calling this function causes an abnormal termination of the *subscription*. All notifications related to the *subscription* will also be terminated.

RvSipSubsAutoRefresh()

Sets or unsets the Auto-refresh option in a *subscription* (Sets it to on or off). The Auto-refresh mode sends a REFRESH request automatically when the *subscription* timer is about to be expired. (In Auto-refresh mode, RvSipSubsExpirationAlertEv() will not be called.)

Note This function sets the Auto-refresh mode per *subscription*. You can use a configuration parameter to set to all the *subscriptions* at once.

RvSipSubsSetAlertTimer()

Defines how long the RvSipSubsExpirationAlertEv() callback function is called before the *subscription* expires. If you do not call this function a default value is used.

Note This function sets the Alert Timer per *subscription*. You can use a configuration parameter to set to all the *subscriptions* at once.

RvSipSubsSetNoNotifyTimer()

Defines how long to wait for a NOTIFY request after receiving a 2xx on a SUBSCRIBE request. When this timer expires, the *subscription* is terminated.

Note This function sets a no-notifier-timer per *subscription*. You can use a configuration parameter to set all the *subscriptions* at once.

RvSipSubsDetachOwner()

Detaches the *subscription* from its owner.

RvSipSubsUpdateSubscriptionTimer()

Sets the *subscription* timer. Calling this function will activate the *subscription* timer again after it expired.

NOTIFICATION CONTROL API

The following API functions provide *notification* control:

RvSipSubsCreateNotify()

After a *subscription* was established successfully, you can use this function to create a Notify object in this *subscription*. The application can get the NOTIFY outbound message (using RvSipNotifyGetOutboundMsg()) to set needed information to this message. To send the NOTIFY message you should use RvSipNotifySend().

RvSipNotifySend()

After creating a Notify object and setting information in the NOTIFY message, you can use this function to send the required NOTIFY message.

RvSipNotifyAccept()

Accepts an incoming NOTIFY request.

RvSipNotifyReject()

Rejects an incoming Notify request.

RvSipNotifyTerminate()

Causes an immediate shutdown of the *notification*.

RvSipNotifyDetachOwner()

Detaches the Notify object from its owner. The owner will not be informed on the Notify object events after detaching from it.

NOTIFICATION PARAMETERS

The *notification* parameters are as follows:

Outbound Message

The NOTIFY message (request or response) that the *notification* is going to send. You get this parameter using the RvSipNotifyGetOutboundMsg() function.

You can call RvSipNotifyGetOutboundMsg() before you call the Notification Control API functions that send a message. You should use this function to add event status information to the message before it is sent.

Notify SubsState

The value of the Subscription-State header that was set to the NOTIFY request of this Notify object.

Stack Instance

The handle to the SIP Stack instance to which this Notify object belongs.

SUBSCRIPTION EVENTS

The Subscription API supplies several events, in the form of callback functions, to which your application may listen and react. In order to listen to an event, your application should first define a special function called the event handler and then pass the event handler pointer to the *SubscriptionMgr*. When an event occurs, the *subscription* calls the event handler function using the pointer.

The following events are supplied with the Subscription API:

RvSipSubsCreatedEv()

Notifies the application that a new incoming *subscription* has been created. The newly created *subscription* always assumes the IDLE state. Your application can exchange handles with the SIP Stack using this callback.

RvSipSubsStateChangedEv()

This event is probably the most useful of the events that the SIP *subscription* reports. Through this function, you receive indications of SIP *subscription* state changes and the associated state change reason. For example, upon receipt of an SUBS_RCVD state indication, your application may decide whether to accept or reject the *subscription*.

RvSipSubsSubscriptionExpiredEv()

Notifies the application that the *subscription* timer expired at this point. The notifier should send a NOTIFY request with a “Subscription-State:terminated” header in it. The subscriber may try to send a refresh SUBSCRIBE request or wait for the NOTIFY request with “terminated” as the Subscription-State header value.

RvSipSubsExpirationAlertEv()

Alerts the application that the *subscription* timer is about to expire. The application should use this callback to send a refresh SUBSCRIBE request.

The time interval before expiration for this callback to be called is set with the "SubscriptionAlertTime" configuration parameter or with the RvSipSubsSetAlertTimer() function for a specific *subscription*.

Note This event function will not be called when working in Auto-refresh mode.

RvSipSubsMsgToSendEv()

The *subscription* calls this event whenever a *subscription*-related message is ready to be sent. You can use this callback for changing or examining a message before it is sent to the remote party.

RvSipSubsMsgReceivedEv()

The *subscription* calls this event whenever a *subscription*-related message has been received and is about to be processed. You can use this callback to examine incoming messages.

RvSipSubsNotifyCreatedEv()

Indicates that a new *notification* was created and exchanges handles with the application.

RvSipSubsNotifyEv()

Notifies the application of a notify status. Through this function you receive indications of SIP notify status and the associated reason. For example, upon receipt of a NOTIFY-RCVD status, your application may decide whether to accept or reject the NOTIFY request.

A subscriber application gets the received NOTIFY request message in this event, and its related *notification* handle. At this callback, the application can get all needed information from the NOTIFY request message. The message will be destructed after calling this callback.

A notifier application gets the received NOTIFY response message that was received for a NOTIFY request. At the end of this callback, the Stack *notification* is destructed.

This event notifies both subscriber and notifier applications about the termination of the *notification*.

RvSipSubsFinalDestResolvedEv()

Notifies the application that the *subscription* is about to send a message after the destination address was resolved.

SUBSCRIPTION STATE MACHINE

The Subscription state machine represents the state of the *subscription* establishment between two SIP User Agents. The RvSipSubsStateChangedEv() callback reports *subscription* state changes and state change reasons. The state change reason indicates how the *subscription* reached the new state.

The *subscription* associates with the following states:

RVSIP_SUBS_STATE_IDLE

The IDLE state is the initial state of the Subscription state machine. Upon *subscription* creation, the *subscription* assumes the IDLE state. It remains in this state until RvSipSubsSubscribe() is called, whereupon it should move to the SUBS_SENT state.

RVSIP_SUBS_STATE_SUBS_SENT

After calling RvSipSubsSubscribe() and sending a SUBSCRIBE request, the *subscription* enters the SUBS_SENT state. The *subscription* remains in this state until it receives a final response from the notifier. If a 2xx class response is received, the *subscription* assumes the SUBS_2XX_RCVD state. If a 3xx class response is received, the *subscription* moves to the REDIRECTED state. If the *subscription* is rejected with a 401 or 407 response, the *subscription* moves to the UNAUTHENTICATED state. For all other response codes, *subscription* assumes the TERMINATED state.

If a NOTIFY request is received before a 2xx response is received, the *subscription* assumes the NOTIFY_BEFORE_2XX_RCVD state.

RVSIP_SUBS_STATE_REDIRECTED

A *subscription* in the SUBS_SENT state may receive a 3xx class response. In this case, the *subscription* assumes the REDIRECTED state. At this point, you may confirm the redirection by calling the RvSipSubsSubscribe() function again. You can also terminate the *subscription* using the RvSipSubsTerminate() function.

RVSIP_SUBS_STATE_UNAUTHENTICATED

A *subscription* in the SUBS_SENT state may receive a 401 or 407 response. In this case, the *subscription* assumes the UNAUTHENTICATED state. At this point, you can re-send your request with authentication information by calling the RvSipSubsSubscribe() function again. You can also terminate the call using the RvSipSubsTerminate() function.

RVSIP_SUBS_STATE_2XX_RCVD

Upon receipt of a 2xx response to an initial SUBSCRIBE request, the *subscription* assumes the 2XX_RCVD state. The *subscription* remains in this state until it receives and accepts a NOTIFY request from the notifier. The SIP Stack sets a timer in this state (called a subsNoNotifyTimer). If this timer expires before the NOTIFY request is received and accepted, the *subscription* assumes the TERMINATED state.

When the application receives and accepts the NOTIFY request, it releases the subsNoNotifyTimer and the *subscription* assumes the ACTIVE or PENDING state (according to the Subscription-State header in the NOTIFY request).

RVSIP_SUBS_STATE_NOTIFY_BEFORE_2XX_RCVD

If the *subscription* received a NOTIFY request in the SUBS_SENT state, the *subscription* assumes the NOTIFY_BEFORE_2XX_RCVD state. When the *subscription* receives the 2xx response on the SUBSCRIBE request, it will assume the ACTIVE or PENDING state (according to a 200 or 202 response).

RVSIP_SUBS_STATE_SUBS_RCVD

Upon receipt of the initial SUBSCRIBE request by a notifier *subscription*, the *subscription* assumes the SUBS_RCVD state. In this state, it is up to the application to decide whether to accept or reject the *subscription* using the Subscription API.

RVSIP_SUBS_STATE_PENDING

The PENDING state indicates that the initial SUBSCRIBE request has been received and understood, but the *subscription* has not been authorized yet.

A subscriber *subscription* reaches this state when a 202 final response is received for an initial SUBSCRIBE request, and a NOTIFY request with “pending” as the Subscription-State header value is received and accepted.

A notifier *subscription* reaches this state when a 202 final response is sent for an initial SUBSCRIBE request.

RVSIP_SUBS_STATE_ACTIVATED

When a notifier *subscription* wants to move from PENDING state to ACTIVE state, it sends a NOTIFY request with a "Subscription-State:active" header in it. When sending the request, the *subscription* assumes the ACTIVATED state. When the notifier receives a 2xx response for this NOTIFY request, the *subscription* assumes the ACTIVE state. If the notifier receives a non-2xx response for this NOTIFY request, it remains in the ACTIVATED state.

RVSIP_SUBS_STATE_ACTIVE

The *subscription* is active—this state indicates a successful *subscription* establishment.

A subscriber *subscription* reaches this state when a 2xx final response is received for the initial SUBSCRIBE request, and a NOTIFY request with "active" as the Subscription-state header value is received and accepted.

A notifier *subscription* reaches this state when a 200 final response is sent for an initial SUBSCRIBE request, or when a 2xx response is received for a NOTIFY request with "active" as the Subscription-State header value.

RVSIP_SUBS_STATE_REFRESHING

A subscriber *subscription* in ACTIVE state may call the Refresh() function to send a refresh SUBSCRIBE request. After sending the refresh SUBSCRIBE request, the *subscription* enters the REFRESHING state. The *subscription* remains in this state until it receives a final response from the notifier.

If a 2xx class response is received, the *subscription* sets the *subscription* timer to the new value that both the subscriber and notifier had agreed on in the refresh SUBSCRIBE message and the 2xx response. If a non-2xx response is received, the *subscription* timer remains as it was.

In both cases, when a response is received to the refresh SUBSCRIBE request, the *subscription* assumes the ACTIVE state.

RVSIP_SUBS_STATE_REFRESH_RCVD

Upon receipt of a refresh SUBSCRIBE request by a notifier *subscription*, the *subscription* assumes the REFRESH_RCVD state. In this state, the application must decide whether to accept or reject the *subscription* refreshing using the Subscription API.

RVSIP_SUBS_STATE_UNSUBSCRIBING

A subscriber *subscription* can call the RvSipSubsUnsubscribe() function to send an UNSUBSCRIBE request. After sending the UNSUBSCRIBE, the *subscription* enters the UNSUBSCRIBING state. The *subscription* remains in this state until it receives a final response from the notifier.

If a 2xx class response is received, the *subscription* assumes the UNSUBSCRIBE_2XX_RCVD state, while waiting for the NOTIFY request that terminates the *subscription*. If a non-2xx response is received, the *subscription* assumes the previous state—the state that was before sending the UNSUBSCRIBE request. For more information, see Table 12-1.

RVSIP_SUBS_STATE_UNSUBSCRIBE_RCVD

Upon receipt of an UNSUBSCRIBE request by a notifier *subscription*, the *subscription* assumes the UNSUBSCRIBE_RCVD state. In this state, the application must decide whether to accept or reject the UNSUBSCRIBE request using the Subscription API.

If the application rejects the request, the *subscription* assumes the previous state—the state that was before receiving the unsubscribe request. For more information, see Table 12-2.

If the application accepts the request, it must immediately send a NOTIFY request with a “Subscription-State:terminated” header in it. When sending this NOTIFY request, the *subscription* assumes the TERMINATING state.

RVSIP_SUBS_STATE_UNSUBSCRIBE_2XX_RCVD

Upon receipt of a a 2xx class response on an UNSUBSCRIBE request, the *subscription* assumes the UNSUBSCRIBE_2XX_RCVD state. The *subscription* remains in this state until it receives a NOTIFY request from the notifier with “Subscription-State:terminated” header in it. The SIP Stack sets a timer in this state (called a subsNoNotifyTimer). If this timer expires before the NOTIFY request is received and accepted, the *subscription* assumes the TERMINATED state.

RVSIP_SUBS_MSG_SEND_FAILURE

Failure in sending a Subscribe request because of a timeout, network error, or 503 response. The application may try to re-send the request to the next IP address in the DNS list. For more information, see the *Call-leg DNS Functions* section in the *Call-leg Functions* chapter in the *SIP Stack Reference Guide*.

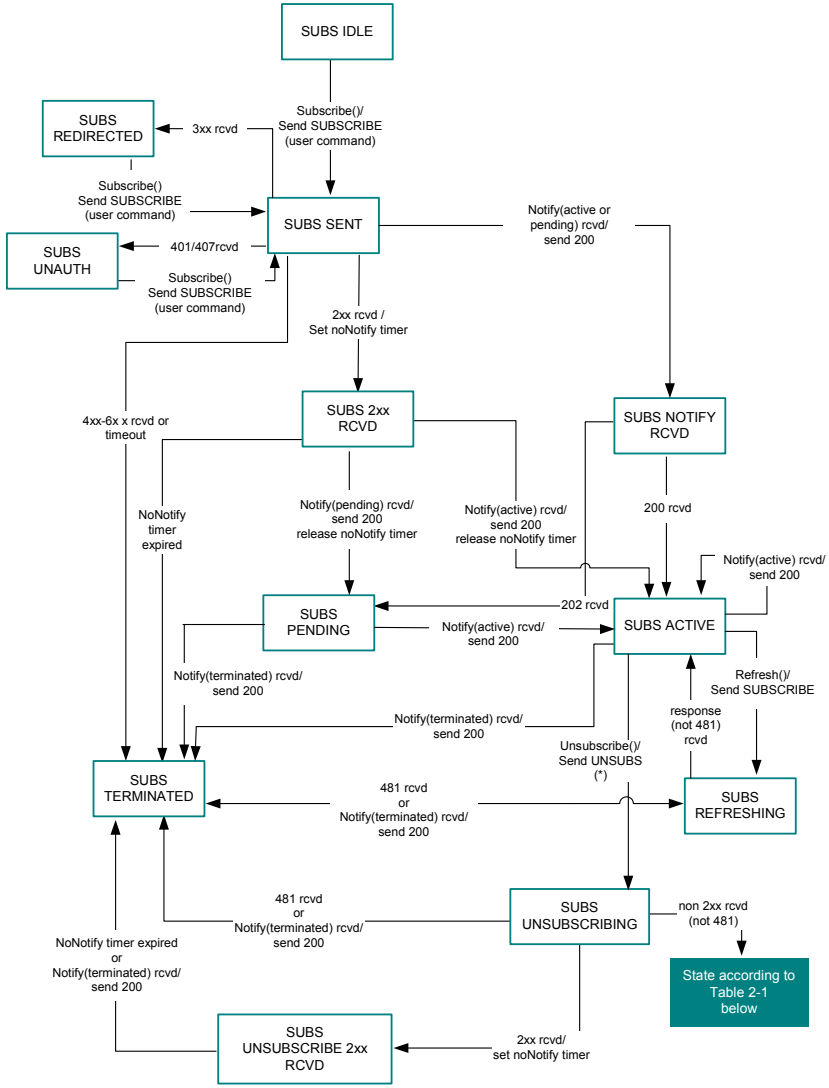
RVSIP_SUBS_STATE_TERMINATING

When a notifier *subscription* sends a NOTIFY request with a “Subscription-State: terminated” header in it, the *subscription* assumes the TERMINATING state. When the notifier receives the 2xx response for this NOTIFY request, the *subscription* assumes the TERMINATED state. If the notifier receives a non-2xx response for this NOTIFY request, it remains in the TERMINATING state.

RVSIP_SUBS_STATE_TERMINATED

This state is the final *subscription* state. When a *subscription* is terminated, it assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *subscription* using the Subscription API functions.

SUBSCRIBER STATE MACHINE



(*)Moving to state SubsUnsubscribing is possible from state Subs-Pending, Subs-Active an Subs-2xx-Rcvd

NOTE Sending a non-2xx response on a Notify request does not influence the state machine. A Notify request influences the state machine only if it is responded to with 2xx.

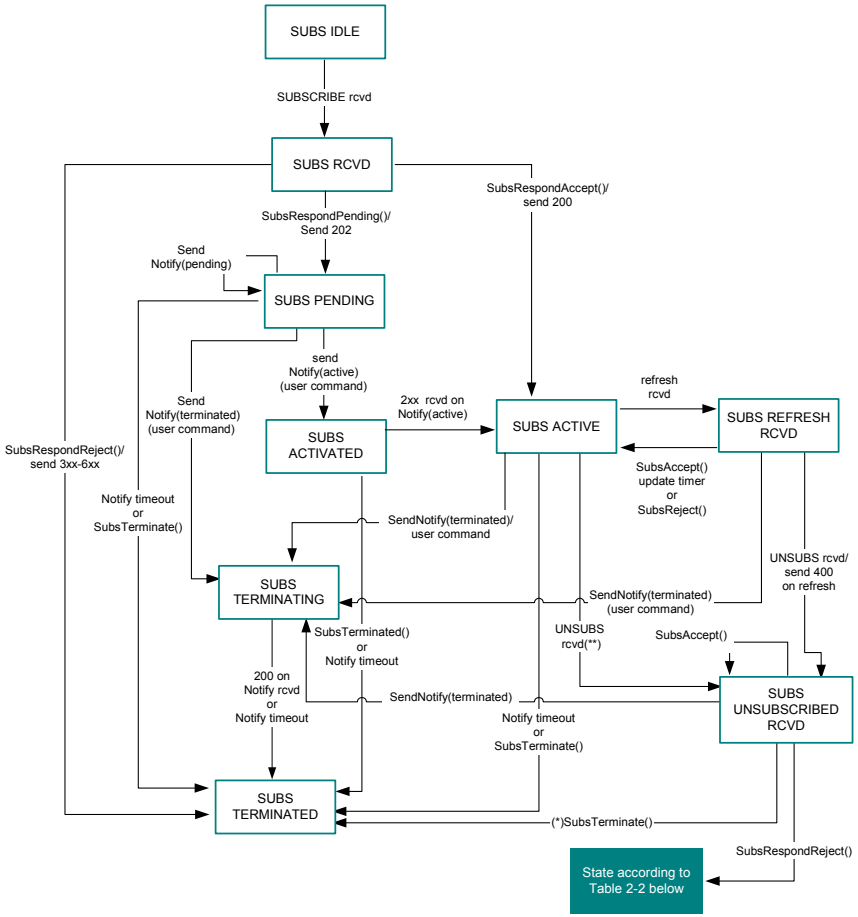
Figure 12-3 *Subscriber State Machine*

Table 12-1 shows how the state changes after receiving a reject on UNSUBSCRIBE.

Table 12-1 *State Changes*

State before Sending UNSUBSCRIBE Request	State after receiving Reject on UNSUBSCRIBE Request	Reason for Behavior
SUBS_ACTIVE	SUBS_ACTIVE	
SUBS_PENDING	SUBS_PENDING	
SUBS_2XX_RCVD	1. SUBS_2XX_RCVD	No NOTIFY with “active” as the Subscription-State header value was received and accepted while the state was unsubscribing.
	2. SUBS_PENDING	A NOTIFY with “pending” as the Subscription-State header value was received and accepted while the state was unsubscribing.
	3.SUBS_ACTIVE	A NOTIFY with “active” as the Subscription-State header value was received and accepted while the state was unsubscribing.

NOTIFIER STATE MACHINE



(*)Calling to SubsTerminate() changes notifier state to Subs-Terminated at any state of subscription

(**)Moving to Subs-Unsubscribe-Rcvd state is possible from Subs-Active, Subs-Pending, Subs-Refresh-Rcvd and Subs-Activated.

Figure 12-4 *Notifier State Machine*

Table 12-2 shows how the state changes after rejecting an UNSUBSCRIBE request.

Table 12-2 *State Changes*

State that Received UNSUBSCRIBE Request	State after Sending Reject on UNSUBSCRIBE	Reason for Behavior
SUBS_ACTIVE	SUBS_ACTIVE	
SUBS_PENDING	SUBS_PENDING	
SUBS_REFRESH_RCVD	SUBS_ACTIVE	Refresh was rejected with 400 when UNSUBSCRIBE was received.
SUBS_ACTIVATED	1. SUBS_ACTIVATED	No 2xx on NOTIFY with "active" as the Subscription-State header value was received yet.
	2. SUBS_ACTIVE	A 2xx on NOTIFY with "active" as the Subscription-State header value was received while the state was UBSUBS_RCVD.

NOTIFICATION OBJECTS

You create a notification object (*notification*) by calling the RvSipSubsCreateNotify() function. By calling RvSipNotifySend(), a NOTIFY request is sent.

An incoming NOTIFY request also creates a *notification*. A *notification* terminates when the notify *transaction* terminates. Figure 12-5 illustrates a typical NOTIFY flow.

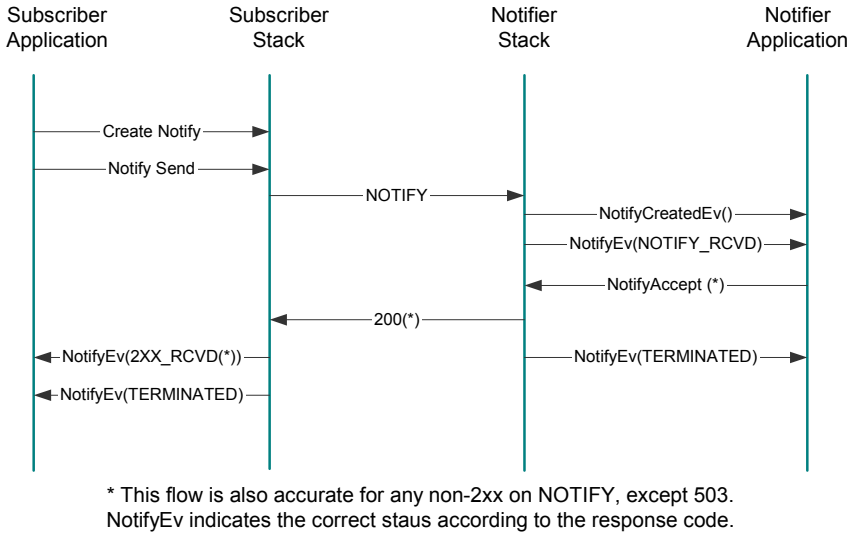


Figure 12-5 *NOTIFY flow*

Figure 12-6 illustrates a special case in which a *notification* failed to send the request message (*transaction* timeout, network error, or a 503 response was received). In this case the *notification* is not terminated, and the application can re-send the NOTIFY request to the next IP address on the *transaction* DNS list. For more information, see the Working with DNS chapter.

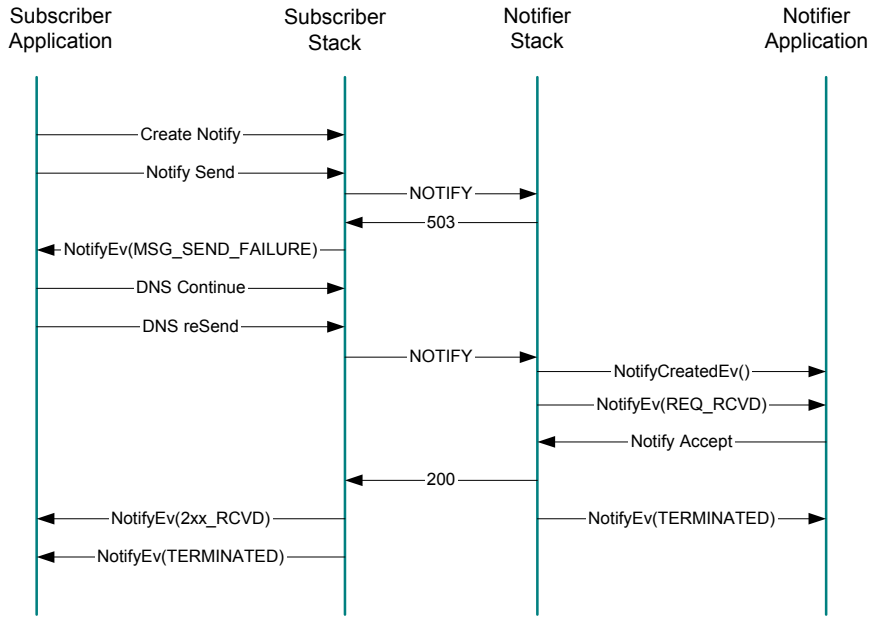


Figure 12-6 *NOTIFY Request Failure*

NOTIFY STATUS

Notify status represents the status of the *notification*. The RvSipSubsNotifyEv() callback reports the *notification* status and status reason.

The *notification* associates with the following statuses:

RVSIP_SUBS_NOTIFY_STATUS_REQUEST_RCVD

Upon receipt of a NOTIFY request by a subscriber *subscription*, the *notification* informs the REQUEST_RCVD status. At this stage, the application must decide whether to accept or reject the NOTIFY using the Subscription Notify API.

RVSIP_SUBS_NOTIFY_STATUS_REDIRECTED

Upon receipt of a 3xx response to a NOTIFY request, the *notification* informs the REDIRECTED status. At this stage, the application may get needed information from the 3xx message. After this stage, the *notification* is terminated and informs the TERMINATED status.

RVSIP_SUBS_NOTIFY_STATUS_UNAUTHENTICATED

Upon receipt of a 401/407 response to a NOTIFY request, the *notification* informs the UNAUTHENTICATED status. At this stage, the application may get needed information from the response message. After this stage, the *notification* is terminated and informs the TERMINATED status.

RVSIP_SUBS_NOTIFY_STATUS_REJECT_RCVD

Upon receipt of a non-2xx response to a NOTIFY request, the *notification* informs the REJECT_RCVD status. At this stage, the application may get needed information from the response message. After this stage, the *notification* is terminated and informs the TERMINATED status.

RVSIP_SUBS_NOTIFY_STATUS_2XX_RCVD

Upon receipt of a 2xx response to a NOTIFY request, the *notification* informs the 2XX_RCVD status. At this stage, the application may get needed information from the 2xx message. After this stage, the *notification* is terminated and informs the TERMINATED status.

RVSIP_SUBS_NOTIFY_STATUS_MSG_SEND_FAILURE

A notifier that failed to send a NOTIFY request (receives a 503 response, *transaction* timeout or network error) informs the MSG_SEND_FAILURE status. At this stage, the application must decide whether to re-send this message to the next IP address from the DNS list, or stop sending this message.

If the application stops sending the message, the *notification* is terminated and informs the TERMINATED status. If application re-sends the request to the next IP address, the SIP Stack use the same *notification* to re-send the NOTIFY request.

RVSIP_SUBS_NOTIFY_STATUS_TERMINATED

The final status of which the *notification* gives notification. Upon receiving the Terminated status, you can no longer reference the *notification*.

RVSIP_SUBS_NOTIFY_STATUS_IDLE

The initial status of a *notification*.

RVSIP_SUBS_NOTIFY_STATUS_REQUEST_SENT

Upon sending a NOTIFY request, the *notification* informs the application of the Request-Sent status. At this stage, the *notification* waits for a response.

RVSIP_SUBS_NOTIFY_STATUS_FINAL_RESPONSE_SENT

Upon sending a NOTIFY response, the *notification* informs the application of the Final-Response-Sent status. After this, the *notification* is terminated and the application is informed of the Terminated status.

EXCHANGING HANDLES WITH THE APPLICATION

The SIP Stack enables you to create your own handle to a *subscription*. This will prove useful when you have your own application *subscription* database. You can provide the SIP Stack with your *subscription* handle, which the Stack must supply when calling your application callbacks.

You can use RvSipSubsMgrCreateSubscription() to exchange handles for subscriber *subscriptions* and RvSipSubsCreatedEv() to exchange handles for notifier *subscription*.

INITIATING A SUBSCRIPTION

The following steps describe how to initiate a *subscription*:

To initiate a *subscription*

1. Declare a handle for the new *subscription*.
2. Call RvSipSubsMgrCreateSubscription(). This function enables you to exchange handles with the SIP Stack.
3. Call RvSipSubsInit() or RvSipSubsInitStr(). These functions initialize the *subscription* with To, From, Event and Expires headers.
4. Call RvSipSubsRvSipSubsSubscribe(). This function sends a SUBSCRIBE request to the remote party.

Sample Code

The following sample code demonstrates an implementation of the *subscription* creation procedure.

```
/*===========================================================================*/
void AppCreateSubscription(IN RvSipSubsMgrHandle hSubsMgr)
{
    /*The handle to the subscription.*/
RvSipSubsHandle hSubs;
```

```
    RvChar   *strFrom = "sip:user1@172.20.0.1:5060";
    RvChar   *strTo = "sip:user2@172.20.10.11:5060";
    RvChar   *strEvent = "hp.event; id=222";
    RvInt32  expiresVal = 3000;
    RvStatus rv;
    /*---------------------------
    Creates a new subscription.
    ----------------------------*/
    rv = RvSipSubsMgrCreateSubscription(hSubsMgr, NULL, NULL, &hSubs);

    if(rv != RV_OK)
    {
        printf("Failed to create new subscription\n");
        return;
    }
    printf("Subscriber subscription %x was created\n",hSubs);
    /*-------------------------------------------------------------
    Calls the init function with the To, From and Event strings.
    -------------------------------------------------------------*/
    rv = RvSipSubsInitStr(hSubs, strFrom, strTo, expiresVal, strEvent);
    if(rv != RV_OK)
    {
        printf("Failed to initiate new subscription\n");
        return;
    }

    printf("Sending a SUBSCRIBE request: %s ->
    %s\n",strFrom,strTo);
    rv = RvSipSubsSubscribe(hSubs);
    if(rv != RV_OK)
    {
        printf("Subscribe failed for subscription %x", hSubs);
        return;
    }
}
/*===========================================================================*/
```

SENDING A NOTIFICATION

The following steps describe how to create and send a NOTIFY request:

1. Declare a handle for the new *notification* and for the NOTIFY request message.
2. Call RvSipSubsCreateNotify(). This function enables you to exchange handles with the SIP Stack.
3. Call RvSipNotifyGetOutboundMsg() to get the NOTIFY message.
4. Set the Subscription-State header in the message.
5. Set the event status information in the message.
6. Call RvSipNotifySend(). This function sends the NOTIFY request to the remote party.

Sample Code

The following sample code demonstrates an implementation of the notify sending procedure.

```
/*-------------------------------------------------------------------------------*/
RvStatus AppCreateNotification(RvSipSubsHandle           hSubs,
                               RvSipSubscriptionSubstate eSubsState,
                               RvSipSubscriptionReason   eReason)
{
    /* The handle to the notification object */
    RvSipNotifyHandle hNotify;
    /*The handle to the NOTIFY message */
    RvSipMsgHandle    hNotifyMsg;
    RvStatus rv;
    /*------------------------------
    Creates the notification object.
    ------------------------------*/
    rv = RvSipSubsCreateNotify(hSubs,
                               (RvSipAppNotifyHandle)hSubs,
                               &hNotify);
    if(rv != RV_OK)
    {
        printf("Failed to create a new notification");
        return rv;
    }

    /*-----------------------------------------------------------------
    Sets the Subscription-State header parameters in the NOTIFY message.
    -----------------------------------------------------------------*/
```

```
    RvSipNotifySetSubscriptionStateParams(hNotify, eSubsState, eReason, UNDEFINED);

    /*-------------------------------------------------
    Gets the outbound NOTIFY request message, in order
    to set the status information to it.
    -------------------------------------------------*/
    rv = RvSipNotifyGetOutboundMsg(hNotify, &hNotifyMsg);
    if(rv != RV_OK)
    {
        printf("Failed to get notify outbound message");
        return rv;
    }
    /*Sets status information in the notify request message... */

    /*-------------------------
    Sends the NOTIFY message.
    -------------------------*/
    rv = RvSipNotifySend(hNotify);
    if(rv != RV_OK)
    {
        printf("Failed to send NOTIFY request");
        return rv;
    }
    printf("a notify message was sent for subscription %x/n", hSubs);
    return RV_OK;
}
/*===========================================================================*/
```

OUT-OF-BAND SUBSCRIPTION

According to SIP-events draft, sending a SUBSCRIBE request is just one way of establishing *subscriptions*. The application can also use other ways to establish *subscriptions* (such as HTTP), or the application can use other SIP *messages* to establish implicit *subscriptions*. In these cases, the application should create an “out-of-band” *subscription*.

An out-of-band *subscription* is created in an ACTIVE state, and can receive or send NOTIFY messages according to its type (subscriber or notifier). The application should create the *subscription* after the application agreed with remote party of the *subscription*.

Terminating an out-of-band *subscription* is done by sending and accepting a NOTIFY request with a “Subscription-State:terminated” header in it (the same as for a regular *subscription*).

An out-of-band *subscription* in a *call-leg* gets its dialog information from the *call-leg*. For an independent out-of-band *subscription*, the application should set all necessary dialog parameters (such as Call-ID, To header with tag, and From header with tag).

Note You must set the Call-ID parameter of an independent out-of-band-*subscription* before you call the *subscription* initialization function (RvSipSubsInit()).

The Out-of-Band API functions are listed below:

RvSipSubsMgrCreateOutOfBandSubscription()

Creates an out-of-band *subscription*. In this function you must specify whether the *subscription* belongs to a subscriber or a notifier.

NOTIFIER OUT-OF-BAND SUBSCRIPTION STATE MACHINE

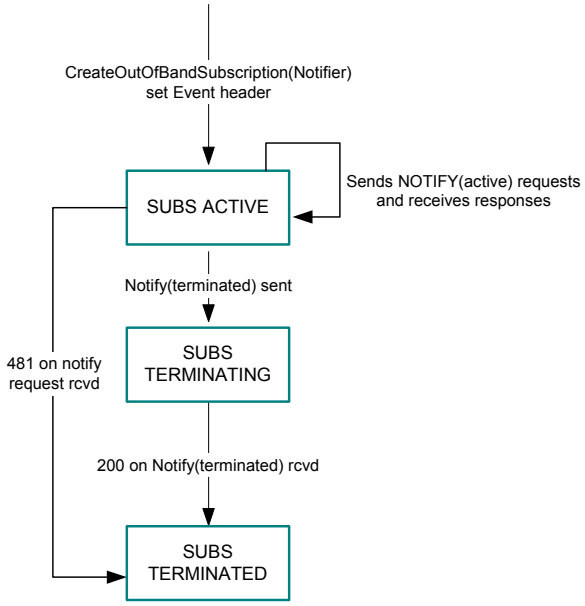


Figure 12-7 *Notifier Out-of-Band State Machine*

SUBSCRIBER OUT-OF-BAND SUBSCRIPTION STATE MACHINE

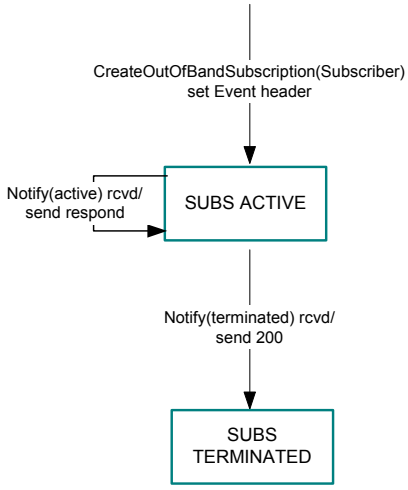


Figure 12-8 *Subscriber Out-of-Band State Machine*

Note The *subscription* expiration timer is not set to an out-of-band *subscription*. Therefore the RvSipSubsSubscriptionExpiredEv() and RvSipSubsExpirationAlertEv callbacks will not be called, and there is no need to set the Expires parameter in an out-of-band *subscription*, or in one of its NOTIFY requests.

Sample Code

The following sample code demonstrates an implementation of the out-of-band *subscription* creation procedure.

```
/*=========================================================================================*/
void AppCreateOutOfBandSubscription(
                RvSipSubsMgrHandle    hSubsMgr,
                RvSipCallLegHandle    hCallLeg,
                RvSipSubscriptionType eSubsType,
                RvChar*               strEvent)
{
    RvSipSubsHandle hSubs;
    RvStatus rv;
    RvSipEventHeaderHandle hEvent;

    /*-----------------------------------
    Creates an out-of-band subscription.
    -----------------------------------*/
    rv = RvSipSubsMgrCreateOutOfBandSubscription(hSubsMgr, hCallLeg, NULL, eSubsType, &hSubs);
    if(rv != RV_OK)
    {
        printf("RvSipSubsMgrCreateOutOfBandSubscription failed");
        return;
    }
    printf("An out-of-band subscription %x was created\n",hSubs);

    /*-----------------------------------------------------
    Creates an Event header and sets it in the subscription.
    -----------------------------------------------------*/
    rv = RvSipSubsGetNewHeaderHandle(hSubs, RVSIP_HEADERTYPE_EVENT, (void**)&hEvent);
    if(rv != RV_OK)
    {
        printf("RvSipSubsGetNewHeaderHandle failed ");
        return;
    }
    rv = RvSipEventHeaderParseValue(hEvent, strEvent);
    if(rv != RV_OK)
    {
        printf("RvSipEventHeaderParseValue failed ");
        return;
    }
    rv = RvSipSubsSetEventHeader(hSubs, hEvent);
    if(rv != RV_OK)
    {
```

```
        printf("RvSipSubsSetEventHeader failed ");
        return ;
    }
    /*-----------------------------------------------
    If notifier--creates and sends notification as usual.
    -----------------------------------------------*/
    if(eSubsType == RVSIP_SUBS_TYPE_NOTIFIER)
    {
        AppCreateNotification(hSubs, RVSIP_SUBSCRIPTION_SUBSTATE_ACTIVE);
        printf("Notify (active) request was sent");
    }
    return;
}
/*=============================================================================================*/
```

Support for Subscription Forking

According to the proxying rules in RFC 3261, a SUBSCRIBE request will receive only one 200-class response. The *subscription*, however, may have been accepted by multiple nodes due to forking. Therefore, the subscriber **must** be prepared to receive NOTIFY requests from multiple notifiers, with From tags that differ from the To tag received in the SUBSCRIBE 200 response.

The application should decide if the creation of multiple *subscriptions* from a single forked SUBSCRIBE is allowed. If so, the subscriber establishes new *subscriptions* by returning a 200-class response to each NOTIFY. Each subscription is then handled as its own entity, and is refreshed independent of the other *subscriptions*. See RFC 3265, sections 3.3.3 and 4.4.9 for multiple *subscription* handling.

Note that an original *subscription* is a *subscription* that sent the original SUBSCRIBE request. A forked *subscription* is a *subscription* that was created by an incoming NOTIFY request.

Working with Subscription Forking

A User Agent Client (UAC) sends a SUBSCRIBE request as usual. This creates a *subscriptions*, sets the dialog and *subscription* parameters, and sends the initial SUBSCRIBE request.

This *subscription* is identified by its Call-ID, From tag and Event header. The To tag value is empty until a response message is received from a UAS. If a NOTITY request arrives before the 200-class response, the *subscription* will inherit its To tag from the NOTIFY From header tag.

Handling Multiple Notify Requests

As a result of forking of the SUBSCRIBE request, the SIP Stack can receive one or more NOTIFY requests. Each NOTIFY matches the *subscription* and nearly matches the dialog. All identifiers are equal, but the From tag of the NOTIFY is not equal to To tag of the dialog. This means that another server is ready to serve the *subscription*.

As a result, the SIP Stack creates a forked *subscription*, based on the original *subscription*. The forked *subscription* inherits the majority of its parameters from the original *subscription*. The forked *subscription* is independent of the original *subscription*, and behaves as a regular *subscription*.

Figure 12-9 illustrates the message flow of *subscription* forking.

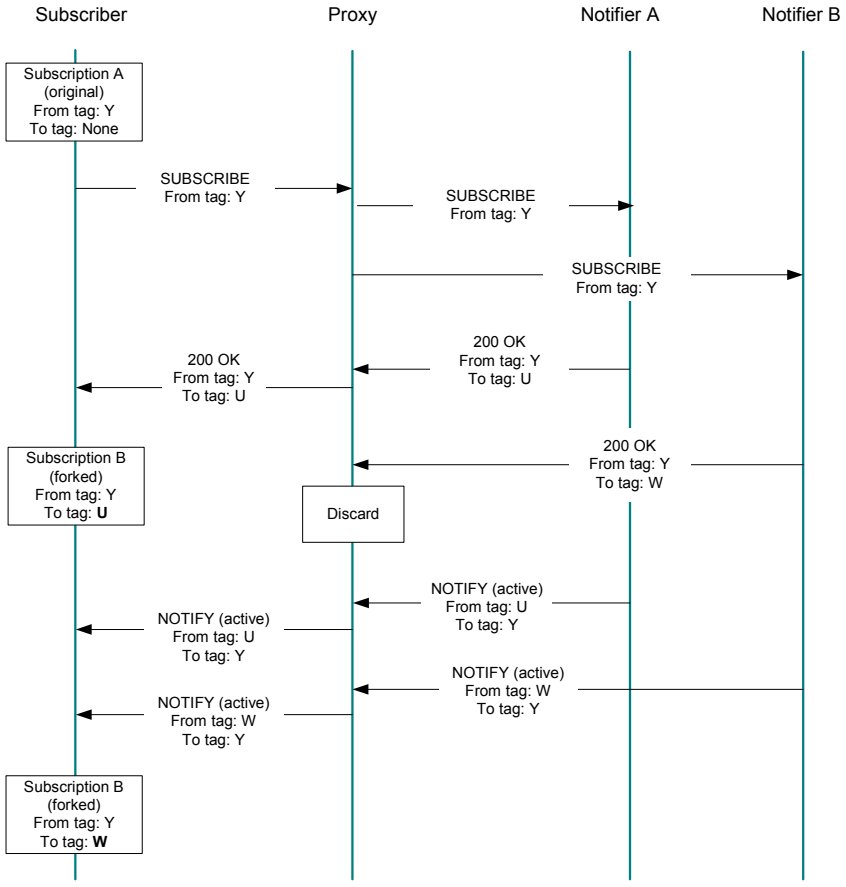


Figure 12-9 *Subscription Forking Message Flow*

Subscription Forking Events

The Subscription API includes the following event for forking-support implementation.

RvSipSubsCreatedDueToForkingEv()

A NOTIFY request that is received as a result of SUBSCRIBE sending may create a forked *subscription*. This event informs the user of the creation of a new forked *subscription*, and exchanges handle with the application. If the user does not wish to handle this forked *subscription*, the user can indicate that the SIP Stack should terminate the new forked *subscription*. In this case, the new (forked) *subscription* will be destructed immediately. Otherwise, the new forked *subscription* will handle the NOTIFY request, update its state machine, send a 200 response to the request, and call to all the regular *subscription* callback functions.

Subscription Forking API

The Subscription API includes the following set of functions dedicated to forking-support implementation.

RvSipSubsGetOriginalSubs()

Returns the handle to the original *subscription* that is related to a given forked *subscription*. If the given *subscription* is an original *subscription*, NULL will be returned.

RvSipSubsSetForkingEnabledFlag(), RvSipSubsGetForkingEnabledFlag()

Functions for setting/getting the *subscription* forking-enabled flag. The forking-enabled-flag defines the *subscription* behavior on receiving multiple NOTIFY request due to proxy forking of the initial SUBSCRIBE request.

If this flag is set to TRUE in the original *subscription*, a new forked *subscription* will be created for every NOTIFY request. If this flag is set to FALSE in the original *subscription*, only completely matched NOTIFY requests will be handled by the *subscription*. Other NOTIFY requests will be rejected with a 481 response code. The default value for the forking-enabled flag is RV_TRUE.

Subscription Forking State Machine

The Forked Subscription state machine is identical to the Subscription state machine in Figure 12-3 with one exception—a forked *subscription* moves from the IDLE state directly to the 2XX_RCVD state immediately after the RvSipSubsCreatedDueToForkingEv() callback is called. Since a forked *subscription* did not initiate the SUBSCRIBE request, it will never receive a 2XX response for it and therefore will never assume the SUBS_SENT, REDIRECTED and NOTIFY_BEFORE_2XX_RCVD states.

SUBSCRIPTION FORKING CALL FLOW

The following call flows illustrate three types of *subscription* forking.

Figure 12-10 illustrates *subscription* forking that is enabled in the SIP Stack and in the original *subscription*.

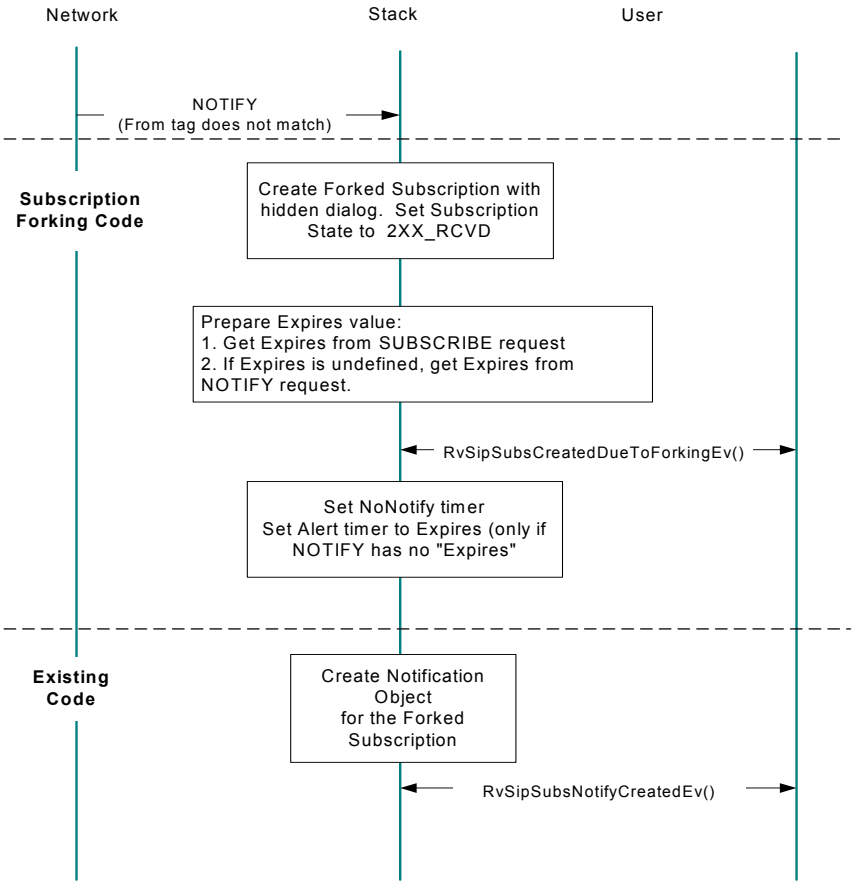


Figure 12-10 *Subscription Forking in SIP Stack and Original Subscription*

Figure 12-11 illustrates *subscription* forking that is disabled in the SIP Stack.

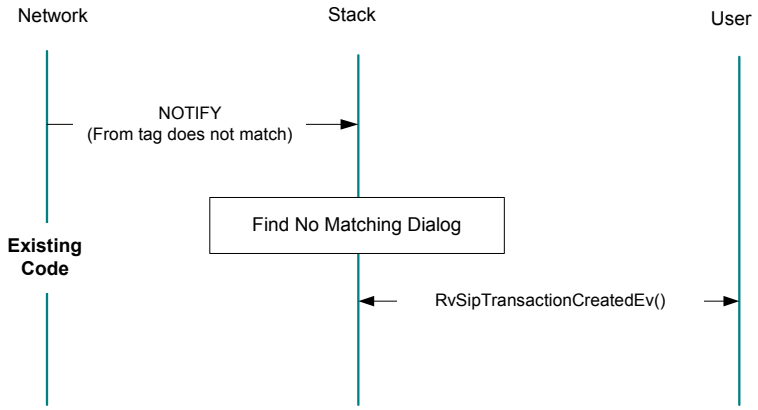


Figure 12-11 *Subscription Forking Disabled in SIP Stack*

Figure 12-12 illustrates *subscription* forking that is enabled in the SIP Stack and disabled in the original *subscription*.

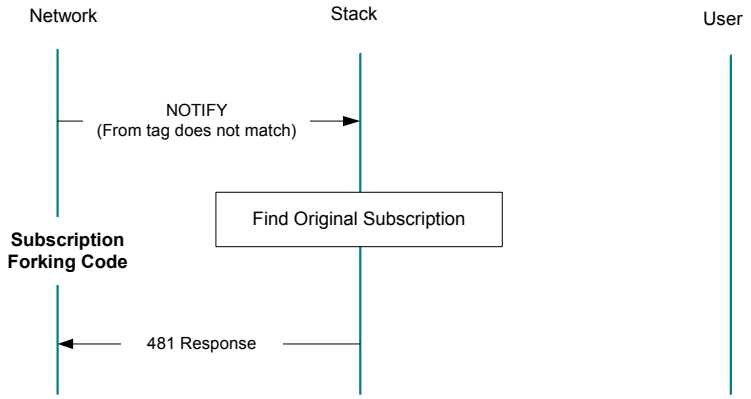


Figure 12-12 *Subscription Forking Enabled in SIP Stack and Disabled in Original Subscription*

Subscription Forking Configuration

The *subscription* forking configuration parameters are as follows:

bEnableSubsForking

Enables the *subscription* forking feature in the SIP Stack. If this parameter is set to RV_FALSE, all NOTIFY requests that do not exactly match an existing *subscription* will be handled as general *transactions*. No forked *subscriptions* will be created. If this parameter is set to RV_TRUE, a new forked *subscription* will be created for each NOTIFY request as explained above. The default value for this parameter is RV_FALSE.

Authentication and DNS

For information on *subscription* authentication, see the Authentication chapter. For information on *subscription* DNS see the Working with DNS chapter.

REFER

For information on using the *subscription* for REFER implementation, see the the REFER chapter.

13

REFER

INTRODUCTION

REFER is a SIP method defined by RFC 3515, “The Session Initiation Protocol (SIP) Refer Method”. The REFER method indicates that the recipient of the REFER request should contact a third party using the contact information provided in the REFER request. RFC 3515 provides a mechanism allowing the party that is sending the REFER to be notified of the outcome of the referenced request. Once it is known whether the reference succeeded or failed, the agent receiving the REFER request notifies the agent that sent the REFER request about this result using the NOTIFY request.

The REFER mechanism utilizes the Subscribe-Notify mechanism and therefore all REFER functionality is handled by the Subscription layer of the SIP Stack. It is recommended to read the Event Notification chapter in this guide before reading this chapter.

REFER-SUBSCRIPTION

According to RFC 3515, a REFER request implicitly establishes a *subscription* to the refer event. The SIP Stack handles both incoming and outgoing REFER requests using the *subscription*. A *subscription* that handles a REFER request is called a *refer-subscription*.

The REFER method can be used in the following two ways, which are handled by the REFER Subscription API:

- A REFER sent within the scope of an existing dialog—UA1 that is in a session with UA2 can create a *refer-subscription* within this session, and send the REFER to UA2 to establish a session with UA3. This can be done to achieve call transfer. You can send the REFER on both early and confirmed dialogs.

- A REFER sent outside the context of a dialog—UA1 can create an independent *refer-subscription*, and send the REFER to UA2 to establish a session with UA3, without connecting a call with UA2 first.

REFER-SUBSCRIPTION API

The REFER-Subscription API includes a set of functions and function callbacks that are dedicated to REFER implementation. These functions are called the REFER-Subscription API. To implement REFER scenarios, you should use the REFER-Subscription API together with the generic API functions of the Subscription layer.

The REFER-Subscription API functions enable you to examine REFER parameters, send, receive and accept REFER requests, and handle the NOTIFY messages associated with the REFER request.

REFER-SUBSCRIPTION PARAMETERS

You can examine *refer-subscription* parameters via Subscription Get API functions. The following parameters are available:

Refer To Header

The Refer-To header contains the address of the referenced party in a REFER process. This address is updated from REFER requests that are sent or received by the SIP Stack. You can set the Refer-To header to an outgoing REFER request using the RvSipSubsReferInit() function. You can access the Refer-To header of an incoming REFER request with a Get function. The header is not modifiable.

Referred By Header

The Referred-By header contains the address of the party initiating the REFER process (the referrer). This address is updated from REFER requests that are sent or received by the SIP Stack. You can set the Referred-By header to an outgoing REFER request using the RvSipSubsReferInit() function. You can access the Referred-By header of an incoming REFER request with a Get function. The header is not modifiable.

REFER-SUBSCRIPTION CONTROL

The following API functions were added to the Subscription layer for REFER control.

RvSipSubsReferInit()

Initializes a *subscription* with the Refer-To and Referred-By headers. To send a REFER Request, you must first create a *subscription* using the RvSipSubsMgrCreateSubscription() function. If the *subscription* is not in the context of an existing dialog, you must also initialize the To and From headers of the *subscription* and optionally initialize its contact addresses using the RvSipSubsDialogInit() function. Only then is the *subscription* ready for REFER initialization. By calling the RvSipSubsReferInit() function, you can set both Refer-To and Referred-By headers to the *refer-subscription*. The Referred-By header is optional and you can supply NULL if you do not wish to use it. Both headers will be copied to the outgoing REFER request.

RvSipSubsReferInitStr()

Has the same functionality as the RvSipSubsReferInit() function. This function initializes a *subscription* with the Refer-To and Referred-By headers. However, when calling RvSipSubsReferInitStr(), you supply the Refer-To and the Referred-By headers in string format. You may also supply a Replaces header string that the SIP Stack will set in the Refer-To header.

RvSipSubsRefer()

Generates and sends a REFER message. After creating a *subscription* and initializing it with the necessary information, including the Refer-To header and optionally the Referred-By header, you can call the RvSipSubsRefer() function. Calling this function will cause a REFER request to be sent to the remote party. The REFER request will include the Refer-To and Referred-by headers that you set to the *refer-subscription*. After sending the REFER request, the *subscription* will assume the RVSIP_SUBS_STATE_SUBS_SENT state.

RvSipSubsReferAccept()

Generates and sends a 202 response to an incoming REFER request. When a REFER message is received, a new *refer-subscription* is created and the RvSipSubsCreatedEv() callback is called. The *refer-subscription* then changes its state to RVSIP_SUBS_STATE_SUBS_RCVD with the RVSIP_SUBS_REASON_REFER_RCVD reason. At this point, the application can call the RvSipSubsReferAccept() function and accept the REFER request.

Calling RvSipSubsReferAccept() causes a 202 response to be sent to the remote party. This function also returns a handle to a newly created object that will be used to contact the referenced party. In a typical situation, a *call-leg* will be

created. The newly created *call-leg* is initialized and ready to contact the referenced party. To issue the INVITE request to the referenced party, you should call RvSipCallLegConnect().

Note The Refer-To address in the incoming REFER request can include a method parameter that will indicate that the referenced party should be contacted with a method other than INVITE. According to the method, the newly created object can be a *call-leg*, *subscription* or *transaction*. More details of such cases are illustrated in REFER Request with Different Method Parameters.

RvSipSubsGetReferNotifyStatus()

Returns the status code that was received in the body of the NOTIFY request of a *refer-subscription*. The NOTIFY message is used to inform the agent sending the REFER about the status of the reference. The body of a NOTIFY begins with a SIP response status line. The response class in this status line indicates the status of the reference attempt. To receive this status without parsing the message body, you should call the RvSipSubsGetReferNotifyStatus() function.

REFER-Subscription Notify Control

The NOTIFY mechanism defined in RFC 3265 is used to inform the agent sending the REFER about the status of the reference attempt. The following API function was added to the Subscription layer for REFER Notify control:

RvSipNotifySetReferNotifyBody()

Builds the body of a NOTIFY request of a *refer-subscription*. Whenever there is a need to send a NOTIFY request, the stack calls the RvSipSubsReferNotifyReadyEv() callback. The application should then create a Notify object using the RvSipSubsCreateNotify() function and set the value of the Subscription-State header using the RvSipNotifySetSubscriptionStateParams() function. When a Notify is sent for a *refer-subscription*, the notify body should indicate the status of the reference attempt. To set such a body the application should call the RvSipNotifySetReferNotifyBody() function with the correct status. The SIP Stack will then build the Notify body according to RFC 3515 and set the correct body and Content-Type header in the NOTIFY request message.

Note The RvSipSubsReferNotifyReadyEv() callback will supply you with the correct parameters to set in the message body.

REFER-Subscription Events

The following callback function was added to the Subscription layer for REFER control:

RvSipSubsReferNotifyReadyEv()

This event informs the application that it should send a NOTIFY request on a *refer-subscription*. The callback is called in the following three stages:

- Immediately after accepting a REFER request. At this point the application should send the initial NOTIFY request.
- After a provisional response was received from the referenced party. At this point the application should send a NOTIFY request indicating that a provisional response was received.
- After a final response was received from the referenced party. At this point the application should send a NOTIFY that includes the final status of the reference attempt. If, however, the object that was used to contact the referenced party was terminated with an error before a final response was received, the callback will also be called and the application should send a NOTIFY that indicates this failure.

This callback function supplies you with a reason that indicates at which stage the *subscription* currently is, and the status code that should be set in the body of the NOTIFY request.

When this callback is called, the application should send a NOTIFY request using the following steps:

1. Create a *notification* in the *refer-subscription*, using RvSipSubsCreateNotify().
2. Set the Subscription-State header in the NOTIFY request, using RvSipNotifySetSubscriptionStateParams(). In the final notify request (stage 3), the Subscription-State header value should be “terminated”. Elsewhere it should be “active”.
3. Set the correct body in the NOTIFY request, using RvSipNotifySetReferNotifyBody(). You should supply this function the status code supplied by this callback function.
4. Use the Notify object to send the NOTIFY request using RvSipNotifySend().

REFER Complete Process Flow

Figure 13-1 illustrates a complete REFER scenario in an established call.

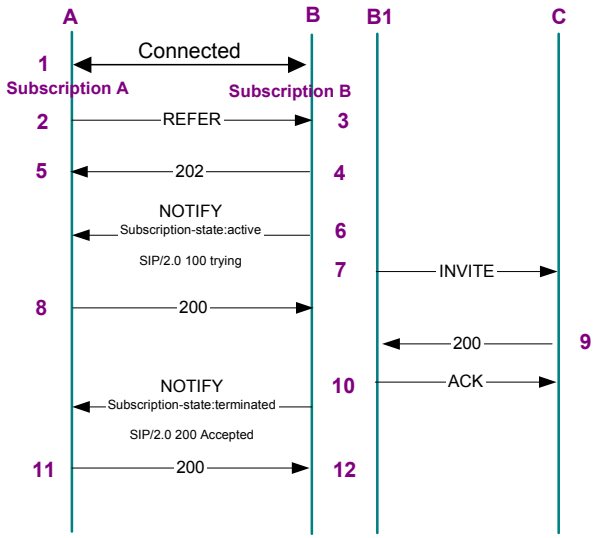


Figure 13-1 *REFER Process Call Flow*

The following four *call-legs* are involved in the complete REFER scenario:

- *Call-leg* A—the *call-leg* that sends the REFER request. This *call-leg* sends the REFER using *Refer-subscription* A that is created in the context of *Call-leg* A.
- *Call-leg* B—the *call-leg* that receives the REFER request. The incoming REFER is handled by *Refer-subscription* B that is created in the context of *Call-leg* B.
- *Call-leg*—a new *call-leg* created to contact the REFER target.
- *Call-leg* C—the referenced party *call-leg*.

The steps for a complete REFER process on an established call are as follows.

1. *Call-leg*s A and B are connected.
2. To send a REFR from *Call-leg* A to *Call-leg* B, the application creates *subscription* A by calling the RvSipSubsMgrCreateSubscription() function. It initializes the *subscription* with REFER parameters using the

RvSipSubsReferInit() function, and sends the REFER message by calling the RvSipSubsRefer() function. After the REFER message is sent, *subscription* A moves to the SUBS_SENT state.

3. *Call-leg* B receives the REFER request and creates a new *Refer-subscription* B to handle this REFER. The new *subscription* assumes the SUBS_RCVD state.
4. The application decides to accept the REFER and calls the RvSipSubsReferAccept() function on *Subscription* B. A 202 response is sent back to *Subscription* A and a new Call-leg B1 is created. B1 is initialized and ready to contact the referenced party. *Subscription* B assumes the SUBS_ACTIVE state.
5. *Subscription* A receives the 202 response and assumes the SUBS_2XX_RCVD state.
6. The application sends an initial NOTIFY request from *Subscription* B. It first creates a *notification* with the RvSipSubsCreateNotify() function and then follows the following steps to initialize and sent the request:
   - Sets the Subscription-State header to “active” using the RvSipNotifySetSubscriptionStateParams() function.
   - Sets the “SIP\2.0 100 trying” body in the NOTIFY message using the RvSipNotifySetReferNotifyBody() function.
   - Sends the NOTIFY request from the new *notification* by calling the RvSipNotifySend() function.
7. Calling the Connect() function on *Call-leg* B1 causes an INVITE message to be sent to the referenced party.
8. *Subscription* A receives the NOTIFY request and calls the RvSipSubsNotifyEv() callback to inform the application. The application calls the RvSipNotifyAccept() function that causes a 200 response to be sent for the NOTIFY request. *Subscription* A then assumes the SUBS_ACTIVE state.
9. A 200 response is sent from *Call-leg* C.
10. *Call-leg* B1 receives the 200 response. It informs *Subscription* B that it has reached a final status. *Subscription* B calls the RvSipSubsReferNotifyReadyEv() callback to inform the application that it is now time to send the final NOTIFY

request. Using the steps described in step 6, the application builds and sends a NOTIFY request to inform the REFER initiator about the result of the reference attempt. This NOTIFY has a "terminated" value in the Subscription-State header.

11. *Subscription* A receives the NOTIFY request and informs the application by calling the RvSipSubsNotifyEv() callback. The application accepts this NOTIFY. Since this is a final NOTIFY with a "terminated" value, accepting it also terminates *Subscription* A. The REFER procedure is completed for *Call-leg* A.
12. *Subscription* B receives the 200 response on the final NOTIFY. It notifies the application of the response with the RvSipSubsNotifyEv() callback. The 200 response on the final NOTIFY request terminates *Subscription* B. The REFER procedure is completed for *Call-leg* B.

    At this point, *Call-leg* A is connected to *Call-leg* B, and *Call-leg* B1 is connected to *Call-leg* C. It is up to the application to decide whether to disconnect the call between *Call-leg* A and *Call-leg* B.

REFER Request with Different Method Parameters

The URI in the Refer-To header can include a method parameter that indicates how to contact the referenced party.

Example 1

The `Refer-To:<sip:C@xxx.com;method=INVITE>` header indicates that the referee should contact the REFER target with an INVITE request. (If there is no method parameter, the case is handled in the same way as if method=INVITE exists.)

Example 2

The "`Refer-To:<sip:C@xxx.com;method=SUBSCRIBE?event=MMM&expires=900>`" header indicates that the referee should contact the refer target with a SUBSCRIBE request, that will include the "`event:MMM`" and "`Expires:900`" headers.

Example 3

The "`Refer-To:<sip:C@xxx.com;method=REFER?Refer-To=C%3Csip:c%40yyy.com%3E>`" header identifies that the referee should contact the refer target with a REFER request which will include the "`Refer-To:C<sip:c@yyy.com>`" header.

Example 4

The "`Refer-To:<sip:C@xxx.com;method=OPTIONS?apple=red>`" header identifies that the referee should contact the REFER target with an OPTIONS request which will include the "`apple:red`" header.

The SIP Stack supports all methods given in the Refer-To header. The RvSipSubsReferAccept() function creates a new object that will be used to contact the REFER target. The type of the new allocated object is determined according to the following method parameters:

- INVITE method (or no method at all) will create a new *call-leg*.
- SUBSCRIBE and REFER methods will create a new *subscription*.
- All other methods will create a new *transaction*.

The SIP Stack automatically initializes the new object with the necessary information, such as the To and From headers, and the header list taken from the Refer-To header. Applications that wish to support only specific methods should check the method parameter before accepting the REFER request. The application can also force the SIP Stack to create a specific object regardless of the method parameter. (For example, create a new *transaction*, and not a *call-leg* or *subscription* for the INVITE, REFER and SUBSCRIBE methods). For more details, see the RvSipSubsReferAccept() function in the *SIP Stack Reference Guide*.

Note When the newly created object is a transaction, the RvSipSubsReferNotifyReadyEv() callback will not be called, and the application is responsible for sending the NOTIFY request once the *transaction* has reached a final status.

Implementing REFER-related Application Callbacks

The following sample codes demonstrate the implementation of two callbacks that are part of the REFER process.

RvSipSubsStateChangedEv() Callback Implementation

The following code is an implementation of the Subscription-state changed callback. The sample demonstrates how to respond to an incoming REFER request with 202, and then connect the new call with the Refer-to party. In this sample, if the method parameter in the Refer-To URI indicates a method different than INVITE, the REFER is rejected with a 400 response.

Sample Code

```
/*===========================================================================================*/
void RVCALLCONV AppSubsStateChangedEvHandler(
                                    IN  RvSipSubsHandle            hSubs,
                                    IN  RvSipAppSubsHandle         hAppSubs,
                                    IN  RvSipSubsState             eState,
                                    IN  RvSipSubsStateChangeReason eReason)
{
    RvStatus                    rv;
    RvSipCommonStackObjectType eObjType = RVSIP_COMMON_STACK_OBJECT_TYPE_UNDEFINED;
    RvSipCallLegHandle          hCallLeg;
    RvSipReferToHeaderHandle    hReferTo;
    RvSipAddressHandle          hReferToAddr;
    RvSipMethodType             eMethod;

    if(RVSIP_SUBS_STATE_SUBS_RCVD != eState)
    {
        /* Handle only the subs rcvd state*/
        return;
    }
    if(RVSIP_SUBS_REASON_REFER_RCVD == eReason ||
       RVSIP_SUBS_REASON_REFER_RCVD_WITH_REPLACES == eReason)
    {
        /* check that the refer request is for a new call-leg object */
        RvSipSubsGetReferToHeader(hSubs, &hReferTo);
        hReferToAddr = RvSipReferToHeaderGetAddrSpec(hReferTo);
        eMethod = RvSipAddrUrlGetMethod(hReferToAddr);
        if(eMethod != RVSIP_METHOD_UNDEFINED &&
           eMethod != RVSIP_METHOD_INVITE)
        {
            printf("Refer has a method parameter different than INVITE. Reject it\n");
            RvSipSubsRespondReject(hSubs, 400, "Unsupported Method In Refer-To header");
            return;
```

```
        }

        /* Accept incoming refer */
        rv = RvSipSubsReferAccept(hSubs, NULL, eObjType, &eObjType, (void**)&hCallLeg);
        if(rv != RV_OK)
        {
            printf("Failed to accept the refer request\n");
            return;
        }

        /* Connect the new call-leg, to the refer-to party */
        rv = RvSipCallLegConnect(hCallLeg);
        if(rv != RV_OK)
        {
            printf("Failed to connect the new call\n");
            return;
        }
    }
}
      /*-----------------------------------------------------------------------------*/
```

RvSipSubsReferNotifyReadyEv() Callback Implementation

The following sample code is an implementation of the Notify-Ready event callback. The sample demonstrates how to create, initialize and send a NOTIFY request to the REFER initiator. This sample also checks the supplied reason, and sets the NOTIFY parameters accordingly.

For example, if the reason is 1XX_RESPONSE_MSG_RCVD, the NOTIFY request should include the following Subscription-State header: "`Subscription-State: active;expires=50`" and a message body that includes the response code.

Sample Code

```
/*-----------------------------------------------------------------------------*/
          void RVCALLCONV AppSubsReferNotifyReadyEv(
                           IN  RvSipSubsHandle                 hSubs,
                           IN  RvSipAppSubsHandle              hAppSubs,
                           IN  RvSipSubsReferNotifyReadyReason eReason,
                           IN  RvInt16                         responseCode,
                           IN  RvSipMsgHandle                  hResponseMsg)
{
```

```
RvSipSubscriptionSubstate  eSubsState;
RvSipSubscriptionReason    eNotifyReason;
RvInt32                    expires = UNDEFINED;
RvInt16                    notifyResponseCode;
RvStatus                   rv = RV_OK;
RvSipNotifyHandle          hNotify;

switch (eReason)
{
case RVSIP_SUBS_REFER_NOTIFY_READY_REASON_1XX_RESPONSE_MSG_RCVD:
case RVSIP_SUBS_REFER_NOTIFY_READY_REASON_INITIAL_NOTIFY:
    notifyResponseCode = responseCode;
    eSubsState         = RVSIP_SUBSCRIPTION_SUBSTATE_ACTIVE;
    expires            = 50;
    break;
case RVSIP_SUBS_REFER_NOTIFY_READY_REASON_FINAL_RESPONSE_MSG_RCVD:
    notifyResponseCode = responseCode;
    eSubsState         = RVSIP_SUBSCRIPTION_SUBSTATE_TERMINATED;
    eNotifyReason      = RVSIP_SUBSCRIPTION_REASON_NORESOURCE;
    break;
case RVSIP_SUBS_REFER_NOTIFY_READY_REASON_TIMEOUT:
case RVSIP_SUBS_REFER_NOTIFY_READY_REASON_ERROR_TERMINATION:
    notifyResponseCode = 503;
    eSubsState         = RVSIP_SUBSCRIPTION_SUBSTATE_TERMINATED;
    eNotifyReason      = RVSIP_SUBSCRIPTION_REASON_NORESOURCE;
    break;
case RVSIP_SUBS_REFER_NOTIFY_READY_REASON_UNDEFINED:
default:
    return;
}

/* create the notification object */
rv = RvSipSubsCreateNotify(hSubs, NULL, &hNotify);
if (RV_OK != rv)
{
    printf("Failed to create notification");
    return;
}
/*initialize the notification object*/
rv = RvSipNotifySetSubscriptionStateParams(hNotify,eSubsState, eReason, expiresParamVal);
if (RV_OK != rv)
```

```
    {
        printf("Failed to set notify parameters");
        return;
    }
    rv = RvSipNotifySetReferNotifyBody(hNotify, statusCode);
    if (RV_OK != rv)
    {
        printf("Failed to set notify body");
        return;
    }

    /*send the notify request*/
    rv = RvSipNotifySend(hNotify);
    if (RV_OK != rv)
    {
        printf("Failed to send notify ");
        return;
    }
}
/*===========================================================================================*/
```

Sending a REFER Request

The following sample code demonstrates how to send a REFER request. In this sample, a REFER is sent outside the context of a dialog. Therefore the *subscription* dialog must be initiated as well.

Sample Code

```
*===========================================================================================*/
static void AppSubsRefer(RvSipSubsMgrHandle hSubsMgr)
{

    RvChar* From       = "sip:A@127.0.0.1";
    RvChar* To         = "sip:B@127.0.0.1";
    RvChar* ReferTo    = "sip:C@127.0.0.1";
    RvChar* ReferredBy = "<sip:referrer@referrer.example>";
    RvSipSubsHandle hReferSubs;
    RvStatus        rv;

    /*--------------------------
      Create a new subscription
```

```
    ----------------------------*/
    rv = RvSipSubsMgrCreateSubscription(hSubsMgr, NULL, NULL, &hReferSubs);
    if(rv != RV_OK)
    {
        printf("Failed to create new refer subscription");
        return;
    }

    /* Initiate the subscription dialog parameters */
    rv = RvSipSubsDialogInitStr(hReferSubs, From, To, NULL, NULL);
    if(rv != RV_OK)
    {
        printf("subscription dialog initialization failed.");
        return;
    }
    /* Initiate the REFER Subscription parameters */
    rv = RvSipSubsReferInitStr(hReferSubs, ReferTo, ReferredBy, NULL);
    if(rv != RV_OK)
    {
        printf("REFER Subscription initialization failed.");
        return;
    }

    /* Send the REFER request */
    rv = RvSipSubsRefer(hReferSubs);
    if(rv != RV_OK)
    {
        printf("referring failed.");
        return;
    }
}

*=============================================================================================*/
```

14

WORKING WITH THE TRANSPORT LAYER

INTRODUCTION

The Transport layer is responsible for the actual transmission of requests and responses over network transports. SIP permits the usage of unreliable transports, such as UDP, and reliable transports, such as TCP and TLS (over TCP), using both IPv4 and IPv6 addresses. TCP, TLS and UDP differ in many ways. The most fundamental difference is that UDP is connection-less, while TCP and TLS are connection-oriented and therefore create a reliable data transfer service.

This chapter focuses on the connection-oriented reliable transport types. The SIP Stack supports TCP and TLS connection-oriented transports. This chapter explains how to maintain a persistent connection and how to use the TCP and TLS transports with the SIP Stack API. In addition, this chapter describes the abilities that the SIP Stack provides for monitoring row buffers that are sent or received from the sockets, and the usage of IPv6 addresses.

PERSISTENT CONNECTION HANDLING

In many cases, a single *connection* (TCP or TLS) may be reused for different *messages*, *transactions* or dialogs. Opening and closing TCP *connections* often is not desirable because of the extra messaging overhead of the TCP handshake and even more so in TLS *connections*. Therefore, RFC 3261 permits connection persistency, which is the reuse of an open client *connection*. The Persistent Connection feature of the SIP Stack adds the capability of identifying that a message can be sent on an existing open *connection*. The message is then sent using this *connection*.

Connection Hash

To reuse open *connection*s, client *connections* (*connections* opened by the UAC) are inserted into a hash table. The *connection* hash key consists of the following elements:

- The *connection* transport (TCP or TLS)
- The *connection* local address (IP and port)
- The *connection* remote address (IP and port)

Whenever a SIP Stack object needs to send a message using a reliable transport (TCP or TLS), it first queries the *connection* hash for existing opened *connections.* If such a *connection* exists, it will be used for sending the message.

Connection Owner

Each *connection* holds a list of owners. A *connection* owner is an element that needs to use the *connection* and therefore asks to be one of the owners of the *connection*. The *connection* notifies each of its owners about *connection* events.

If, for example, a *transaction* wishes to send a message using TCP transport from address X to address Y, the *transaction* first queries the *connection* hash for an existing suitable *connection*. If such a *connection* exists, the *transaction* will attach itself to the *connection* and will become one of the *connection* owners. The *transaction* can then use the *connection* for sending messages. When the *transaction* no longer wishes to use this *connection*, it should detach from the *connection*. After detaching from the *connection*, the *transaction* is no longer allowed to use the *connection* and it will not be informed of *connection* events. In the case where there is no suitable *connection* in the hash, a new *connection* will be constructed. This new *connection* will be inserted into the hash and other elements will be able to use it as well.

Note An element must not use a *connection* unless this element is a *connection* owner.

Connection Termination Rule

A basic rule of the persistent *connection* feature is that a *connection* will not terminate as long as it has owners. A SIP Stack element that uses a *connection* never specifically terminates it. An element that is no longer using a *connection* needs to detach from the *connection*, by removing itself from the *connection* Owners List. The *connection* will disconnect automatically when the Owners List is empty.

Connection Capacity Percent

RFC 3261 recommends that *connections* should be kept open for a period of time after the last message was exchanged over the *connection*. However, the exact time period to leave the *connection* open is defined by the implementation. For example a fixed (location-wise) UA using a default outbound proxy can leave the *connection* open forever, or until the proxy closes it.

The SIP Stack enables the application to leave client *connections* open even after the *connections* are no longer in use (meaning that they do not have owners). To achieve this, the application can use the *connectionCapacityPercent* configuration parameter. The *connectionCapacityPercent* configuration parameter determines the recommended percentage of opened *connections* that the SIP Stack is allowed to hold at any given moment from its pool of *connections*.

The SIP Stack is allowed to exceed the percentage of opened *connections* only if all the opened *connections* are actually in use and have owners. When the connectionCapacityPercent parameter is greater than 0, the SIP Stack will not close *connections* that are no longer in use. Such *connections* will be kept in a separate list and will remain open as long as their resources are not required.

Once the percentage of opened *connections* exceeds the allowed connectionCapacityPercent, the SIP Stack will start closing *connections* from this list each time it is required to open a new *connection*. A *connection* that is kept in this list is not removed from the *connection* hash. Such a *connection* can be located in the hash for usage and, in this case, will be removed from the list and will no longer be a candidate for closure.

Since it is recommended to avoid resource problems for the creation of new objects, and since *connection* closure is a process that takes time, it is highly recommended to set the connectionCapacityPercent to an optimal value, smaller than one hundred percent. This way the SIP Stack will always have available resources for the creation of new *connections*.

Application Connections

The Persistent Connection feature lets the application open an application *connection* and become the *connection* owner. Application *connections* are handled as any other *connections*. They are inserted into the hash and can be used by any SIP Stack object. As long as the application does not detach from an application *connection*, the *connection* will have at least one owner (which is the application) and therefore this *connection* will not be disconnected. Applications might find it useful to open such *connections* to outbound proxies.

Persistency Levels

The persistent *connection* feature of the SIP Stack offers three modes of operation defined in the RvSipTransportPersistencyLevel enumeration:

- Undefined persistency (RVSIP_TRANSPORT_PERSISTENCY_LEVEL_UNDEFINED)
- Transaction persistency (RVSIP_TRANSPORT_PERSISTENCY_LEVEL_TRANSC)
- Transaction user persistency (RVSIP_TRANSPORT_PERSISTENCY_LEVEL_TRANSC_USER)

Undefined Persistency

When the SIP Stack is configured to use an undefined persistency level, the following two rules apply:

- SIP Stack objects do not look for suitable *connections* in the hash before sending a message, and therefore always open new *connections* for sending requests. (Responses are still sent on the *connection* on which the request was received).
- Newly created *connections* are not inserted into the hash.

The above two rules actually mean that each request, and its response sent by any of the SIP Stack objects, use a different new *connection*. ACK on a non-2xx response is sent on the same *connection* as the INVITE. (ACK on a 2xx response is always sent on a new *connection* since it is not part of the INVITE transaction.) In any case, there is no *connection* reuse in the SIP Stack.

Transaction Persistency

When the SIP Stack is configured to use the *transaction* persistency level, the following three rules apply:

- A *transaction* or a *transmitter* that wishes to send a request will first try to locate a suitable *connection* in the hash.
- If there is a suitable open *connection*, the *transaction* will use it. If there is not, the *transaction* will open a new *connection* and insert it into the *connections* hash. In both cases, the *transaction* will attach itself to the *connection* and become the *connection* owner.
- The *transaction* will detach from the *connection* only before the *transaction* terminates.

The above three rules imply that at the *transaction* persistency level, SIP Stack *transactions* that send messages to the same destination will share the same *connection*. The *transaction* can belong to different SIP Stack objects, such as a *call-leg*, *register-client* or subscription. As long as a *connection* in the hash fits, the *transaction* will use it. The *connection* will disconnect only when the last *transaction* detaches from it.

Since a *transaction* detaches from a *connection* only before termination, an INVITE message and its ACK sharing the same addresses will be sent on the same *connection* (when the response is not 2xx).

Transaction User Persistency

A Transaction User (TU) is an object that uses *transactions* for sending requests. A *call-leg*, a subscription and a *register-client* are all Transaction Users. When the SIP Stack is configured to use the Transaction User persistency level, the following two rules apply:

- SIP Stack *transactions* behave as defined in the *transaction* persistency level.
- A TU tries to use the same *connection* for all outgoing requests (sent by different client *transactions*).

A TU that sends a request queries the client *transaction* for the *connection* that was used. The TU then attaches to this *connection* and becomes the *connection* owner. At this stage, both the TU and the *transaction* itself are the *connection* owners. This guarantees that the *connection* will not disconnect when the *transaction* terminates and detaches from it. When trying to send the next request, the TU constructs a new client *transaction* and supplies the *transaction* with the *connection*. If this *connection* fits, the *transaction* will use it. If not, a new *connection* will be created and the TU will detach from the previous *connection* and attach to the new one. It will then use this new *connection* for further requests in the same matter.

Persistency Level Configuration

The persistency level of the SIP Stack objects is determined upon initialization. However, you can change the persistency level of specific objects using a dedicated API.

By default, the SIP Stack is initialized to use the “Undefined persistency” level. To change the persistency level, you should set the ePersistencyLevel parameter of the RvSipStackCfg configuration structure.

The following lines of code demonstrates how to initialize the SIP Stack with a “Transaction persistency” level:

```
/*=========================================================================*/
RvSipStackHandle hStack;
RvSipStackCfg stackCfg;
RvSipStackInitCfg(sizeof(stackCfg),&stackCfg);

stackCfg.tcpEnabled = RV_TRUE;
stackCfg.ePersistencyLevel = RVSIP_TRANSPORT_PERSISTENCY_LEVEL_TRANSC;

RvSipStackConstruct(sizeof(stackCfg),&stackCfg,&hStackMgr);
```

```
/*=========================================================================*/
```

SIP Stack Object Persistency APIs and Events

Each of the SIP Stack objects—*call-legs*, subscriptions, *register-clients*, *transactions* and transmitters—offers several API functions to query and change the persistency level of the object, and to query and determine the *connection* that the object will use. In the following explanations, XXX represents each of the SIP Stack object types.

RvSipXXXSetPersistency()

You can use the RvSipXXXSetPersistency() API function to change the persistency definition of each of the SIP Stack objects at runtime. This function receives a Boolean value that indicates whether or not the application wishes the specific object to be persistent. The persistency behavior of each object depends on the object type. Persistent TU objects will apply their persistency to the *transactions* they create.

The following API functions are supplied:

RvSipCallLegSetPersistency()

Sets the persistency level of *call-legs* and *subscriptions*.

Note To set the persistency level of a *subscriptions*, you must first get the Subscription-associated *call-leg* and then use the Call-leg API. For more information, see the *Call-leg Functions* chapter in the *SIP Stack Reference Guide*.

RvSipRegClientSetPersistency()

Sets the persistency level of reg-client objects.

RvSipTransactionSetPersistency()

Sets the persistency level of *transactions*.

RvSipTransmitterSetPersistency()

Sets the persistency level of *transmitter*s.

RvSipXXXGetPersistency()

You can use the RvSipXXXGetPersistency() function to query whether a specific object is persistent or not.

The following API function are supplied:

RvSipCallLegGetPersistency()

Gets the persistency level of *call-legs* and *subscriptions*.

Note To set the persistency level of a *subscriptions*, you must first get the Subscription-associated *call-leg* and then use the Call-leg API. For more information, see the *Call-leg Functions* chapter in the *SIP Stack Reference Guide*.

RvSipRegClientGetPersistency()

Gets the persistency level of reg-client objects.

RvSipTransactionGetPersistency()

Gets the persistency level of *transactions*.

RvSipTransmitterGetPersistency()

Gets the persistency level of *transmitters*.

RvSipXXXSetConnection()

You can use the RvSipXXXSetConnection() function to set a specific *connection* to a SIP Stack object. The application can create its own *connections* and insert them into the *connection* hash. Several *connections* can use the same transport, local and remote addresses. To insure that a specific SIP Stack object will use a specific *connection*, the application can set this *connection* to the SIP Stack object. The SIP Stack object will then attach to the *connection* and try to use it for further requests. If the *connection* address fits, it will be used. Otherwise the object will detach from the *connection* and use a different address.

Note You can set *connections* only to persistent objects.

The following API functions are supplied:

RvSipCallLegSetConnection()

Sets a *connection* to *call-legs* or *subscriptions*. The *call-leg* or subscription will set the *connection* to every client *transaction* it uses as long as the *connection* fits the local and remote addresses of the request. when the *connection* no longer fits, the *call-leg* or subscription will automatically detach from it.

Note To set the persistency level of a *subscriptions*, you must first get the Subscription-associated *call-leg* and then use the Call-leg API. For more information, see the *Call-leg Functions* chapter in the *SIP Stack Reference Guide*.

RvSipRegClientSetConnection()

Sets a *connection* to a *register-client*.

RvSipTransactionSetConnection()

Sets a *connection* to a *transaction*.

RvSipTransmitterSetConnection()

Sets a *connection* to a *transmitter*.

RvSipXXXGetConnection()

You can use this function to query a persistent object about the *connection* it is using.

Note Non persistent TUs do not attach to their *connection* and therefore the function will return NULL. The function will also return NULL for server transactions.

The following API functions are supplied:

RvSipCallLegGetConnection()

Gets the *connection* currently used by a *call-leg* or a subscription.

RvSipRegClientGetConnection()

Gets the *connection* currently used by a *register-client*.

RvSipTransactionGetConnection()

Gets a *connection* currently used by a *transaction*.

RvSipTransmitterGetConnection()

Gets a *connection* currently used by a *transmitter.*

RvSipXXXNewConnInUseEv()

This callback function notifies the application that the object is now using a new *connection*. The *connection* can be a totally new *connection* or a suitable *connection* that was found in the hash.

Working with Connections

The Transport layer API provides the means to establish, maintain, and release transport *connections*. The Transport API relates to the following entities:

- Connection (*connection*)
- Transport Manager (*TransportMgr*)

Connections

*Connection*s are used to reliably deliver data between two user agents using TCP or TLS transport. A SIP Stack *connection* is identified by its Local address, Remote address and Transport protocol (TCP or TLS). Your application can create and initialize *connections*, connect and disconnect *connections*, and apply *connections* for the use of specific SIP Stack objects. All application *connections* are inserted into the *connection* hash upon initialization. A *connection* is a stateful object which can assume any state from a set defined in the Transport API. A *connection* state represents the state of the *connection* establishment between two SIP User Agents.

Transport Manager

The *TransportMgr* manages the collection of *connections*. The *TransportMgr* is mainly used for creating new *connections*.

Working with Handles

All *connections* and the *TransportMgr* are identified using handles. You must supply these handles when using the Transport API:

RvSipTransportMgrHandle

Defines the *TransportMgr* handle. You receive this handle by calling RvSipStackGetTransportMgrHandle().

RvSipTransportConnectionHandle

Defines a *connection* handle. You receive this handle when creating a new client connection using the RvSipTransportMgrCreateConnection() function. Or, you receive it through the RvSipTransportConnectionCreatedEv() callback when the SIP Stack creates a new server connection.

CONNECTION API

The Transport API contains a set of functions and function callbacks that enable you to initialize new connections, set or examine connection parameters, connect and disconnect connections, and attach or detach from *connections*.

CONNECTION PARAMETERS

A *connection* holds the following parameters:

Local and Remote addresses

A Transport *connection* is opened from a specific local address to a specific remote address. The local and remote address parameters indicate these addresses. Each address includes the IP, port and address type (IPv4 or IPv6). The local and remote addresses along with the *connection* transport type provide the *connection* key. The *connection* is inserted into the *connection* hash according to this key and can be located and used by different SIP Stack objects.

Transport Type

Indicates whether the *connection* is a TCP or TLS *connection*. The transport type is part of the *connection* key.

Num Of Owners

As explained above, a *connection* can have several owners. The *connection* notifies each of its owners about *connection* events. As long as a client *connection* has owners, the *connection* remains connected. When the last owner detaches from the *connection*, the *connection* disconnects. This parameter holds the number of owners currently attached to the *connection*.

Connection State

Indicates the state of the TCP session setup between two SIP User Agents. You can only access the state parameter with a Get function and it is not modifiable.

Is Client

Indicates whether the *connection* is a TCP client or server.

Is Enabled

Indicates whether the *connection* is currently in the *connection* hash and therefore can be used by SIP Stack objects. The application can insert and remove a *connection* from the *connection* hash. (See the Connection Control Functions chapter).

Connection Creation and Initialization

The Transport API provide the following functions for creating and initializing new *connections*:

RvSipTransportMgrCreateConnection()

Constructs a new un-initialized *connection*. You supply this function with the *connection* owner handle and the event handlers to be used for this owner. Each owner can provide different event handlers. The *connection* keeps the owner handle with the event handler pointers of the owner. The owner is notified about *connection* events with the specific event handlers that were supplied. The new created *connection* always assumes the IDLE state. In the IDLE state the *connection* in not yet initialized and cannot be used by persistent SIP Stack objects.

RvSipTransportConnectionInit()

Initializes a *connection* with the required configuration parameters. This function receives a configuration structure of the RvSipTransportConnectionCfg type that includes the required *connection* configuration. You can call this function only in the IDLE *connection* state. After initialization is completed, the *connection* moves to the READY state and is automatically inserted into the *connection* hash. The *connection* can then be located and used by other SIP Stack persistent objects.

Note This function does not connect the *connection*. The *connection* will be automatically connected when a SIP Stack object uses it for sending a message or if you specifically call the RvSipTransportConnectionConnect() function. In both cases, the *connection* will assume the CONNECTING and then the TCP_CONNECTED states.

Connection Control Functions

The following API functions provide *connection* control:

RvSipTransportConnectionConnect()

Connects the *connection*. You can call this function only in the READY state. Calling this function will cause the *connection* to move to the CONNECTING state. The *connection* will move to the TCP_CONNECTED state when an indication is received from the network that the *connection* was successfully connected.

RvSipTransportConnectionTerminate()

The behavior of this function depends on the *connection* state. If the *connection* is in the TCP_CONNECTED state, the *connection* will start a normal disconnection process. *Connections* will move to the CLOSING state. TLS *connections* will first perform TLS closing procedures and then will move to the CLOSING state.

For all other states, the *connection* will close its internal socket if the socket was opened, and will terminate. After termination, the *connection* will assume the TERMINATED state.

Note If the *connection* has messages that it is about to send, the messages will be lost. It is, therefore, not recommended to use this function. If you no longer need this *connection*, call the RvSipTransportConnectionDetachOwner() function. The *connection* will be closed only when the last owner is detached. This means that if the *connection* is still being used by other SIP Stack objects, it will not be closed until these objects detach from it.

RvSipTransportConnectionAttachOwner()

Attaches a new owner to the supplied *connection* with a set of callback functions that will be used to notify this owner about *connection* events. You can use this function only if the *connection* is connected or is in the process of being connected. You cannot attach an owner to a *connection* that started its disconnection process.

Note The *connection* will not disconnect as long as it has owners attached to it.

RvSipTransportConnectionDetachOwner()

Detaches an owner from the supplied *connection*. If the *connection* is left with no other owners, it will be closed. If the same owner is attached to a *connection* more than once, the first matching owner will be removed. After detaching from a *connection*, you will stop getting *connection* events and you must no longer use the *connection* handle.

RvSipTransportConnectionEnable()

Inserts a *connection* into the hash so that persistent objects will be able to use it.

RvSipTransportConnectionDisable()

Removes a *connection* from the hash so that persistent objects will not be able to use it. Objects that are already using the *connection* (that are in the *connection* owner's list) will be able to continue to use the object even though it is out of the hash. However, other objects will not be able to use the *connection* as long as the *connection* is disabled.

CONNECTION EVENTS

The *connection* supplies several events in the form of callback functions to which your application may listen and react. To be notified of *connection* events, you must first become one of the *connection* owners. This can happen when the application created the *connection* or when it attached itself to an existing *connection*. In both cases, the owner supplies its event handler callback functions. These function are used to notify the owner about the *connection* events.

The following events are supplied with the Transport Connection API:

RvSipTransportConnectionStateChangedEv()

The *connection* is a stateful object that can assume different states according to the Connection state machine. Through this function, you receive notifications of *connection* state changes and the associated state change reason. In a regular *connection* life cycle, the reason for the state is set to RVSIP_TRANSPORT_CONN_REASON_UNDEFINED. When the *connection* is closed because of an error, the reason is set to RVSIP_TRANSPORT_CONN_REASON_ERROR.

Note You do not have to register to this callback if you do not want to get *connection* states.

RvSipTransportConnectionStatusEv()

The *connection* notifies about events that do not effect the *connection* state using the *connection* status callback. If, for example, there was an error in the *connection*, the *connection* will notify the application with RVSIP_TRANSPORT_CONN_STATUS_ERROR. The *connection* will then disconnect with the RVSIP_TRANSPORT_CONN_REASON_ERROR reason.

Note You do not have to register to this callback if you do not want to get *connection* statuses.

CONNECTION STATES

The Connection state machine represents the state of the *connection* between two SIP User Agents. The RvSipTransportConnectionStateChangedEv() callback reports *connection* state changes and state change reasons. The state change reason indicates why the *connection* reached the new state.

The *connection* associates with the following states:

RVSIP_TRANSPORT_CONN_STATE_IDLE

The IDLE state is the initial state of the Connection state machine. Upon creation of the *connection*, the *connection* assumes the IDLE state. It remains in this state until the RvSipTransportConnectionInit() function is called, whereupon it should move to the READY state.

RVSIP_TRANSPORT_CONN_STATE_READY

After calling the RvSipTransportConnectionInit() function that initializes the *connection* and inserts it into the *connection* hash, the *connection* enters the READY state. The *connection* will move from the READY state to the CONNECTING state in the following cases:

- If the application specifically called the RvSipTransportConnectionConnect() function on this connection.
- If one of the persistent SIP Stack objects located this *connection* in the hash and connected it to send a message.

Calling the RvSipTransportConnectionTerminate() function in the READY state will move the *connection* to the TERMINATED state.

RVSIP_TRANSPORT_CONN_STATE_CONNECTING

Calling the RvSipTransportConnectionConnect() function in the READY state will move the *connection* to the CONNECTING state. While in the CONNECTING state, the SIP Stack tries to connect the *connection* to the remote party. If the connect attempt succeeds, the *connection* assumes the TCP_CONNECTED state. Otherwise, the *connection* assumes the TERMINATED state with the RVSIP_TRANSPORT_CONN_REASON_ERROR reason.

RVSIP_TRANSPORT_CONN_STATE_TCP_CONNECTED

Upon receiving the network connected event, the *connection* moves to the TCP_CONNECTED state. While in this state, the *connection* can send and receive SIP *messages*. Calling RvSipTransportConnectionTerminate() in this state will move the *connection* to the CLOSING state and the *connection* will start the disconnection procedure. This will also be the case for client *connections* that are left with no owners.

RVSIP_TRANSPORT_CONN_STATE_CLOSING

A client *connection* that remains with no owners, or any *connection* on which the application called the RvSipTransportConnectionTerminate() function moves to the CLOSING state. While in this state, the *connection* cleans and then shuts down the established socket *connection*. The *connection* will move to the CLOSED state upon receiving a network event that indicates that the *connection* was closed by the remote party.

RVSIP_TRANSPORT_CONN_STATE_CLOSED

The *connection* moves to the CLOSED state upon receiving a network event that indicates that the *connection* was closed by the remote party. In the CLOSED state, the *connection* closes its internal socket. The *connection* then moves to the TERMINATED state.

RVSIP_TRANSPORT_CONN_STATE_TERMINATED

The final *connection* state. When a *connection* is terminated, the *connection* assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *connection*.

Client Connection State Machine

Figure 14-1 illustrates the Client Connection state machine.

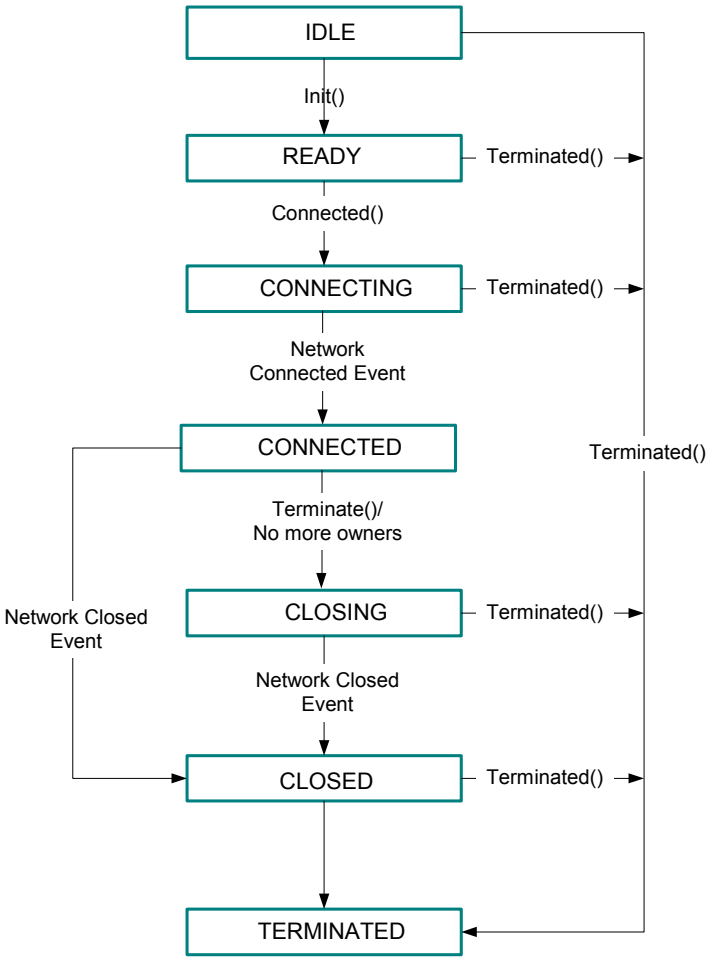


Figure 14-1 *Connection State Machine*

Sample Code

The following sample code demonstrates how to create an application *connection*. The application becomes the *connection* owner and supplies the *connection* with its event handlers. In this example, the application chooses to register only on the RvSipTransportConnectionStateChangedEv() callback function.

```
/*=========================================================================================*/
static RvStatus AppCreateAndInitConnection1(
            IN  RvSipTransportMgrHandle           hTransportMgr,
            OUT RvSipTransportConnectionHandle    *phConn)
{
     RvStatus                              rv;
     RvSipTransportConnectionCfg           connCfg;
     RvSipTransportConnectionEvHandlers    evHandlers;

     /*Initializes the event handler structure.*/
     memset(&evHandlers,0,sizeof(evHandlers));
     evHandlers.pfnConnStateChangedEvHandler = AppConnStateChangedEv;

     /*Creates a connection object and supplies the event handlers. In this sample, the
     owner is set to NULL.*/
     rv = RvSipTransportMgrCreateConnection(hTransportMgr, NULL, &evHandlers,
                                                    sizeof(evHandlers), phConn);
     if(rv != RV_OK)
     {
          printf("Failed to create a connection object");
          return rv;
     }

     /*Initializes the configuration structure. Setting the local address is optional.*/
     memset(&connCfg,0,sizeof(RvSipTransportConnectionCfg));
     connCfg.eTransportType = RVSIP_TRANSPORT_TCP;
     connCfg.strDestIp      = "172.20.1.1";
     connCfg.destPort       = 5060;

     rv = RvSipTransportConnectionInit (*phConn,&connCfg,sizeof(connCfg));
     if(rv != RV_OK)
     {
          printf("Failed to initialize the a connection object");
          return rv;
```

```
    }
    return RV_OK;

}
/*=========================================================================================*/
```

When working with calls or subscriptions, a message can be record-routed and the destination can be determined according to the Route list. For more information on initiating a TCP call, see the Working with Call-legs (Dialogs) chapter.

Sample Code

The following sample code demonstrates an implementation of the RvSipTransportConnectionStateChangedEv() callback function. In this implementation, the application only prints a message to the screen when the *connection* is connected.

```
/*=========================================================================
static RvStatus RVCALLCONV AppConnStateChangedEv(
    IN  RvSipTransportConnectionHandle              hConn,
    IN  RvSipTransportConnectionOwnerHandle         hOwner,
    IN  RvSipTransportConnectionState               eState,
    IN  RvSipTransportConnectionStateChangedReason  eReason)
{

      switch(eState) {
      case RVSIP_TRANSPORT_CONN_STATE_TCP_CONNECTED:
        printf("Connection %x state changed to TCP CONNECTED\n");
            break;
      default:
            break;
      }
      return RV_OK;
}
/*=========================================================================*/
```

SERVER CONNECTIONS

Connections that are opened to the SIP Stack are referred to as server *connections*. The same stateful objects that implement client *connections* are used for server *connections*. As with client *connections*, the *TransportMgr* manages the server *connections*. In contrast to client *connections*, the application cannot create server *connections*.

Generally, server *connections* should be closed by the party that initiated the *connection*. It is highly recommended not to terminate server *connections* from the application to prevent unexpected interruptions in message reception and sending flow.

Most server *connections* are not reusable and are not inserted into the *connection* hash. (For an exception to this, see Server Connection Reuse.) However, a *transaction* that uses a server *connection* becomes the *connection* owner. Since several messages can be sent on a single server *connection*, a server *connection* can be owned by several *transactions*. When a server *transaction* terminates, it detaches from the server *connection*.

Server Connection API

Since the same objects implement server and client *connections*, both have the same parameters and API functions for parameter inspection and modification. For more information, see Working with Connections.

Note that some of the Control and Set API functions of the *connection* are irrelevant for server *connections*, such as RvSipTransportConnectionConnect().

Server Connection Events

The *TransportMgr* provides several events regarding a server *connection* in the form of callback functions for server *connection* control. To get the events, the application should register to the corresponding callbacks after SIP Stack initialization by calling the RvSipTransportMgrSetEvHandlers() function.

The following events are supplied with the Transport Manager API:

RvSipTransportConnectionCreatedEv()

Notifies application about the creation of a server *connection*, as a result of accepting an incoming *connection*. This callback supplies a handle to the server *connection*. Additional information about the *connection*, such as the remote address, can be obtained by the Connection API. Using an OUT parameter, the application can order the SIP Stack to drop the *connection* immediately before the SIP Stack will process any data that was received on the *connection*.

RvSipTransportConnectionStateChangedEv()

A server *connection* is a stateful object that can assume different states according to the Connection state machine. Through this callback, you receive notifications of server *connection* state changes and the associated state change reason.

Note This callback signature is identical to the callback used for client *connection* state changes, and the states used for server *connections* are a subset of the client *connection* states.

SERVER CONNECTION STATES

The server *connection* states are a subset of the *connection* states. For more information, see Connection States. The *TransportMgr* RvSipTransportConnectionStateChangedEv() callback reports server *connection* state changes and state change reasons. The state change reason indicates why the server *connection* reached the new state.

The server *connection* associates with the following states:

RVSIP_TRANSPORT_CONN_STATE_IDLE

The IDLE state is the initial state of the Connection state machine. Upon creation of the *connection*, the *connection* assumes the IDLE state. Since the server *connection* is created upon accepting the incoming *connection*, it moves to the CONNECTED state immediately after the RvSipTransportConnectionCreatedEv() is reported to the application. If the application requests the SIP Stack to drop the *connection*, the *connection* will move to the TERMINATED state instead of the CONNECTED state.

RVSIP_TRANSPORT_CONN_STATE_TCP_CONNECTED

The server *connection* moves to the TCP_CONNECTED state immediately after creation if the application approved the *connection* and did not request to drop it (through the RvSipTransportConnectionCreatedEv() callback OUT parameter). While in this state, the *connection* can receive and send SIP *messages*. Closing the *connection* by the remote side will move the *connection* to the CLOSED state and the *connection* from the TCP_CONNECTED to CLOSED state.

Note It is highly recommended **not** to close the *connection* using the RvSipTransportConnectionTerminate() function.

RVSIP_TRANSPORT_CONN_STATE_CLOSED

The *connection* moves to the CLOSED state upon receiving a network event that indicates that the *connection* was closed by the remote party. In the CLOSED state, the *connection* closes its internal socket. The *connection* then moves to the TERMINATED state.

RVSIP_TRANSPORT_CONN_STATE_TERMINATED

The final *connection* state. When a *connection* is terminated, the *connection* assumes the TERMINATED state. Upon reaching the TERMINATED state, you can no longer reference the *connection*.

Server Connection State Machine

Figure 14-2 illustrates the Server Connection state machine.

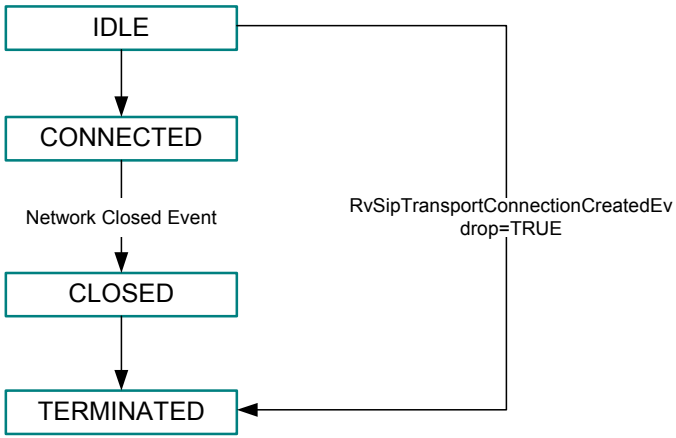


Figure 14-2 *Server Connection State Machine*

Closing Server Connections

As opposed to client *connection*s, it is usually not recommended to close a server *connection* when the last owner detaches from it. It is best to leave closing the *connection* to the *connection* initiator. However, the SIP Stack provides three options of handling the closing of a server *connection* using the *serverConnectionTimeout* configuration parameter. The application options are:

- serverConnectionTimeout = 0

  The *connection* is closed immediately when the last owner detaches from it. This behavior is not recommended.

- serverConnectionTimeout = –1

  The SIP Stack never closes Server *connections*, which always wait to be closed by the *connection* initiator. This is also the default behavior of the SIP Stack.

- serverConnectionTimeout > 0

  When the last owner detaches from the *connection*, a timer with the value of "serverConnectionTimeout" is set. The SIP Stack will close the *connection* when the timer expires. If in that time a new owner will attach itself to the *connection* (a message will be received on the *connection*), the timer will be stopped.

Sample Code

The following sample code demonstrates how an application can inspect incoming *connections*. Registration of the application to the callback is not shown.

```
/*=========================================================================*/
static RvStatus RVCALLCONV AppTransportConnectionCreatedEv(
                     IN  RvSipTransportMgrHandle             hTransportMgr,
                     IN  RvSipAppTransportMgrHandle          hAppTransportMgr,
                     IN  RvSipTransportConnectionHandle      hConn,
                     OUT RvSipTransportConnectionAppHandle   *phAppConn,
                     OUT RvBool                              *pbDrop)
{
    RvStatus rv;
    RvChar   strIPLocal[RVSIP_TRANSPORT_LEN_STRING_IP];
    RvChar   strIPRemote[RVSIP_TRANSPORT_LEN_STRING_IP];
    RvUint16 portLocal;
    RvUint16 portRemote;
    RvSipTransportAddressType eAddressType;

    rv = RvSipTransportConnectionGetLocalAddress(hConn,strIPLocal, &portLocal,
                                          &eAddressType);
    if (RV_OK != rv)
    {
        printf("RvSipTransportConnectionGetLocalAddress() failed");
        *pbDrop = RV_TRUE;
    }
    rv = RvSipTransportConnectionGetRemoteAddress(hConn,strIPRemote,
                  &portRemote, &eAddressType);
    if (RV_OK != rv)
```

```
    {
        printf("RvSipTransportConnectionGetRemoteAddress() failed");
        *pbDrop = RV_TRUE;
    }

    if (portRemote < 5000 ||  portRemote > 60000)
    {
        printf("0x%p incoming connection is DISAPPROVED by Application: remote addr
                %s:%d, local addr %s:%d", hConn, strIPRemote, portRemote, strIPLocal,
                portLocal);
      *pbDrop = RV_TRUE;
    }

    return RV_OK;
}
/*=========================================================================*/
```

Server Connection Reuse

When using a reliable transport, SIP entities use a *connection* to send a request. This *connection* typically originates from an ephemeral port. The SIP protocol includes mechanisms which ensure that responses to a request reuse the existing *connection* that is typically still available. The SIP protocol also includes provisions for reusing existing client *connections* for other requests sent by the originator of the *connection*. (For more information, see Persistent Connection Handling). However, new requests sent in the opposite direction are unlikely to reuse the existing *connection*. This frequently causes a pair of SIP entities to use one *connection* for requests sent in each direction, as shown in Figure 14-3.

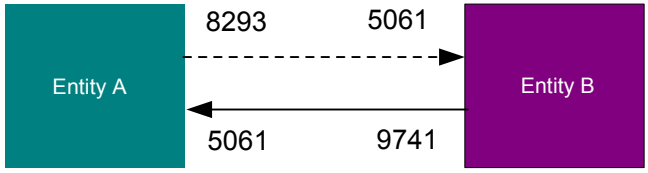


Figure 14-3 *Double Connection between SIP Entities*

The Connection Reuse feature (defined in *draft-ietf-sip-connect-reuse-03*) implements a *connection* sharing mechanism that enables user agents (UAs) to use incoming server *connections* for sending outgoing requests, so that a single *connection* between two entities will be sufficient.

Both the client and the server take part in the *connection* reuse mechanism.

CLIENT SIDE

The client opens new *connections* with the server. The client should indicate to the server if a specific opened *connection* can be reused by the server for new requests. This indication is carried as a new parameter called "alias" that is added to the Via header of a request sent on this *connection*.

For example:

```
Via: SIP/2.0/TLS 10.54.32.1:5061;branch=z9hG4bKa7;alias
```

The alias parameter indicates that the server should handle this *connection* as a *connection* that came from the address of the sent-by portion in the Via header (10.54.32.1:5061) and not the real destination address (that includes an ephemeral port). The sent-by part of the Via header is referred to as the "advertised address" of this *connection*.

Another example is of a request that arrived over a *connection* from IP 10.54.32.1 and port 8241. The top Via is:

```
Via: SIP/2.0/TLS proxy-farm-example.com; branch=z9hG4bK7;alias
```

The *connection* alias is "`proxy-farm-example.com`". A server that wants to send a request to this host (`proxy-farm-example.com`) can use this *connection*, with no need to make a DNS resolution.

A client application that wishes to enable a *connection* for server reuse may add the alias in the RvSipXXXFinalDestResolvedEv() callback function.

Sample Code

In the code example below, a client enables a *connection* for server reuse in RvSipTranscFinalDestResolvedEv() by doing the following:

1. Updating the top Via header—only in requests.
2. Setting an alias parameter ("`;alias`").
3. Setting the "advertised address" of the alias parameter (which is the host string) to be "`alias.xxx`".
4. Removing the port number.

```
/*-------------------------------------------------------------------------*/
RvStatus RVCALLCONV AppTranscFinalDestResolvedEv (
IN    RvSipTranscHandle        hTransc,
IN    RvSipTranscOwnerHandle   hLuaTransc,
IN    RvSipMsgHandle           hMsgToSend)

{
    RvSipViaHeaderHandle        hVia;
```

```
    RvSipHeaderListElemHandle   hListElem = NULL;
    RvStatus                    rv = RV_OK;
    RvSipTransport eTransport;

    if(RvSipMsgGetMsgType(hMsgToSend) == RVSIP_MSG_REQUEST)
    {
        /* Get the top Via header. */
        hVia = RvSipMsgGetHeaderByType(hMsgToSend, RVSIP_HEADERTYPE_VIA,
                                                    RVSIP_FIRST_HEADER, &hListElem);

         /* Get the transport parameter from the top Via header. */
        eTransport = RvSipViaHeaderGetTransport(hVia);
        if(eTransport == RVSIP_TRANSPORT_UDP)
        {
            /* No need to set an alias for request over UDP... */
            return RV_OK;
        }
        /* Sets the ";alias" parameter in the Via header. */
        rv = RvSipViaHeaderSetAliasParam(hVia, RV_TRUE);
        if(RV_OK != rv)
        {
            printf("Failed to set the ;alias parameter in top Via ");
            return rv;
        }
        /* Set the alias string in the sent-by part of the Via header. */
        rv = RvSipViaHeaderSetHost(hVia, "alias.xxx");
        if(RV_OK != rv)
        {
            printf("Failed to set the alias string in top Via header");
            return rv;
        }
        /* Remove the port from the Via header */
        rv = RvSipViaHeaderSetPortNum(hVia, UNDEFINED);
        if(RV_OK != rv)
        {
            return rv;
        }
    }
    return RV_OK;
}
/*============================================================================*/
```

Server Side

When the SIP Stack receives a request message on an incoming *connection*, it checks the Via header and looks for the alias parameter. The alias parameter indicates that the originator of the request wants to create a Transport layer alias. If the alias parameter exists, the SIP Stack consults the application whether this *connection* is authorized to be reused, using the dedicated callback function, RvSipTransportConnectionServerReuseEv().

If the application decides that this *connection* is authorized, it calls to RvSipTransportConnectionEnableConnByAlias(). In this function the SIP Stack hashes the *connection* with the advertised address, so the application will be able to use this *connection* for sending further requests. Reusing the connection is done simply by sending a request to the advertised address. For example, if the application wishes to send a request to 10.54.32.1:5061 or to proxy-farm-example.com (which are advertised addresses declared as aliases and enabled by application) the application will simply use the addresses as the Request-URI. During the sending process, the SIP Stack will find such a server *connection* in the hash and use it.

Server Connection Reuse API functions and Events

The Server Connection Reuse API functions and events are listed below:

RvSipTransportConnectionEnableConnByAlias()

When a request is received, and its top Via header has an alias parameter, this *connection* can be reused for sending outgoing requests. However, before using this *connection*, the application has to authorize it (especially when this is a TLS *connection*). Only when the application is certain that this *connection* is authorized it should enable the *connection* for future usage by calling this function.

In this function the SIP Stack inserts the *connection* into the *connections* hash table by its alias name. Therefore, when you try to send a request to this alias name, the *connection* will be found and reused.

RvSipTransportConnectionServerReuseEv()

This callback notifies the application that a new request was received with an alias parameter in its top Via header. The server *connection* should be reused, but it has to be authorized by the application first.

Sample Code

The following sample code demonstrates an implementation of the RvSipTransportConnectionServerReuseEv() callback function. In this implementation, the application checks the *connection* authorization (out of the scope of this sample) and if authorized, enables it.

```
/*-----------------------------------------------------------*/
void RVCALLCONV AppTransportConnectionServerReuseEv(
        IN  RvSipTransportMgrHandle              hTransportMgr,
        IN  RvSipAppTransportMgrHandle           hAppTransportMgr,
        IN  RvSipTransportConnectionHandle       hConn,
        IN  RvSipTransportConnectionAppHandle    hAppConn)
{
    RvStatus  rv;
    RvBool    bAuthorized;

    bAuthorized = AppIsConnectionAuthorized(hConn, hAppConn);

    if(bAuthorized == RV_TRUE)
    {
        rv = RvSipTransportConnectionEnableConnByAlias(hConn);
        if(rv != RV_OK)
        {
            printf("Failed to Enable Connection By Alias");
            return;
        }
    }
}
/*-----------------------------------------------------------*/
```

Authorizing a Server Connection

Authorizing *connection* aliases is essential to prevent *connection* hijacking. The following authorization process is recommended in *draft-ietf-sip-connect-reuse-03*. To correctly authorize an alias, the SIP node authorizing the request needs to recognize both the active *connection* and the alias as the same resource. The only way to accomplish this is if both the active *connection* and the alias can be authenticated by using the same credentials, using TLS mutual authentication as follows:

- Performing a DNS procedure on the subjectAltName of the originator certificate will give the “advertised-address” of the Via header.
- Performing a DNS procedure on the advertised-address will give the received IP address.

The application can use the SIP Stack Transport and Resolver APIs to authorize a *connection*. You can see an example of *connection* authorization in the simpleConnectionReuse sample application in the the Sample Applications chapter.

Using TCP Transport

RFC 3261 defines the use of TCP transport by using the transport=TCP parameter in a SIP URL address that indicates the destination of the message. When working with *transactions* or *register-clients*, the next hop is determined by the Request-URI of the message. Therefore, to send a message with TCP, you need to set the transport=TCP parameter in the Request-URI parameter of the object.

For *connection*-oriented operations, the SIP Stack uses non-blocking sockets. The result of using non-blocking sockets is that the sending of a message can be completed after an API call has returned. For example, the call for RvSipTransactionRequest() using TCP may return before the request message is sent. The SIP Stack will manage the sending operation until it is completed.

Using TLS Transport

TLS is a security mechanism that operates on the Transport layer, on top of TCP transport. By using TLS as a *connection* transport, a SIP entity can send and receive data in a secure authenticated manner.

TLS, together with the commonly used Public Key Infrastructure certification distribution mechanism achieves the following goals:

- Guarantees the identity of a remote computer
- Transmits messages to that remote computer in a secure encrypted manner.

TLS uses pairs of asymmetrical encryptions keys to guarantee the identity of a remote computer. The public key of each remote computer is published in a certificate.

A certificate is a document digitally signed by a certificate authority that both sides of the *connection* agreed to trust before the TLS *connection* establishment has started. (VeriSign and Thawte are examples of such certificate authorities). In the TLS *connection* establishment process, the certificate of the remote computer is retrieved and verified and a new key and encryption algorithm is negotiated for the specific *connection*.

TLS Connection Establishment

TLS *connection* establishment requires the completion of the following three phases:

- **Phase 1: TCP connection establishment**—as stated above, TLS uses TCP as its underlying transport protocol. Therefore, a TLS handshake can start only after a TCP *connection* has reached the CONNECTED state.
- **Phase 2: TLS handshake**—the basic TLS handshake process consists of several TCP messages which go from client to server and from server to client, in which the client retrieves the server's certificate, verifies it, negotiates an encryption key and algorithm for the session, and both parties make sure that the security of the handshake has not been compromised. For more information on the TLS handshake see RFC 2246 and RFC 3546.
- **Phase 3: Post connection assertion**—In this phase, the client makes sure that the certificate handed to it by the server does indeed belong to server. This step is taken to prevent the situation in which a server named *malise.com* will present a valid certificate of *someonelse.com*.

After these phases have been completed, encrypted messages can be transmitted on the *connection* in a secure manner.

TLS and SIP

RFC 3261 defines the use of TLS as a transport mechanism by using the "sips:" scheme. When using the "sips:" scheme in a URI—or any other header that indicates the next hop of a message, such as Route, Via, and so on—RFC 3261 mandates the transport to be TLS. (For this reason TLS will not guarantee a secure delivery end-to-end, but only to the next hop).

SIP Stack and TLS

The SIP Stack uses an open source library called "openSSL" that provides TLS and encryption services. For more information about openSSL, see the openSSL project website at http://www.openssl.org.

To compile the SIP Stack with TLS, use the RV_TLS_ON compilation flag. (When compiling on UNIX systems, you can use the tls=on compilation line parameter).

TLS Stack Objects

The TLS uses TLS engine objects and TLS *connections* to allow sending messages with TLS transport.

TLS Engine

A TLS engine is an entity that binds together several parameters needed for TLS, such as SSL version, engine certificate, trusted root certificates authorities, and so on. In most cases, a TLS engine will be constructed immediately after SIP Stack initialization and will "live" for the entire duration of the SIP Stack's life. All TLS engines are destructed with the SIP Stack when it is destructed.

Using A TLS engine lets the application use similar TLS parameters on different *connections*. In a "simple" client application that only wants to authenticate servers, you will usually use one TLS engine with no certificate, and several trusted root CAs.

When implementing a proxy, a TLS engine will most likely be associated with one "leg" of the proxy. This way the proxy can present one TLS policy to its local organization and a different TLS policy—perhaps one with a stronger encryption—to an outside organization or the internet.

TLS Engine API

A TLS engine is represented by the RvSipTransportTlsEngineHandle handle. The following functions are for constructing a TLS engine and setting its parameters:

RvSipTransportTlsEngineConstruct()

This function constructs a TLS engine.The RvSipTransportTlsEngineCfg structure received by this function includes Engine configuration parameters.

The RvSipTransportTlsEngineCfg contains the following members:

- **RvSipTransportTlsMethod**—indicates the version of SSL to use: SSLv2, SSLv3 or TLS.
- **strPrivateKey, ePrivateKeyType, privateKeyLen**—informs the engine of its private key. The private key is given as a string.
- **strCert, certLen**—defines the certificate that an engine will present on TLS handshakes.
- **certDepth**—defines the depth that an engine will consider legal in a certificate chain to which it is presented.

This set of parameters cannot be changed after an engine has been initialized.

RvSipTransportTlsEngineCheckPrivateKey()

After an engine has been constructed, you may use this function to make sure the certificate and the private key loaded into the engine match.

RvSipTransportTlsEngineAddTrustedCA()

A TLS engine can trust zero, one or more root certificates. Once an engine trusts a root certificate, it will approve all valid certificates issued by that root certificate. Trusted certificates are (usually) root certificates. You add trusted certificates to an engine by using RvSipTransportTlsEngineAddTrustedCA() after the engine has been constructed.

RvSipTransportTlsEngineAddCertificateToChain()

An engine may hold a certificate that is not issued directly by a root certificate, but by a certificate authority delegated by that root certificate. To add this intermediate certificate to the chain of certificates that the engine will present during a handshake, use RvSipTransportTlsEngineAddCertificateToChain() after the engine has been constructed.

Sample Code

The following sample shows how to initialize a “server” TLS engine that will display certificates upon request.

Note In all code examples in this chapter, openSSL is used to load and manipulate certificates and key files. Other means of key and certificate loading can also be used.

Sample Code

```
/*=========================================================================================*/
#include <openssl/ssl.h>
#define SERVER_KEY_N_CERT_FILE "server.keyAndCert.pem"

static RvSipTransportMgrHandle g_hTransportMgr;

static void InitTlsSecurity()
{
      RvStatus                        rv                      = RV_OK;
      BIO                             *inKey                  = NULL;
      BIO                             *inCert                 = NULL;
      EVP_PKEY                        *pkey                   = NULL;
      X509                            *x509                   = NULL;
      RvSipTransportTlsEngineCfg      TlsEngineCfg;
      RvChar                          privKey[STRING_SIZE]= {'\0'};
```

```
  unsigned char                   *keyEnd               = privKey;
  RvInt                           privKeyLen            = 0;
  RvChar                          cert[STRING_SIZE]     = {'\0'};
  unsigned char                   *certEnd              = cert;
  RvInt                           certLen               = 0;
  RvSipTransportTlsEngineHandle hTlsServerEngine        = NULL;

  memset(&TlsEngineCfg,0,sizeof(TlsEngineCfg));

/*Loads the key for the server engine.*/
inKey=BIO_new(BIO_s_file_internal());
if (BIO_read_filename(inKey,SERVER_KEY_N_CERT_FILE) <= 0)
{
    HandleErrorFunction("Cannot load key");
}

pkey=PEM_read_bio_PrivateKey(inKey,NULL,NULL,NULL);
privKeyLen = i2d_PrivateKey(pkey,NULL);
privKeyLen = i2d_PrivateKey(pkey,&keyEnd);
BIO_free(inKey);

/*Loads the certificate for the server engine.*/
inCert=BIO_new(BIO_s_file_internal());
if (BIO_read_filename(inCert,SERVER_KEY_N_CERT_FILE) <= 0)
{
    HandleErrorFunction("Can not load certificate");
}

x509=PEM_read_bio_X509(inCert,NULL,NULL,NULL);
certLen = i2d_X509(x509,NULL);
certLen = i2d_X509(x509,&certEnd);
BIO_free(inCert);

/*Initializes the configuration structure for the server engine.*/
TlsEngineCfg.eTlsMethod         = RVSIP_TRANSPORT_TLS_METHOD_TLS_V1;
TlsEngineCfg.strCert            = cert;
TlsEngineCfg.certLen            = certLen;
TlsEngineCfg.strPrivateKey      = privKey;
TlsEngineCfg.privateKeyLen      = privKeyLen;
TlsEngineCfg.ePrivateKeyType    = RVSIP_TRANSPORT_PRIVATE_KEY_TYPE_RSA_KEY;
TlsEngineCfg.certDepth          = 5;
```

```
    /*Uses the initialized configuration to build the server TLS engine*./
    rv = RvSipTransportTlsEngineConstruct(g_hTransportMgr,
                                          &TlsEngineCfg,
                                          sizeof(TlsEngineCfg),
                                          &hTlsServerEngine);
    if (RV_OK != rv)
    {
        HandleErrorFunction("failed to construct TLS server engine");
    }
    /*Makes sure that the private key matches the certificate installed on the engine.*/
    rv = RvSipTransportTlsEngineCheckPrivateKey(g_hTransportMgr,hTlsServerEngine);
    if (RV_OK != rv)
    {
        HandleErrorFunction("Key and private certificate don't match");
    }
}
/*=========================================================================================*/
```

TLS Connection

A TLS *connection* is an entity that represents a TLS *connection* on which data can be transmitted in a secure manner. When a TLS *connection* gets to the TCP CONNECTED state, a TLS handshake can be initiated. When the TLS handshake and positive *connection* assertion have been completed, data can be transmitted on the *connection*.

TLS Connection API

A TLS *connection* is represented by the RvSipTransportConnectionHandle handle. The TLS Connection API functions are as follows:

RvSipTransportConnectionTlsHandshake()

Moves the TLS *connection* from the HANDSHAKE_READY state to the HANDSHAKE_STARTED state by starting a TLS handshake. The *hEngine* parameter indicates which engine is responsible for this *connections* handshake. One option is to examine the remote IP address of the *connection*, and using this address, decide which engine is responsible for this handshake.

The *pfnVerifyCertEvHandler* parameter determines the certificate verification callback. For more information on this callback, see the RvSipTransportVerifyCertificateEv() paragraph in the callbacks section.

RvSipTransportConnectionGetCurrentTlsState()

Gets the current TLS state of a *connection*.

RvSipTransportConnectionTlsGetEncodedCert()

Gets the encoded certificate of a *connection*.

RvSipTransportConnectionTlsRenegotiate()

Restarts the TLS handshake on the next received message. As a result, the certificate will be renegotiated and the session key will be regenerated. The state of the *connection* is not affected by the undergoing process of the renegotiation. The application does not receive any indication about renegotiation success. However, in case of failure, the *connection* will be terminated.

TLS Connection Events

The TLS *connection* supplies several events in the form of callback functions to which your application may listen and react. Use RvSipTransportMgrSetEvHandlers() to register on these callbacks.

Note To work with TLS, you must implement and register on the RvSipTransportConnectionTlsStateChangedEv() callback.

RvSipTransportConnectionTlsSequenceStartedEv()

The SIP Stack lets you create your own handle to a TLS *connection*. This will prove useful when you have your own application TLS *connection* database. You can provide the SIP Stack with your TLS *connection* handle, which the SIP Stack will supply when calling your application callbacks.

After a TLS *connection* has reached the TCP CONNECTED state, the RvSipTransportConnectionTlsSequenceStartedEv() callback will be called, allowing the application to exchange handles with the SIP Stack.

RvSipTransportConnectionTlsStateChangedEv()

Informs the application of the various stages of the TLS *connection* establishment. In this callback the application will be informed of the HANDSHAKE_READY state. In this state the application must call the RvSipTransportConnectionTlsHandshake() function to start the handshake process. An application that wishes to work with TLS **must** implement this callback.

RvSipTransportConnectionTlsPostConnectionAssertionEv()

As described in the introduction, a TLS attack can be performed in the following manner: computer *mallice.com* holds the valid certificate of *p.com* and displays it in the handshake process. The certificate is valid so the handshake will be completed successfully. If this attack is preformed, the user might deliver data to an unauthenticated party. To prevent this, the SIP Stack compares parameters from within the certificate to the *connection* destination (usually the URI or the Route headers). If this comparison fails, the RvSipTransportConnectionTlsPostConnectionAssertionEv() event is called, allowing the application to override the decision of the SIP Stack, and complete the TLS *connection* establishment.

RvSipTransportVerifyCertificateEv()

This callback, which is passed in the RvSipTransportConnectionTlsHandshake() function, lets the application control the default certificate verification. The parameter meaning is different for TLS client *connections* and TLS server *connections*. For further elaboration of this parameter see the *SIP Stack Reference Guide*.

Passing a non-NULL parameter will enable the application to examine incoming certificates, analyze data on these certificates and override the pass/fail decision on the certificates. In this callback you can retrieve data regarding the certificate that is being examined.

The RvSipTransportTlsGetCertVerificationError() function is used to examine the certification error.

The RvSipTransportTlsEncodeCert() function is used to retrieve the encoded certificate.

Sample Code

The following sample code shows how to examine some of the data stored in a certificate during the handshake process.

```
/*-------------------------------------------------------------------------*/
RvInt32 AppTransportVerifyCertificateEventHandler(
                        IN    RvInt32                       prevError,
                        IN    RvSipTransportTlsCertificate  certificate)
{
    RvChar          szCert[2048]     = {'\0'};
    RvChar          szLogData[2048]  = {'\0'};
    RvChar          szTmpData[2048]  = {'\0'};
    X509            *pCert              = NULL;
```

```
    RvInt           certLen             = TLS_CERT_STR_LENGTH;
    RvStatus        rv                  = RV_OK;
    RvChar          *pszCert            = (RvChar*)&szCert;

    /*Obtains the certificate as string from the certificate object.*/
    rv = RvSipTransportTlsEncodeCert(certificate,&certLen,szCert);
    if (RV_OK != rv)
    {
        HandleErrorFunction("failed to get certificate");
    }

    /*Converts the certificate to open SSL format.*/
    pCert = d2i_X509(0,(unsigned char**)&pszCert,certLen);
    sprintf(szLogData,"Cert Analysis - issuer:");

    /*Who is the issuer of the certificate?*/
    X509_NAME_oneline(X509_get_issuer_name(pCert),szTmpData,2048);
    strcat(szLogData,szTmpData);
    strcat(szLogData,", subject name: ");

    /*Who is the subject of this certificate?*/
    X509_NAME_oneline(X509_get_subject_name(pCert),szTmpData,2048);
    strcat(szLogData,szTmpData);

    /*Frees the certificate.*/
    X509_free(pCert);
    printf(szLogData);

    /*Does not change the SSL pass/fail decision.*/
    return prevError;
}
/*=========================================================================================*/
```

TLS CONNECTION STATES

The TLS Connection state machine represents the state of the TLS *connection* between two SIP User Agents. The RvSipTransportConnectionTlsStateChangedEv() callback reports TLS *connection* state change.

The TLS *connection* associates with the following states:

RVSIP_TRANSPORT_CONN_TLS_STATE_UNDEFINED

No TLS sequence was initiated on the *connection*.

RVSIP_TRANSPORT_CONN_TLS_STATE_HANDSHAKE_READY

The *connection* is TCP-connected and ready to start the TLS handshake. To move the *connection* to the TLS_CONNECTED state, call RvSipTransportConnectionTlsHandshak().

RVSIP_TRANSPORT_CONN_TLS_STATE_HANDSHAKE_STARTED

The *connection* started the handshake process.

RVSIP_TRANSPORT_CONN_TLS_STATE_HANDSHAKE_COMPLETED

The handshake procedure on the *connection* was completed. Post *connection* verification might still be needed if the *connection* requested the certificate of the remote party.

RVSIP_TRANSPORT_CONN_TLS_STATE_CONNECTED

Data can be sent on the *connection*.

RVSIP_TRANSPORT_CONN_TLS_STATE_HANDSHAKE_FAILED

The TLS handshake failed. No data can be transmitted on the *connection*.

RVSIP_TRANSPORT_CONN_TLS_STATE_CLOSE_SEQUENSE_STARTED

The *connection* has received or sent a close request but is not closed yet.

RVSIP_TRANSPORT_CONN_TLS_STATE_TERMINATED

The *connection* is terminated. After this point the *connection* may not be accessed again.

TLS Connection State Machine

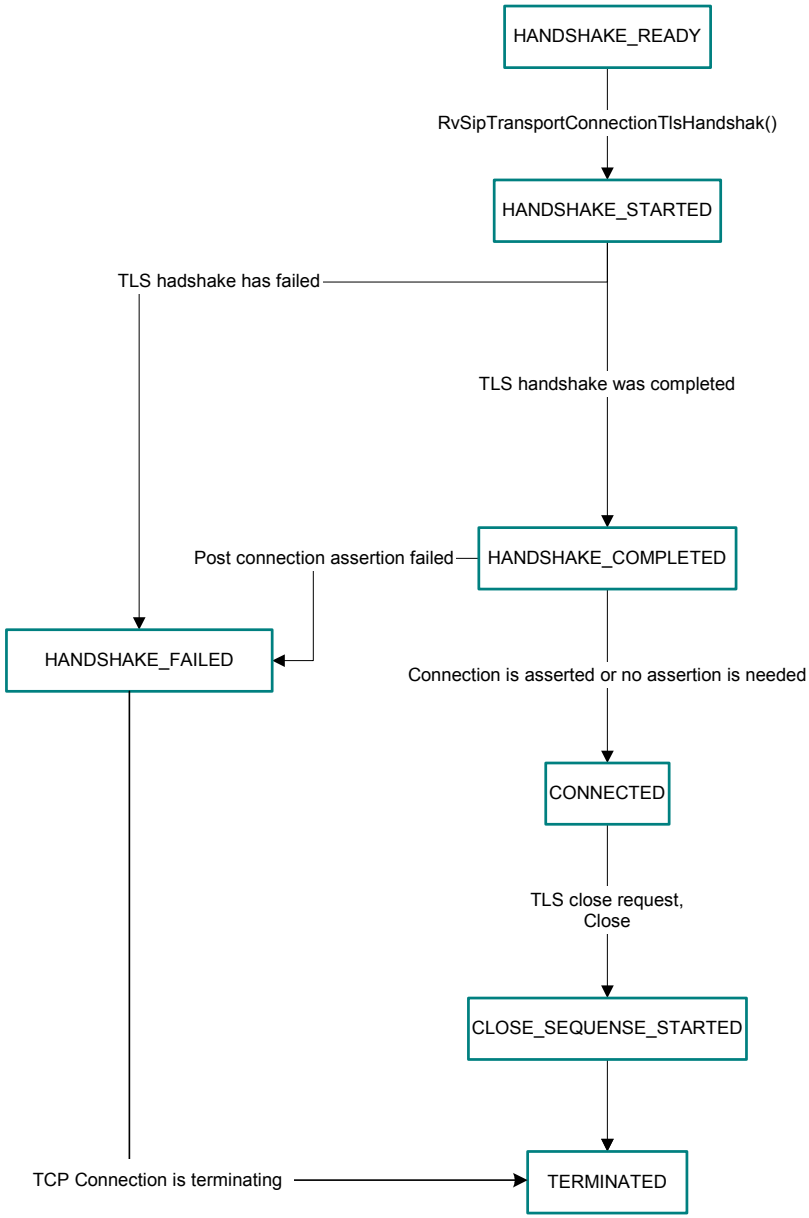


Figure 14-4 *TLS Connection State Machine*

Sample Code

The following sample code shows how to start a TLS handshake

```
/*=========================================================================================*/
static RvSipTransportTlsEngineHandle       g_hTlsServerEngine;

RvStatus RVCALLCONV TransportConnectionTlsStateChanged(
                              IN    RvSipTransportConnectionHandle               hConnection,
                             IN    RvSipTransportConnectionAppHandle            hAppConnection,
                              IN    RvSipTransportConnectionTlsState             eState,
                              IN    RvSipTransportConnectionStateChangedReason   eReason)
{
    RvStatus rv   = RV_OK;

    switch (eState)
    {
    case RVSIP_TRANSPORT_CONN_TLS_STATE_HANDSHAKE_READY:
                  rv = RvSipTransportConnectionTlsHandshake(hConnection,
                             g_hTlsServerEngine,
                             RVSIP_TRANSPORT_TLS_HANDSHAKE_SIDE_DEFAULT,
                             NULL);
        break;
    default:
        break;
    }

    return rv;
}
/*=========================================================================================*/
```

CONFIGURATION PARAMETERS

The following configuration parameters are supplied for TLS enabled applications:

numOfTlsAddresses

The number of TLS addresses on which the application wishes to listen.

localTlsAddresses and localTlsPorts

The local TLS addresses on which the SIP Stack will listen.

numOfTlsEngines

The maximum number of TLS engines. TLS engines are used to give a set of properties to a TLS *connection*.

maxTlsSessions

The maximum number of TLS sessions. A TLS session is the TLS equivalent to a TCP *connection* and contains TLS data required to manage the TLS *connection*. For more information see the Configuration chapter.

Both local and remote peers can change their multihoming address sets by removing the binding to some of the multihoming IPs, or by binding sockets to the multihoming IPs. Local multihoming addresses can be updated both for *connection* and local address objects.

The following API functions can be used to change the local multihoming addresses:

The following callback can be used for monitoring changes of the multihoming addresses by the remote peer:

RvSipTransportConnectionStatusEv()

The *connection* notifies the application about events that do not affect the *connection* state using the *connection* status callback. Upon receiving notification about changes in the multihoming addresses from the remote peer, the SIP Stack calls the status callback. With the call to the callback, the SIP Stack provides the following:

- The address that was changed
- The type of change (address removal or addition)

Sample Code

The following code sample monitors the status of the remote multihoming addresses.

```
/*============================================================*/
void RVCALLCONV AppConnectionStatusEvHandler(
                 IN RvSipTransportConnectionHandle        hConn,
                 IN RvSipTransportConnectionOwnerHandle   hOwner,
                 IN RvSipTransportConnectionStatus        eStatus,
                 IN void*                                 pInfo)
{
      switch(eStatus)
     {
```

```
            pritf("Connection 0x%x: remote multihoming address %s was %s", hConn,
            pAddrInfo->address.strIP,
          }
           break;

     default:
                printf("Connection 0x%x: status %d was reported", hConn, eStatus);
     }
     return;
}
/*=====================================================================*/
```

In some standards, the primary address is referred to as a path. Paths can be changed by peers at any moment after the *connection* was established.

Working with IPv6 Addresses

IPv6 (Internet Protocol Version 6) is the "next generation" protocol designed by the IETF to replace the current internet protocol, IPv4. IPv6 overcomes a number of problems in IPv4, such as the limited number of available IPv4 addresses. It also adds many improvements to IPv4 in areas such as routing and network auto-configuration. IPv6 is expected to gradually replace IPv4, with the two coexisting for a number of years during a transition period.

The most significant change is that IPv6 supports an address scheme that uses 128 bit address space compared with the 32 bit IPv4 address. The SIP Stack enables an application to work with the IPv4 IP Stack, IPv6 IP Stack or dual stacks. Supporting the IPv6 scheme affects the low-level services needed from the operating system as well as message and headers syntax. It does not affect the way SIP works with *transaction call-legs* or other objects.

IPv6 Address Syntax

In IPv6 addresses there are eight groups of four digits each. The hexadecimal number system is used for the digits. Thus, each group occupies 16 bits of space and the entire address represents (but does not always require) 128 bits.

For example, 3ffe:6a88:85a3:08d3:1319:8a2e:0370:7344. is a valid address. If a 4 digit group is 0000, it may be omitted, thus in the syntax of IPv6, 3ffe:6a88:85a3:0000:1319:8a2e:0370:7344 is the same as 3ffe:6a88:85a3::1319:8a2e:0370:7344.

Following this rule, if more than two consecutive colons result from this omission, they may be reduced to two colons, as long as there is only one group of more than two consecutive colons. Thus, all the following addresses are valid and have the same meaning,

- 2001:2353:0000:0000:0000:0000:1428:57ab

- 2001:2353:0000:0000:0000::1428:57ab
- 2001:2353:0:0:0:0:1428:57ab
- 2001:2353:0::0:1428:57ab
- 2001:2353::1428:57ab

However, 2001::25de::cade is invalid. Also, leading zeros in all groups can be omitted, thus 2001:2353:02de::0e13 is the same as 2001:2353:2de::e13.

If the address is an IPv4 address in disguise, the last 32 bits may be written in decimal. Thus, ::ffff:192.168.89.9 is the same as ::ffff:c0a8:5909, but not the same as ::192.168.89.9 or ::c0a8:5909.

Scope ID

The scope ID parameter is required by the operating system. IPv6 addresses may use a scope ID to resolve the network interface that should be used.

Compiling the SIP Stack with IPv6

To compile the SIP Stack with IPv6 support, you can use one of the following options:

- Set the RV_NET_TYPE flag in *rvuserconfig.h* to RV_NET_IPV6.
- Compile the SIP Stack with the RV_CFLAG_IPV6 flag, or use `ipv6=on` in the make command.

Make sure that your system supports IPv6 and that IPv6 is installed.

IPv6 Addresses and SIP

The SIP protocol denotes that IPv6 addresses should be included in square brackets, []. For example, when using the IPv6 local loop address in a To header, the To header will appear as follows:

```
To: <sip:[::1]>
```

Initializing the SIP Stack with IPv6

Once the SIP Stack is compiled with IPv6, it is possible to initialize the SIP Stack with IPv6 addresses. To do so, follow the described steps:

1. Create a string that corresponds with an IPv6 address (such as, fec0::1234:123).
2. Enclose the address in square brackets (such as [fec0::1234:123]).
3. Add a scope ID to the string. To separate the scope ID from the string, use the percent sign (%) (such as [fec0::1234:123]%2).
4. Use this string as one of the local addresses in the SIP Stack configuration structure (such as localTcpAddress).

It is possible to start the SIP Stack with a mixture of IPv6 and IPv4 addresses.

IPv6 Addresses and the Transport Address Structure

Several SIP Stack API functions receive a structure pointer of the RvSipTransportAddr type as an input parameter. If you want to set an IPv6 address to a RvSipTransportAddr structure, follow these steps:

1. Create a string that corresponds to an IPv6 address (such as fec0::1234:123).
2. Set the string to the *strIP* member of the structure.
3. Set a scope ID to the *Ipv6Scope* member of the structure.
4. Set the *eAddrType* member of the structure to RVSIP_TRANSPORT_ADDRESS_TYPE_IP6.

Do not use square brackets with RvSipTransportAddr structure.

Transport Layer Raw Buffer and Message Monitoring

The Transport layer of the SIP Stack provides several events in the form of callback functions that are dedicated to inspecting and controlling the buffers that are sent or received by the SIP Stack sockets. The callback functions also inspect and control messages that are received by the transport layer and not yet processed. The events are implemented for all transports—UDP and TCP transports. The buffers contain encoded SIP *messages*. To get the events, an application should register to the events using the RvSipTransportMgrSetEvHandlers() function.

Raw Buffer Events

The following events are supplied with the Transport Layer API:

RvSipTransportBufferToSendEv()

Before the SIP Stack passes the buffer containing exactly one encoded SIP *message* to the "sending" system call, it calls this callback. Additional details, such as local and remote addresses for UDP sockets, or *connection* handles for TCP sockets, are provided with the callback. As a result of the details and buffer inspection, an application can instruct the SIP Stack to discard the buffer by updating the corresponding OUT parameter of the callback.

If the application did not register to the callback, the SIP Stack will send the buffer.

Note When discarding a buffer, the SIP Stack layers (Transaction, Call-leg, and so on) will behave as if the buffer was sent.

RvSipTransportBufferReceivedEv()

Just before parsing an incoming SIP *message* the SIP Stack gives the application the chance to look at the message raw buffer. The callback supplies a buffer with exactly one textual SIP message, the local and remote addresses of the message, and the *connection* handle if TCP was used. Based on the callback information, the application can instruct the SIP Stack to discard the buffer using an OUT parameter.

Note When TCP is used the buffer can be a result of several accumulated packets.

RvSipTransportMsgReceivedExtEv()

Notifies the application that a new SIP *message* was received and parced, but not yet processed.

RvSipTransportConnectionDataReceivedEv

Notifies the application that data was received on a *connection*. The data may not be a full message since a SIP *message* can arrive in multiple packets on TCP.

RvSipTransportConnectionParserResultEv()

The SIP Stack allows an application to control the quality of the *connection* in terms of successfully parsed messages. For each received buffer containing one SIP message, the SIP Stack reports the result of the buffer parsing to the application through this callback. The *connection* handle is provided with the callback. If the application decides about an unacceptable level of errors in messages that are received on the *connection*, it can terminate the *connection* using the Connection API.

15

WORKING WITH DNS

INTRODUCTION

SIP uses DNS procedures to allow a client to resolve a SIP Uniform Resource Identifier (URI) into the IP address, port, and transport protocol of the next hop to contact. The SIP Stack DNS feature is implemented in the Resolver and Transmitter layers of the SIP Stack and propagates upwards to all layers.

The DNS queries that the SIP Stack performs are non-blocking. This means that when the SIP Stack is waiting for a response from a DNS server, the SIP Stack can perform other operation, such as message sending, receiving and processing, call handling, and so on.

By default, the SIP Stack is compiled with basic DNS functionality that includes:

- A/AAAA queries to resolve IP addresses of hosts given as a domain name.
- NAPTR queries to resolve URI addresses, given as Tel URIs.

The SIP Stack enables a second mode of operation, an enhanced DNS mode. When using the enhanced DNS mode, the behavior of the SIP Stack will comply with the client DNS procedures defined in RFC 3263 and RFC 3824. This includes usage of NAPTR and SRV DNS queries, and the ability to maintain a list of resolved addresses that the application will be able to try one after the other, in case of send failure.

CONFIGURING DNS PARAMETERS

To work with DNS servers, the SIP Stack requires the following information:

- A list of DNS servers—tells the SIP Stack which DNS servers the SIP Stack should work with.

- A list of domain suffixes—tells the SIP Stack which domain suffixes should be appended to FQDNs. (For example, “hp.com” is the suffix for the “host1.hp.com” host.)

  Using suffixes allows using a short version of a name that is within the suffix domain.

The SIP Stack supports three ways of obtaining the information that is required. The information can be provided using the RvSipStackCfg configuration structure when initializing the SIP Stack. For more information, see RVSipStackCfg Configuration Structure of the Configuration chapter.

If no data (or partial data) is located in the configuration structure when the SIP Stack initializes, the SIP Stack will try to obtain the data from the operating system.

A third way is to provide (or change) that data during runtime. You can use the RvSipResolverMgrSetDnsServers() and the RvSipResolverMgrSetDnsDomains() functions to set the DNS servers and the list of DNS suffixes during runtime.

DNS/SRV TREE

RFC 3263 specifies the usage of three DNS record types:

- NAPTR—used to determine the transport for a specific domain.
- SRV—used to determine the port and transport of a specific SIP service.
- A (or AAAA for IPv6)—used to determine the IP/IPv6 address of a specific host.

The combination of the three types of records spans a tree, as illustrated in Figure 15-1. Each NAPTR query can result in several SRV records. An SRV query can result in several host records, each with several IP addresses (A/AAAA records).

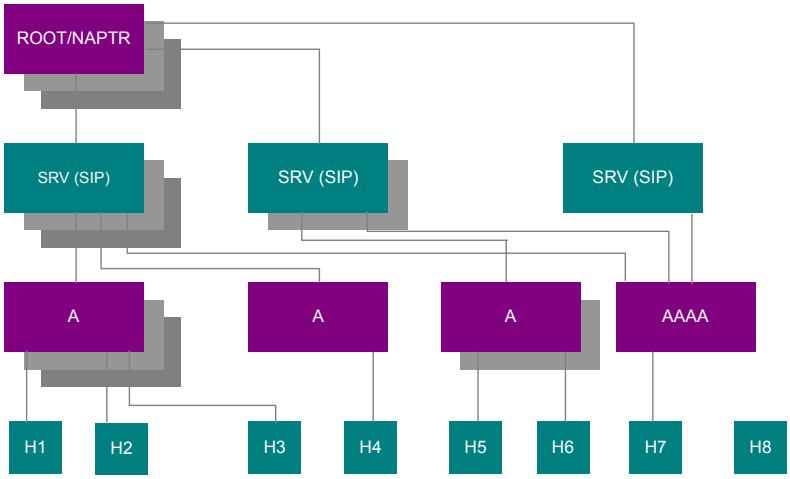


Figure 15-1 *A DNS/SRV Spanned Tree*

DNS/ENUM Record

RFC 3824 specifies the usage of ENUM NAPTR records for relating between E.164 numbers and URIs. For more information on ENUM, see the Advanced Features chapter.

SIP Stack Implementation

Before sending a request, the *transaction* obtains the destination host name from the request message. If the SIP Stack is compiled without Enhanced DNS features, it will try to resolve the host using a simple A/AAAA DNS, ENUM or NAPTR queries. If the SIP Stack is compiled with Enhanced DNS support, the SIP Stack will try to obtain the transport, port and IP of the host name as specified in RFC 3263 and RFC 3824.

The answers from the DNS server are kept in a DNS List object. The DNS List object holds the following sublists:

- SRV list—the result of the NAPTR query.
- Host list—the result of an SRV query applied on the first element of the SRV list. After the DNS SRV query is made, the first element in the SRV list is moved to the used SRV element member of the DNS list.

- IP addresses list—the result of an A/AAAA query applied on the first element of the Host list. After the A/AAAA query is made, the first element in the Host list is moved to the used host element member of the DNS list.

In addition, the DNS List object holds the host record used to resolve the IP and the SRV record used to resolve the port. Note that the SRV list, Host list and used SRV are only kept if the SIP Stack was compiled with the Enhanced DNS feature.

The records in each list are sorted by their priorities. Figure 15-2 illustrates a DNS List object.

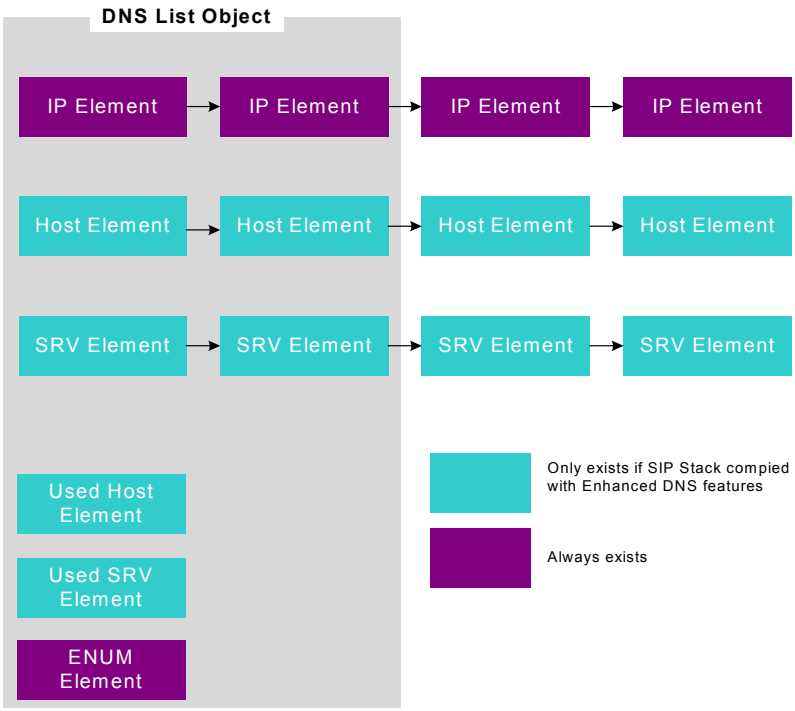


Figure 15-2 *DNS List Object*

When an ENUM query returns as a result of a valid URI, the Stack tries to resolve this URI automatically, using the general URI resolution methodology. During URI resolution, the following algorithm is used for filling the DNS list object.

For each element (such as, host, IP):

1. Remove (pop) the element from the list.
2. Query the DNS for that element, and insert the results into the DNS list.
3. Repeat steps 1 and 2 for the next sub-list until the IP sub-list is filled.

This means that when the DNS querying process has ended, every DNS sub-list (for example, hosts) will contain all the results retrieved from the DNS, excluding the result for which the next sub-list was constructed. Finally, the *transaction* will pop an IP from the IP sub-list and use it to send the request. To view the SRV element used to build the Hosts list, call RvSipTransportDNSListGetUsedSRVElement(). To view the host element used to build the IP addresses list, call RvSipTransportDNSListGetUsedHostElement().

The *transaction* sends a request to the first IP address found in the IP list. If a failure occurs, the application can choose to send the request to the next element in the list. The new request is identical to the previous, but has a different value of the Via branch and therefore constitutes a new SIP transaction.

The new *transaction* will obtain the DNS List object without the IP address that failed, and will send the request according to the supplied list. (A *transaction* always uses the first IP found on its DNS List object.)

Once all IP addresses for a specific host are exhausted, the *transaction* will pop a host from the host sub-list and the DNS will be queried for that host in the list. The same procedure applies when all the hosts are exhausted, in which case the DNS will pop the next SRV record and query the DNS for hosts, and so on.

Note Accessing DNS servers can be costly. Therefore, especially when the DNS caching mechanism is turned off, it is recommended to limit the number of retries using the *maxElementsInSingleDnsList* configuration parameter, or by manually manipulating the list. For more information, see DNS Caching.

State Machine and API Functions

In general, whenever a message send has failed, the *transaction* moves to a new state that indicates that failure, RVSIP_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE. In the state change callback, the reason that is attached to the state indicates the reason of the failure. The reason can be one of the following:

RVSIP_TRANSC_REASON_TIME_OUT

A *transaction* has timed out. (For example, an INVITE did not get a final response.)

RVSIP_TRANSC_REASON_NETWORK_ERROR

A network error was reported. (For example, a TCP connection could not be established.)

RVSIP_TRANSC_REASON_503_RECEIVED

A 503 response was received by the transaction.

In the RVSIP_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE state, the application can choose one of two actions:

- Try to use the next address in the list.
- Terminate the transaction.

Using the Next Address in the List

In order to send the request to the next address in the list, the application must start by calling the RvSipTransactionDNSContinue() function. Calling this function terminates the *transaction* and creates a new *transaction* that is ready to continue where the old *transaction* failed. The new *transaction* is a copy of the previous *transaction* with a new branch value and a copy of the DNS List (without the ENUM query result, if it exists, and the failed IP).

The application should then continue by calling the RvSipTransactionRequest() function on the newly created transaction. The *transaction* will send the request to the first IP address found on its DNS List object.

In the case where RVSIP_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE was reached because of a 503 response on INVITE, calling

RvSipTransactionDNSContinue() will not terminate the original transaction. The application must send ACK on the original *transaction* by calling the RvSipTransactionAck() function.

Note The application must first call RvSipTransactionDNSContinue() to copy the *transaction* and then call RvSipTransactionAck() to trigger an ACK on the original transaction.

TERMINATING THE TRANSACTION

To terminate the transaction, the application should call the RvSipTransactionTerminate() function.

Sample Code

The following sample code demonstrates how to implement a state changed callback. In this sample, the request message will be sent to the next address on the list upon failure.

```
/*-----------------------------------------------------------------------------------------*/
void RVCALLCONV AppTranscStateChangedEvHandler(
            IN RvSipTranscHandle             hTransaction,
            IN RvSipTranscOwnerHandle        pOwner,
            IN RvSipTransactionState         eNewState,
            IN RvSipTransactionStateChangeReason eReason)
{
  RvSipTranscHandle  hCloneTransc = NULL;
  RvChar              strMethod[512] = {'\0'};
  RvStatus            rv = RV_OK;
  RvSipAddressHandle  hUri = NULL;

  rv =  RvSipTransactionGetRequestUri(hTransaction,&hUri)
  if (rv != RV_OK)
  {
      /*Performs error handle routine.*/
  }
  rv = RvSipTransactionGetMethodStr(hTransaction, 512, strMethod);
  if (rv != RV_OK)
  {
      /*Performs error handle routine.*/
  }
```

```
  switch(eNewState)
  {
  case RVSIP_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE:
      {

           /*Creates a new "cloned" transaction.*/
           rv= RvSipTransactionDNSContinue(hTransaction, (RvSipTranscOwnerHandle)NULL,
                                                  &hCloneTransc);
           if(rv != RV_OK)
           {
               /*Performs the error handle routine.*/
           }
           /* 503 on INVITE will be acked!! */
           if (0 == stricmp(strMethod,"INVITE") &&
               RVSIP_TRANSC_REASON_503_RECEIVED == eReason)
           {
                rv = RvSipTransactionAck(hTransaction,NULL);
                if (rv != RV_OK)
                {
                     /*Performs the error handle routine.*/
                }
           }


           /*Requests on the new "cloned" transaction.*/
           rv = RvSipTransactionRequest(hCloneTransc, strMethod, hUri);
           if(rv != RV_OK)
           {
               /*Performs error handle routine.*/
           }
      } /* case RVSIP_TRANSC_STATE_CLIENT_MSG_SEND_FAILURE: */
  break;
  default:
  break;
  } /* switch(eNewState) */
}
/*=========================================================================================*/
```

Manipulating the DNS List Object

After calling the RvSipTransactionDnsContinue() function, the application receives a handle to the new cloned transaction. By calling the RvSipTransactionDNSGetList() function, the application can get the handle to the DNS List object (RvSipTransportDNSListHandle) and manipulate the list.

By manipulating the DNS list, the application can:

- Add records taken from internal source, rather than retrieved from an external DNS server.
- Remove unwanted records.

DNS List API

The DNS List API is part of the Transport API. In order to use the API, you need the following handles:

- *TransportMgr* handle (received from the RvSipStackGetTransportMgrHandle() function).
- DNS List object handle (received from the RvSipTransactionDNSGetList() function).

The following API functions are available:

RvSipTransportDNSGetEnumResult()

Gets the result of an ENUM NAPTR query.

RvSipTransportDNSListGetIPElement()

Gets an IP element from the list.

RvSipTransportDNSListRemoveTopmostIPElement()

Removes the top-most IP element.

RvSipTransportDNSListPopIPElement()

Pops the top-most IP element.

RvSipTransportDNSListPushIPElement()

Pushes an IP element to the list.

RvSipTransportGetNumberOfDNSListEntries()

Gets the number of elements in each sublist of the DNS List object.

SRV and Host Records Functions

Similar functions apply to SRV and Host records:

RvSipTransportDNSListGetXXXXElement()

RvSipTransportDNSListRemoveTopmostXXXXElement()

RvSipTransportDNSListPopXXXXElement()

RvSipTransportDNSListPushXXXXElement()

The insertion and extraction of records to and from the lists is done using the following structures:

RvSipTransportDNSSRVElement

For SRV elements.

RvSipTransportDNSHostNameElement

For hosts elements.

RvSipTransportDNSIPElement

For IP elements. In order to translate a string to an IP address, or an IP address to a string for the IP member of the structure, use RvSipTransportConvertStringToIp() and RvSipTransportConvertIpToString().

Sample Code

The following sample code demonstrates how to replace the top IP element of a transaction.

```
/*=========================================================================================*/
RvStatus ReplaceTopMostIp(IN RvSipTranscHandle        hTransaction,
                          IN RvSipTransportMgrHandle hTransportMgr)
{
      RvStatus                      rv = RV_OK;
      RvSipTransportDNSListHandle  hDnsList = NULL;
      RvChar                        strIP="172.3.4.45";
      RvSipTransportDNSIPElement   element;

      /*Obtains the DNS list from the transaction.*/
      rv = RvSipTransactionDNSGetList(hTransaction, &hDnsList);
      if (rv != RV_OK)
      {
           return rv;
      }

      /*Removes the topmost IP address from the list.*/
      rv = RvSipTransportDNSListRemoveTopmostIPElement (hTransportMgr,hDnsList)
      if (rv != RV_OK)
      {
           return rv;
      }

      /*Prepares the new IP element to include in the list.*/
      rv = RvSipTransportConvertStringToIp(
                      hTransportMgr,
                      strIP,
                      RVSIP_TRANSPORT_ADDRESS_TYPE_IP,
                      &element.ip);
      if (rv != RV_OK)
      {
           return rv;
      }
      element.protocol = RVSIP_TRANSPORT_UDP;
      element.port = 5060;
      element.bIsIpV6 = RV_FALSE;

      /*Pushes the element into the list.*/
      rv = RvSipTransportDNSListPushIPElement(hTransportMgr, hDnsList, &element);
      if (rv != RV_OK)
      {
```

```
            return rv;
        }
}
/*=============================================================================================*/
```

DNS Support for Call-legs, Subscriptions, and Register-Clients

As mentioned above, DNS support propagates upwards to objects that use the Transaction layer. Both *call-legs* and *subscriptions* have a state that indicates a Msg-Send-Failure and dedicated APIs that enable the application to manipulate the DNS List object and re-send the request to the next IP address.

Call-leg Layer

The RVSIP_CALL_LEG_STATE_MSG_SEND_FAILURE state indicates a message send failure. The application can obtain the DNS List object handle by calling the RvSipCallLegDNSGetList() function. Once the application has a handle to the list, it can manipulate the list using the transport API.

Call-leg DNS API

Calling RvSipCallLegDNSContinue() causes the SIP Stack to prepare an internal cloned *transaction* and terminate the original transaction. If the application wishes, it can call RvSipCallLegGetOutboundMsg() and manipulate the message before sending it. To actually try and send the message to the next address in the DNS list, the application should call the RvSipCallLegDNSReSendRequest() function.

Calling RvSipCallLegDNSGiveUp() indicates that the application does not want to keep trying to send the message, and will terminate the call leg. *Call-leg transactions* use the same API function with the specific *transaction* handle. For more information on these functions, see the *Call-leg DNS Functions* section in the *Call-leg Function*s chapter of the *SIP Stack Reference Guide*.

Subscriptions Layer

The RVSIP_SUBS_STATE_MSG_SEND_FAILURE state indicates a message send failure. The application can obtain the DNS List object handle by calling the RvSipSubsDNSGetList() API function. Once the application has a handle to the list, it can manipulate the list using the transport API.

Subscription DNS API

Calling RvSipSubsDNSContinue(), causes the SIP Stack to prepare an internal cloned *transaction* and terminate the original transaction. To actually try and send the message to the next address in the DNS list, the application should call the RvSipSubsDNSReSendRequest() function. Calling RvSipSubsDNSGiveUp() indicates that the application does not want to keep trying to send the message. For more information, see the *Subscription DNS API* section in the *Event Notification Functions* chapter of the *SIP Stack Reference Guide.*

Register-Client Layer

The RVSIP_REG_CLIENT_STATE_MSG_SEND_FAILURE state indicates a message send failure. The application can obtain the DNS List object handle by calling the RvSipRegClientDNSGetList() function. Once the application has a handle to the list, it can manipulate the list using the Transport API.

Register- Client DNS API

Calling the RvSipRegClientDNSContinue() function causes the SIP Stack to prepare an internal cloned *transaction* and terminate the original transaction. If the application wishes, it can call RvSipRegClientGetOutboundMsg() and manipulate the message before sending it. To actually try and send the message to the next address in the DNS list, the application should call the RvSipRegClientDNSReSendRequest() function. Calling RvSipRegClientDNSGiveUp() indicates that the application does not want to keep trying to send the message, and will terminate the register-client.

Changing DNS List before Message is Sent

Each SIP Stack object (*call-leg*, register-client, and so on) has an RvSipXXXXFinalDestResolvedEv() callback. In this callback, it is possible to obtain the DNS list and manipulate it. You can use RvSipTransactionGetTransmitter() to obtain the *transmitter* of the sending *transaction* and perform different operations on the transmitter, as follows:

1. Call RvSipTransmitterDNSGetList() to obtain the DNS list held by the transmitter.
2. Call the functions described in Manipulating the DNS List Object to manipulate the list.
3. You can also call RvSipTransmitterSetDestAddress() to implicitly set the address to which the message will be sent.

To change the DNS list asynchroniously:

1. Call RvSipTransmitterHoldSending() to hold the sending of the message. While the SIP Stack holds the message, you can perform DNS manipulation on the DNS list held by the transmitter.
2. Once you are done with DNS manipulation, you can call RvSipTransmitterResumeSending() to complete the message sending.

For more information, see the Working with Transmitters chapter.

DNS Caching

The SIP Stack can be configured to support DNS caching. When DNS caching is enabled, positive and negative DNS answers are kept in the caching module for further usage. The module caches positive answers from the DNS server according the TTL value found in the DNS response.

DNS Caching Compilation Flags

Several compilation flags are used to configure the DNS caching feature. All the compilation flags are located in the *common/config/rvusrconfig.h* file and are described below:

RV_DNS_USES_CACHING

To enable caching, the RV_DNS_USES_CACHING compile time constant should be set to RV_YES.

RV_DNS_CACHE_HASH_SIZE

Cache lookup is implemented using static open hashing. The size of hash table is given by the compile time constant, RV_DNS_CACHE_HASH_SIZE.

RV_DNS_CACHE_PAGE_SIZE

All memory needed for caching is pre-allocated statically on initialization. The actual data is kept on memory pages. The page size (in bytes) is determined by the compile time constant, RV_DNS_CACHE_PAGE_SIZE, and should be >= 512. If the page size is too small to accommodate a single DNS record, this record will not be cached.

DNS_CACHE_PAGES_NUMBER

The number of pages used for caching data is given by RV_DNS_CACHE_PAGES_NUMBER. There is no lower limit on the number of pages, but if the number of pages is too low, new records will not be cached.

RV_DNS_USES_HOSTS

When DNS caching is enabled, the DNS module can be configured to look for host name resolution in the operating system host file (*/etc/hosts* for example). To instruct the SIP Stack to search the host file, the RV_DNS_USES_HOSTS should be set to RV_TRUE.

16

WORKING WITH RESOLVERS

INTRODUCTION

The SIP Stack uses resolvers to produce data that is related to DNS. A resolver object (*resolver*) is responsible for retrieving a piece of DNS information. In some cases this information can be obtained by a single DNS query (such as NAPTR resolution), in other cases several queries are required (such as host resolution when the network type, <IPv4 or IPv6>, is unknown).

The SIP Resolver is used mainly by the *transmitter* to implement RFC 3263: Session Initiation Protocol (SIP): Locating SIP Servers. This RFC describes the ways to retrieve the transport of the destination, IP address and port. The destination details can be obtained separately or in a single query by calling the Resolvers API functions. The Resolver layer, which is exposed to the application, can be used for connection reuse and mutual TLS, when DNS operations are needed.

RESOLVER ENTITIES

The resolver layer has two entities:

- Resolver (*resolver*)
- Resolver Manager (*ResolverMgr*)

RESOLVER

A *resolver* is responsible for obtaining one data element according to the RFC 3263 algorithm. The answers from the DNS are stored in the a DNS list object. For more information, see the Working with DNS chapter.

The DNS data element types correspond to the following RvSipResolverMode enumeration components:

RVSIP_RESOLVER_MODE_UNDEFINED

Represents no-resolution mode.

RVSIP_RESOLVER_MODE_FIND_TRANSPORT_BY_NAPTR

Instructs the resolver to try obtaining NAPTR records for a domain. The resolver will use the protocol field of the NAPTR record to determine the transport, for example, getting the available transports for hp.com. The result will be a list of SRV record pointers.

RVSIP_RESOLVER_MODE_FIND_TRANSPORT_BY_3WAY_SRV

Instructs the resolver to try finding a transport for a domain by applying SRV queries of <_service>._<protocol>.<domain>, until the resolver receives a successful answer. For example, the resolution of sip:aaa@hp.com will include queries in the sip._<protocol>.hp.com format, first trying UDP, then TCP, and finally TLS. If the resolver gets a positive answer, it will stop querying, and as a result, provide a host pointer and a port number.

RVSIP_RESOLVER_MODE_FIND_HOSTPORT_BY_SRV_STRING

Instructs the resolver to send one SRV query with the string supplied and to save the host and port retrieved in the answer in the DNS list, for example, trying to get the host and port for _sip._udp.hp.com.

RVSIP_RESOLVER_MODE_FIND_HOSTPORT_BY_TRANSPORT

Instructs the resolver to create one service._<protocol>.domain query string and try sending an SRV query for it. The host and port retrieved in the answer will be stored in the DNS list. For example, trying to get the TCP/pres SRV record for hp.com will send a _pres._udp.hp.com SRV query and will result in a host pointer and a port number.

RVSIP_RESOLVER_MODE_FIND_IP_BY_HOST

Instructs the resolver to try finding the IP of a specific host. The resolver will try both IPv4 and IPv6 (using A and AAAA queries respectively). For example, when trying to get the IP for host1.hp.com, the result will be an IP address.

RVSIP_RESOLVER_MODE_FIND_URI_BY_NAPTR

Gets the NAPTR record for an ENUM record in a DNS server. For example, trying to resolve a phone number, such as +97237679623, will result in an NAPTR query, for 3.2.6.9.7.6.7.3.2.7.9.e164.arpa..The result of this kind of query will be a regular expression.

Resolver Manager

The *ResolverMgr* manages the collection of resolvers and is mainly used for creating new resolvers. The *ResolverMgr* takes part in the address resolution process by holding a list of DNS servers that each of the resolvers will contact. The application can use the Resolver Manager API to change this list.

Working with Handles

All *resolvers* and the *ResolverMgr* are identified using handles. You must supply these handles when using the Resolver API. RvSipResolverMgrHandle defines the *ResolverMgr* handle. You receive this handle by calling RvSipStackGetResolverMgrHandle().

RvSipResolverHandle defines a *resolver* handle. You receive the Resolver handle when creating a resolver with RvSipResolverMgrCreateResolver().

Resolver API

The Resolver API contains a set of functions and function callbacks that allow you to control resolver functionality.

Resolver Control

The following Resolver API functions provide resolver control:

RvSipResolverResolve()

Starts a DNS algorithm mechanism. After calling RvSipResolverResolve(), the resolver will try to obtain data from the DNS servers specified in the SIP Stack. The retrieved data will be stored in the *hDns* parameter supplied by the caller. Using the *eMode* parameter, you can specify the algorithm session type that the resolver will try to accomplish. The *strQueryString* parameter should contain the base string for the DNS query. The *eScheme* parameter specifies the scheme to concatenate to the *strQueryString* parameter in case that SRV is applied for transport. You can set *bIsSecure* to indicate that any record that does not represent a secure connection will be discarded. The *knownPort* and *knownTransport* parameters are used when a record retrieved from the DNS server does not contain the data (for example, an A record does not contain port or transport, and SRV records do not contain transport). The *pfnResolveCB* parameter indicates which callback the resolver should call when the DNS algorithm session has ended.

Once a resolution process has ended, the resolver can be reused and the answer for the next query will be stored in the same DNS list.

Resolver Callbacks

The application has to specify an RvSipResolverReportDataEv() callback when activating the resolver (by calling RvSipResolverResolve()). This callback will pop whenever the DNS algorithm session has ended, and will indicate if the algorithm session was successful or not.

After the callback has popped, the DNS list that you supplied the resolver should contain the answers. You can obtain the list by using the RvSipResolverGetDnsList() function. Once you are done using the resolver, you can call RvSipResolverTerminate() to terminate the *resolver*. You will need to destruct the DNS list manually.

Resolver Manager Functions

The *ResolverMgr* controls the SIP Stack collection of resolvers. You use the Resolver Manager API to create new *resolvers*. In addition, the *ResolverMgr* holds the list of DNS servers that the *resolvers* will contact to resolve DNS names.

The following Resolver Manager API functions are provided:

RvSipResolverMgrCreateResolver()

Creates a new resolver and exchanges handles with the application. As soon as the function returns a valid status code, the new *resolver* is ready for use.

RvSipResolverMgrSetDnsServers()/RvSipResolverMgrGetDnsServers()

Sets a new list of DNS servers to the *ResolverMgr*. In the process of address resolution, host names are resolved to IP addresses by sending DNS queries to the SIP Stack DNS servers. If one server fails the next DNS server on the list is queried. When the SIP Stack is constructed, a DNS servers list is set to the SIP Stack using the computer configuration. The *ResolverMgr* keeps the list. The application can use the RvSipResolverMgrSetDnsServers() function to provide a new set of DNS servers to the *ResolverMgr*. It can use the RvSipResolverMgrGetDnsServers() function to get the current DNS servers list.

RvSipResolverMgrSetDnsDomains()/RvSipResolverMgrGetDnsDomains()

Sets a new list of DNS domains to the *ResolverMgr*. The SIP Stack provides Domain Suffix Search Order capability, which specifies the DNS domain suffixes to be appended to the host names during name resolution. When attempting to resolve a fully qualified domain name (FQDN) from a host that includes a name only, the system will first append the local domain name to the host name and query DNS servers. If this is not successful, the system will use the Domain Suffix list to create additional FQDNs in the order listed and will query DNS servers for each FQDN. When the SIP Stack initializes, the DNS

domain list is set according to the configuration of the computer and is managed by the *ResolverMgr*. The application can use the RvSipResolverMgrSetDnsDomains() function to provide a new set of DNS domains to the SIP Stack. It can use the RvSipResolverMgrGetDnsDomains() function to get the current list of DNS domains.

Sample Code

The following sample code illustrates the use of a *resolver* with a DNS list.

```
/*=========================================================================================*/
      /* Global parameters*/
      static  RvSipTransportMgrHandle hTransportMgr;
      static  RvSipResolverMgrHandle hResolverMgr;
      static  HRPOOL hPool;


      /* A callback to process answers */
      RvStatus RVCALLCONV ReportDataEvHandler(
            IN RvSipResolverHandle          hResolver,
            IN RvSipAppResolverHandle          hOwner,
            IN RvBool                          bError,
            IN RvSipResolverMode               eMode)
      {
        RvStatus                              rv = RV_OK;
        RvSipResolverGetDnsList hDNSlist;

        if (error)
          DoErrorActions();
        else
        {
          RvChar ip[60];
          RvSipTransportDNSIPElement    IPElement;
          RvSipResolverGetDnsList(hResolver,&hDNSlist);
          RvSipTransportDNSListPopIPElement(hTransportMgr,hDNSlist,&IPElement);
          RvSipTransportConvertIpToString(hTransportMgr,IPElement.ip,RVSIP_TRANSPORT_ADDRESS_
                                          TYPE_IP,60,ip);
          printf("the ip of host.com is %s", ip);
        }
        return RV_OK;
      }
```

```
      void ResolveHostToIp()
      {
        RvStatus                        rv              = RV_OK;
        RvSipResolverHandle             hStackResolver  = NULL;
        RvSipTransportDNSListHandle     hDNSlist;

        /* 1. Building a DNS list*/
        rv = RvSipTransportDNSListConstruct(hTransportMgr,hPool,10,&hDNSlist);
        if (RV_OK != rv)
          HandleError();

        /* 2. Creating a resolver */
        rv = RvSipResolverMgrCreateResolver(hResolverMgr,(RvSipAppResolverHandle)NULL,
                                            &hStackResolver);
      if (RV_OK != rv)
      HandleError();

        /* 3. Trying to resolve the host "host.com" to IP */
        rv = RvSipResolverResolve(hStackResolver,
                                  RVSIP_RESOLVER_MODE_FIND_IP_BY_HOST,
                                  "host.com",
                                  RVSIP_RESOLVER_SCHEME_UNDEFINED,
                                  5060,
                                  RVSIP_TRANSPORT_UDP,
                                  hDNSlist,
                                  ReportDataEvHandler);
        if (RV_OK != rv)
          HandleError();

      }
/*=========================================================================================*/
```

17

SIP STACK LOG

INTRODUCTION

The SIP Stack includes a logging module that produces output for debugging, monitoring and tracking of the activity of applications built with the SIP Stack.

The Log output provides detailed information important for application development and ongoing system maintenance. Suggestions for using the Log include:

- **Post Mortem Monitoring**

  If you want to know the full history of calls over your network, use the Log output to monitor all network events related to the SIP Stack.

- **Problem Tracking**

  The Log provides a chronological record of the SIP Stack activities that enables you to track exactly where and when a problem occurred.

- **Reduce Debugging Costs**

  By using the Log as a debugging tool for applications built on top of the SIP Stack, you can save valuable programmer time and reduce development costs.

SOURCE IDENTIFIERS

The Log uses various source identifiers for monitoring SIP Stack behavior. You can set up the Log to analyze any of the source identifiers and specify the level of logging required for each one independently. Table 17-1 shows the main source identifiers that the Log analyzes:

Table 17-1 *Source Identifiers Analyzed by the Log*

Source Identifiers	Description
CALL	*Call-leg* related operations.
AUTHENTICATOR	*Authenticator* related operations.
REG_CLIENT	*Register-Client* related operations.
TRANSACTION	*Transaction* related operations.
TRANSMITTER	*Transmitter* related operations.
MESSAGE	SIP *message* related operations.
TRANSPORT	*Transport* related operations.
PARSER	SIP Message Parser related operations.
STACK	SIP Stack initialization, configuration and termination related operations.
MSGBUILDER	SIP Message-Builder related operations.
SUBS	*Subscription* related operations.

A full list of available source identifiers can be found in the RvSipStackModule enumeration.

LOG FILE

By default, the Log is written to a file called *SipLog.txt*.

LOG MESSAGES

Log messages are printed to the log file along with information about the message type. The following Log message types are available:

- **INFO—**describes SIP Stack activity.
- **DEBUG**—provides detailed information about SIP Stack activity.

- **WARN**—warning about a possible non-fatal error.
- **ERROR**—indicates that a non-fatal error has occurred, such as faulty application behavior, insufficient allocations or illegal network activity.
- **EXCEP**—indicates that a fatal error has occurred that prevents the SIP Stack from continuing to operate.
- **LOCKDBG—**debug information about locking activities in the different modules of the SIP Stack. If your application is not multithreaded it is recommended not to use LOCKDBG messages. These messages have no benefit for such applications and they increase the log file size significantly.
- **ENTER**—A message that is printed to the log each time a Core or ADS function is called. By default it is not recommended to use Enter messages that increase the log file significantly.
- **LEAVE**—A message that is printed to the log just before leaving a Core or ADS API function. By default it is not recommended to use Leave messages that increase the log file significantly.

LOG CONFIGURATION

The RvSipStackCfg configuration structure contains parameters that enable configuring the log for each of the source identifiers. For example, the *callLogFilters* configuration parameter will set the log level for the CALL source identifier. You can also configure the log level to groups of source identifiers using the following configuration parameters:

- defaultLogFilters—the log level that will apply to all the SIP Stack modules including the SIP Stack, ADS and CORE modules.
- coreLogFilters—the log level that will apply to all CORE modules overriding the default log level.
- adsLogFilters—log level that will apply to all ADS modules.

These parameters are used only if they contain a non-zero value that overrides the default log level.

Note The specific sources of the CORE module are placed in the RvSipStackCoreLogFiltersCfg structure. The specific sources of the ADS module are placed in the RvSipStackAdsLogFiltersCfg structure. Both structures are part of the RvSipStackCfg structure.

After setting the log level to the above groups of source identifiers, the *StackMgr* will check if any of the specific sources was set to a non-zero log level. If so, this level will be set for the specific source overriding the default log level. The RvSipLogFilters enumeration contains a filter for each of the log message types.

The following filters are available:

- RVSIP_LOG_DEBUG_FILTER
- RVSIP_LOG_INFO_FILTER
- RVSIP_LOG_WARN_FILTER
- RVSIP_LOG_ERROR_FILTER
- RVSIP_LOG_EXCEP_FILTER
- RVSIP_LOG_LOCKDBG_FILTER
- RVSIP_LOG_ENTER_FILTER
- RVSIP_LOG_LEAVE_FILTER

You can combine these filters to define exactly what log messages each source will produce. The log can be configured on the SIP Stack construction using the configuration structure, or at runtime using the RvSipStackSetNewLogFilters() API function.

Sample Code

The following code demonstrates how to instruct the Call-leg layer to print only INFO and ERROR messages to the Log.

```
/*=========================================================================*/
RvSipStackCfg stackCfg;
RvSipStackInitCfg(sizeof(stackCfg), &stackCfg));
stackCfg.callLogFilters = RVSIP_LOG_INFO_FILTER | RVSIP_LOG_ERROR_FILTER;
/*=========================================================================*/
```

Sample Code

The following code demonstrates how to set the Log filters for all layers of the SIP Stack simultaneously. In this sample, each layer is instructed to print INFO, DEBUG, ERROR, WARN and EXCEP messages to the Log. This is also the recommended set of prints that will enable you to fully monitor the Stack activities.

```
/*=========================================================================*/
```

```
RvSipStackCfg stackCfg;
RvSipStackInitCfg(sizeof(stackCfg), &stackCfg);
stackCfg.defaultLogFilters = RVSIP_LOG_INFO_FILTER | RVSIP_LOG_DEBUG_FILTER |
                             RVSIP_LOG_ERROR_FILTER | RVSIP_LOG_WARN_FILTER |
                             RVSIP_LOG_EXCEP_FILTER;
/*=========================================================================*/
```

MESSAGE STRUCTURE

The default Log contains rows of messages. Each row consists of the following fields:

```
<thread id><date> <time> <message type> <source identifier>
<log message>
```

The following is an example of a message row:

```
T 00000d50 07/15/03 12:18:49 INFO   - CALL         -
RvSipCallLegAccept - Accepting call-leg 0x01F9D7C0
```

The fields in a message row represent the following:

- thread id—the thread that is currently running and printing to the log.
- date—the date on which the logged event took place.
- time—the time at which the logged event took place.
- message type—see Log Messages for a list of message types.
- source identifier—see Source Identifiers for a list of the SIP Stack source identifiers analyzed by the Log.
- log message—the actual Log message item.

CONTROLLING THE LOG

You can control both the Log output device (file, network or console) and the Log message structure by registering your own Log callback function to the SIP Stack. Each time the SIP Stack wants to add a line to the Log, your callback function will be called instead of the default Log function. You register your callback function during the SIP Stack initialization process, as shown in The following code.

Sample Code

The following code demonstrates how to replace the default log function with a user callback function. The user call back function prints the Log to a file named *MyLog.txt* and adds a line number before each line in the Log.

```
/*=========================================================================*/
RvSipStackCfg stackCfg;
RvSipStackHandle hStack;

RvStatus AppInitSipStack()
{
      FILE *logFile;

      RvSipStackInitCfg(sizeof(stackCfg), &stackCfg));

      /*Open the log file for writing*/
      logFile = fopen("MyLog.txt","w");

      /*Register the log function in the configuration structure*/
      stackCfg.pfnPrintLogEntryEvHandler = AppLogPrintFunction;

      /*Save the file pointer as a log context (optional)*/
      stackCfg.logContext = logFile;

      return RvSipStackConstruct(sizeof(stackCfg), &stackCfg, &hStack);
}
/*=========================================================================*/
/*The log callback function*/
void RVCALLCONV AppLogPrintFunction(
                          IN void*           context,
                          IN RvSipLogFilters filter,
                          IN const RvChar    *formattedText)
{
    char data[1500];
    static int line=1;
    FILE *logFile = (FILE*)context;

    /*Add a line number to the log*/
    sprintf(data,"%8d:", line);
    line ++;

   /*Add the log text*/
    strcat(data,formattedText);
    strcat(data,"\n");

    /*Write the log to the log file*/
```

```
            fwrite(data, strlen(data), 1, logFile);
            fflush(logFile);
        }
/*=============================================================================*/
```

Note When you replace the default logging, the formatted text includes only the message type, source identifier, and log message.

Compilation Log Control

The application can control the compiled log messages by setting the "RV_LOGMASK" compilation flag found in the *rvusrconfig.h* file to one, or a combination of the following definitions:

- RV_LOGLEVEL_EXCEP
- RV_LOGLEVEL_ERROR
- RV_LOGLEVEL_WARNING
- RV_LOGLEVEL_INFO
- RV_LOGLEVEL_DEBUG
- RV_LOGLEVEL_ENTER
- RV_LOGLEVEL_LEAVE
- RV_LOGLEVEL_SYNC
- RV_LOGLEVEL_ALL
- RV_LOGLEVEL_NONE

For example:

- `#define RV_LOGMASK RV_LOGLEVEL_ALL`—will enable all logging.
- `#define RV_LOGMASK RV_LOGLEVEL_NONE`—will remove all logging.
- `#define RV_LOGMASK RV_LOGLEVEL_EXCEP+RV_LOGLEVEL_ERROR`—will enable only exception and error logs.

The amount of log lines in the compiled code has a tremendous effect on the code size of your application. For more information, see the Reducing Footprints chapter.

18

MEMORY POOL

INTRODUCTION

The SIP Stack does not allocate memory dynamically. All memory is allocated during the initialization process and is managed by the SIP Stack. The memory is divided into blocks called pages. The page size and the number of pages are configurable. The collection of all pages is called a memory pool. The memory pool manages the pages, supplying a simple API that allows the user to receive and recycle memory bytes when needed. The memory pool can supply and manage both consecutive and non-consecutive memory. In this way, memory is managed more efficiently.

The SIP Stack uses different memory pools for different needs. Each of these memory pools defines different page sizes and page numbers. Some of the API functions of the SIP Stack require application memory allocation. These functions must receive a valid memory pool as a parameter, and must sometimes also receive a memory page as a parameter. To use these functions, your application must define a memory pool according to your specific requirements.

DEFINITIONS

You use handles to reference memory pools and memory pages. HRPOOL defines a memory pool handle. HPAGE defines a memory page.You can find both definitions in *rpool_API.h*.

API FUNCTIONS

You can find the API functions of the memory pool in the file, *rpool_API.h*. The Memory Pool API includes the following functions:

RPOOL_Construct()

Constructs a new memory pool. The memory for the new pool is allocated when calling this function. This function receives parameters that indicate the number of pages and the size of each page in the newly constructed pool.

RPOOL_Destruct()

Destroys a memory pool. Memory allocated to the pool is freed.

RPOOL_GetPage()

Gets one page from the memory pool.

RPOOL_FreePage()

Releases a given page. This function returns the released page to the memory pool.

RPOOL_CopyToExternal()

Copies a given number of bytes from a given page into a given buffer.

RPOOL_AppendFromExternalToPage()

Copies a given number of bytes from a given buffer onto a given page. The function output is the offset of the string on the page.

RPOOL_Strlen()

Returns the length of a NULL terminated string located on the specified page in a specified offset.

Note RPOOL_CopyToExternal() copies non-consecutive memory into a consecutive buffer. RPOOL_AppendFromExternalToPage() copies a consecutive buffer to a non-consecutive memory.

USING THE MEMORY POOL

The sample codes in this section demonstrate how to use the SIP Stack memory pool.

CONSTRUCTING AND DESTRUCTING A MEMORY POOL

The following code demonstrates how to construct and destruct a memory pool and how to get a new page for using in calls to SIP Stack API functions.

Sample Code

```
/*=========================================================================*/
void AllocatePageExample()
{
     RvStatus retStatus;

     /*Defines a memory pool handle.*/
     HRPOOL appPool;

     /*Defines a memory page handle.*/
     HPAGE appPage;

     /*Constructs a new memory pool with 10 pages and 20 bytes on each page.*/
     appPool = RPOOL_Construct(20, 10, NULL, RV_TRUE, "My application");

     if (appPool == NULL)
     {
          printf("Error constructing memory pool");
     }

     /*Gets a memory page from the memory pool. appPage is the handle to this page.*/
     retStatus = RPOOL_GetPage(appPool, 0, &appPage);

     if (retStatus != RV_OK)
     {
          printf("Error getting page from the memory pool");
     }

     /*Calls the SIP Lite Stack API function with appPool and appPage as parameters.*/

     /*Frees the appPage page.*/
     RPOOL_FreePage(appPool, appPage);

     /*Destructs the appPool memory pool.*/
     RPOOL_Destruct(appPool);
}
/*=========================================================================*/
```

Note You can construct the application memory pool when you initialize your application and keep using the memory pool until the application terminates.

COPYING A PAGE INTO A BUFFER

The following code demonstrates how to copy the content of a page into a consecutive buffer.

Sample Code

```
/*=========================================================================================*/
RvStatus PrintPageContent(IN HRPOOL    memoryPool,
                          IN HPAGE     memoryPage)
{
      RvStatus retStatus;

      /* Defines a consecutive buffer of size 21.*/
      RvChar  memoryString[21];

      /* Copies 20 bytes from the memoryPage page to the memoryString buffer. The
      copying begins at the beginning of the page. Note that the memoryPage page
      belongs to the memoryPool memory pool.*/

      retStatus = RPOOL_CopyToExternal(memoryPool, memoryPage, 0, memoryString, 20);
      if (retStatus != RV_OK)
      {
               return retStatus;
      }
      memoryString[20] = '\0';

      /* Prints the memory string to screen.*/
      printf("The memory string is %s", memoryString);

      return RV_OK;
}
/*=========================================================================================*/
```

COPYING A BUFFER TO A PAGE

The following code demonstrates how to copy one buffer to another buffer using a memory page.

Sample Code

```
/*=========================================================================================*/
RvStatus CopyBuffersWithPage(IN HRPOOL memoryPool,
                             IN HPAGE  memoryPage)
{
      RvStatus  retStatus;
      RvInt32   offset, size;
      RvChar    source[50], dest[50];

      /*Sets the source buffer.*/
      strcpy(source,"Hello, How are you?");
      size = strlen(source)+1;

      /*Copies the source buffer to the page.*/
      retStatus = RPOOL_AppendFromExternalToPage memoryPool,memoryPage, source, size,& offset)
      if(retStatus != RV_OK) return RV_ERROR_UNKNOWN;

      printf("%s - was added to the page at offset %d\n",source,offset);

      /*Copies information from the page to the destination buffer.*/
      retStatus = RPOOL_CopyToExternal(memoryPool,memoryPage,offset, dest,size);
      if(retStatus != RV_OK) return RV_ERROR_UNKNOWN;
      printf("the dest buffer contains - %s\n",dest);
      return RV_OK;
}
/*=========================================================================================*/
```

19

ADVANCED FEATURES

EXTENSION SUPPORT

SIP provides a mechanism for allowing a client to request that a particular protocol extension be used to process a request. The client uses either the Require or Proxy-Require headers to indicate the requested extensions. Extensions are identified using “option-tags,” which are unique string names defined in the extension specifications.

The server declines the request with a 420 (Bad Extension) response message, if the server does not support the requested extension, and includes the set of unsupported option-tags in an Unsupported header. Clients and servers advertise their extension support using the Supported header (which also contains option-tags).

The SIP Stack provides an easy way to use this mechanism using the following configuration parameters found in the configuration structure, RvSipStackCfg:

- **supportedExtensionList**

  Specifies the SIP Stack supporting ability. It is a list of supported option-tags, separated by commas, which are supported by the SIP Stack.

- **addSupportedListToMsg**

  Indicates whether or not the SIP Stack should add the supported header with the list in the supportedExtensioinList to every request and response except ACK.

- **rejectUnsupportedExtensions**

  Indicates whether or not the *call-legs* should reject requests that include a Require header with an option-tag not supported by the SIP Stack. When configured to RV_TRUE, if the server

does not recognize the option-tag on the Require header, the server responds with a 420 response message and adds an Unsupported header with the unsupported option-tag.

For more information, see the Configuration chapter.

Sample Code

The following code configures the SIP Stack to support two extensions:

- PRACK—(reliable provisional responses) specified by the 100rel option tag
- MyExtn—(a proprietary extension) specified by the MyExt option tag.

The SIP Stack is configured to reject unsupported extensions.

```
/*=========================================================================*/
static RvStatus AppStackInitialize(OUT RvSipStackHandle *phStack)
{
        RvStatus rv;
        RvSipStackCfg stackCfg;
        RvSipStackInitCfg(sizeof(stackCfg),&stackCfg);

        /*Configures the Stack to support 100rel and MyExt*./
        stackCfg.supportedExtensionList = "100rel,MyExt";
        /*Configures the Stack to add the supported list to messages.*/
        stackCfg.addSupportedListToMsg = RV_TRUE;
        /*Configures the Stack to reject unsupported extensions.*/
        stackCfg.rejectUnsupportedExtensions = RV_TRUE;
        /*Calls the Stack initialization function.*/
        rv = RvSipStackConstruct(sizeof(stackCfg),&stackCfg,phStack);
        return rv;
}
/*=========================================================================*/
```

The *rejectUnsupportedExtension* configuration parameter influences the behavior of *call-legs* only. If you work with the Transaction API, you can use the function, RvSipTransactionIsUnsupportedExtRequired(). If the result is RV_TRUE, you may reject the request using the 420 response code. In such a case, the Unsupported header will be added to the reject message automatically.

Sample Code

The following sample code demonstrates how to identify and reject a request with an unsupported extension when working with the Transaction layer. The request is handled in the context of the *transaction* state changed callback.

```
/*============================================================================*/
void RVCALLCONV AppTransactionStateChangedEvHandler(
                IN RvSipTranscHandle hTransc,
                IN RvSipTranscOwnerHandle hAppTransc,
                IN RvSipTransactionState eState,
                IN RvSipTransactionStateChangeReason eReason)
{
  RvBool bIsUnsupportedRequired = RV_FALSE;
  switch(eState)
  {
  case RVSIP_TRANSC_STATE_SERVER_GEN_REQUEST_RCVD:
         RvSipTransactionIsUnsupportedExtRequired(hTransc,&bIsUnsupportedRequired);
         if(bIsUnsupportedRequired == RV_TRUE)
         {
                RvSipTransactionRespond(hTransc,420,NULL);
         }
         else
         {
                RvSipTransactionRespond(hTransc,200,NULL);
         }
         break;
  default:
         break;
  }
}
/*============================================================================*/
```

MULTIHOMED HOST

The Multihomed Host feature allows sending and receiving requests from different local IP addresses. This feature can be used to send requests from different network cards or from the same network card with several different ports. This feature also enables the dynamic opening and closing of the local IP addresses, which are used for request sending and reception, at any moment of the SIP Stack life cycle. This sub-feature is called Dynamic Local Addresses (DLA).

It is possible to set unlimited UDP, TCP and TLS addresses, using the following configuration parameters found in the configuration structure, RvSipStackCfg:

- localUdpAddresses
- localUdpPorts
- localTcpAddresses
- localTcpPorts
- numOfExtraUdpAddresses
- numOfExtraTcpAddresses
- numOfTlsAddresses
- localTlsAddresses
- localTlsPorts

The SIP Stack will wait for incoming SIP messages on the addresses that are set.

Note When listening to several TCP or TLS addresses, the *maxConnection* configuration parameter should be increased accordingly.

MULTIHOMED HOST API FUNCTIONS

An API function is provided in each of the SIP Stack layers, allowing you to define the address from which the requests are sent. The following API functions are provided:

- RvSipCallLegSetLocalAddress()
- RvSipTransactionSetLocalAddress()
- RvSipRegClientSetLocalAddress()
- RvSipTransmitterSetLocalAddress()

Each of these functions receives five parameters:

- Specified object handle
- Transport type of the local address you wish to set (UDP/TCP/TLS)
- Address type of the local address you wish to set (IPv4/IPv6)

- Local IP address as a string
- Local port

When a request is sent, the local address is chosen according to the destination transport and address types. For example, if a request destination is a TCP IPv6 address, the local address to use is also a TCP IPv6 address. Therefore, each of the SIP Stack objects holds several local addresses for each combination of transport and address types.

If you wish to set the local address of several transports, you should call the RvSipXXXSetLocalAddress() function several times for each requested combination. If, for example, a *call-leg* sends its initial request with TCP/IPv4 and further requests with UDP/IPv4, you may want to set both the TCP/IPv4 and the UDP/IPv4 local addresses.

Note The local address string you provide for the Set function must exactly match the local address that was inserted in the configuration structure at the initialization of the SIP Stack. If you configured the SIP Stack to listen to a 0.0.0.0 local address, you must use the same notation here.

If you do not set a local address to an object, the first suitable configured local address will be used. If there is no suitable address, for example if you wish to send a request with UDP/IPv6 address but did not define such a local address, the request sending will fail.

The usage of a local address is different for the different transports.

In UDP, when the local address is set, the SIP Stack uses the socket bound to this address to send the request.

In TCP/TLS, when the local outbound address is set, the SIP Stack matches a listening address (one of the address given for TCP/TLS in the configuration) and sets this address in the Via header of the message. The request will be sent from the same address but from an ephemeral port number.

Note Only requests will be sent through the local address. Responses will always be sent back through the same local address as their request. When TCP/TLS is used, the response will use the same *connection* on which the request was received. Only if this *connection* was closed for some reason will a new *connection* be constructed.

Dynamic Local Addresses (DLA)

The Dynamic Local Addresses (DLA) feature of the SIP Stack enables the application to open and close local IP addresses at runtime. To enable the DLA feature, you have to set the *bDLAEnabled* configuration parameter to RV_TRUE. By default, the DLA feature is disabled since DLA consumes extra resources such as memory and mutexes. The *maxNumOfLocalAddresses* configuration parameter determines the maximum number of local addresses that the SIP Stack will be able to open concurrently. You should set this number upon initialization of the SIP Stack.

DLA API Functions

The following API functions are provided for DLA usage:

- RvSipTransportMgrLocalAddressAdd()
- RvSipTransportMgrLocalAddressRemove()
- RvSipTransportMgrLocalAddressFind()
- RvSipTransportMgrLocalAddressGetDetails()
- RvSipTransportMgrLocalAddressGetFirst()
- RvSipTransportMgrLocalAddressGetNext()

Sample Code

The following code configures two UDP addresses, removes the first UDP address, and opens a new UDP address in the SIP Stack.

```
/*=========================================================================================*/
RvSipStackInitCfg(sizeof(stackCfg),&stackCfg);
/*Configures the Stack to listen to two UDP addresses.*/
strcpy(stackCfg.localUdpAddress,"172.20.4.23");
stackCfg.localUdpPort = 5060;
stackCfg.numOfExtraUdpAddresses = 1;
stackCfg.localUdpAddresses      = calloc(stackCfg.numOfExtraUdpAddresses,sizeof(RvChar*));
stackCfg.localUdpPorts          = calloc(stackCfg.numOfExtraUdpAddresses,sizeof(RvUint16));
stackCfg.localUdpAddresses[0]   = malloc(strlen("172.20.4.24")+1);
strcpy(stackCfg.localUdpAddresses[0],"172.20.4.24");
stackCfg.LocalUdpPorts[0]     = 5061;
stackCfg.DLAEnabled=true
/*Calls the Stack initialization function.*/
RvSipStackConstruct(sizeof(stackCfg),&stackCfg,&hStackMgr);
.....
RvSipTransportLocalAddrHandle hLocalAddr = NULL;
RvSipTransportAddr            addrDetails;
RvStatus                      rv;
/* Removes the first opened UDP address */
```

```
rv = RvSipTransportMgrLocalAddressGetFirst(hTransportMgr,
         RVSIP_TRANSPORT_UDP, &hLocalAddr);
if (rv != RV_OK)
{
         if (rv == RV_ERROR_NOT_FOUND)
        {
             printf("No UDP address is opened currently in the Stack\n");
        }
}
else
{
         RvSipTransportMgrLocalAddressRemove(hTransportMgr, hLocalAddr);
}
/* Opens new UDP address */
addrDetails.eAddrType = RVSIP_TRANSPORT_ADDRESS_TYPE_IP;
addrDetails.eTransportType = RVSIP_TRANSPORT_UDP;
addrDetails.port = 6060;
strcpy(addrDetails.strIP,"172.20.4.24");
rv = RvSipTransportMgrLocalAddressAdd(hTransportMgr,
         &addrDetails, sizeof(addrDetails),RVSIP_FIRST_ELEMENT,NULL/*Base Address*/
         ,hLocalAddr);
if (rv == RV_OK)
{
         printf("New UDP address was successfully opened in the Stack\n");
}
/*=========================================================================================*/
```

HIGH AVAILABILITY

The SIP Stack provides a save and restore mechanism that enables the application to back up calls that have reached the CONNECTED state or *subscriptions* in the ACTIVE state. Backing up calls and *subscriptions* lets application developers implement redundancy capabilities in their systems, allowing back-up systems to "take over" when the primary system goes down. Given that media streams remain active, the backup object (*call-leg* or subscription) will not be affected. By using the high availability mechanism, the application is able to restore a *call-leg* or subscription transparently to the user.

When storing a connected *call-leg* or active subscription, all of the *call-leg* and subscription parameters are copied into a consecutive buffer. The application can then save this buffer to a file or any other desired media. This buffer is used when restoring the backup object.

The SIP Stack provides Call-leg Layer and Subscription Layer API functions to support the high availability mechanism.

CALL-LEG LAYER

The Call-leg Layer API functions which support the high availability mechanism are as follows:

- RvSipCallLegGetConnectedCallStorageSize()—returns the size of buffer needed for storing all the parameters of a connected call.
- RvSipCallLegStoreConnectedCall—copies all *call-leg* parameters from a given *call-leg* to a given buffer. If the *call-leg* holds a list of subscription, only active *subscriptions* will be stored. This buffer should be supplied when restoring the *call-leg*.
- RvSipCallLegRestoreConnectedCall—restores all *call-leg* parameters from a given buffer. The *call-leg* will assume the CONNECTED state.
- RvSipCallLegRestoreOldVersionConnectedCall—to restore a call that was stored by Stack version 3.0 to a Stack of a newer version, you have to compile the new Stack with the RV_SIP_HIGH_AVAL_3_0 compilation flag, and use the RvSipCallLegRestoreOldVersionConnectedCall() function.

SUBSCRIPTION LAYER

The Subscription Layer API functions which support the high availability mechanism are as follows:

- RvSipSubsGetActiveSubsStorageSize—returns the size of buffer needed for storing all the parameters of an active subscription.

- RvSipSubsStoreActiveSubs—copies all subscription parameters from a given subscription to a given buffer. This buffer should be supplied when restoring the subscription.
- RvSipSubsRestoreActiveSubs—restores all subscription parameters from a given buffer. The subscription will assume the ACTIVE state.

Sample Code

The following code replaces a connected *call-leg* with a newly created *call-leg* using the store/restore mechanism. The sample assumes that the received *call-leg* is in the CONNECTED state. The new *call-leg* is returned.

```
/*=========================================================================================*/
RvSipCallLegHandle AppStoreRestoreConnectedCall(
IN RvSipCallLegMgrHandle    hCallLegMgr,
IN RvSipCallLegHandle       hCallLeg)
{
RvInt32            storageSize;
void*              storageBuffer;
RvSipCallLegHandle hNewCallLeg;
RvStatus     rv;
/*Gets the size of the storage buffer.*/
RvSipCallLegGetConnectedCallStorageSize(hCallLeg, &storageSize);
storageBuffer = malloc(storageSize);
/*Stores the call information.*/
rv = RvSipCallLegStoreConnectedCall(hCallLeg, storageBuffer,storageSize);
if(rv != RV_OK)
{
        printf("Failed in RvSipCallLegStoreConnectedCall call=0x%x (rv=%d)", hCallLeg,rv);
        return NULL;
}
/*Terminates the call and creates a new call.*/
RvSipCallLegTerminate(hCallLeg);
RvSipCallLegMgrCreateCallLeg(hCallLegMgr, NULL, &hNewCallLeg);
/*Restores the terminated call-leg parameters into the newly created call-leg.*/
rv = RvSipCallLegRestoreConnectedCall (hNewCallLeg,storageBuffer,storageSize);
if(rv != RV_OK)
{
        printf("Failed in RvSipCallLegRestoreConnectedCall (rv=%d)", rv);
        return NULL;
}
```

```
return hNewCallLeg;
}
/*=========================================================================================*/
```

REPLACES HEADER

The Replaces header is a SIP header defined by *draft-ietf-sip-replaces*. This header is used to logically replace an existing SIP dialog with a new SIP dialog. The Replaces header contains the Call-ID, To tag and From tag, which identify a dialog to be replaced.

The Replaces header may also contain an “early-only” flag. This flag, if present, indicates that the Replaces header refers only to a dialog which is in an early state. This means that a dialog has not yet been established (for example, if it received a 1xx response, but did not yet receive a final response).

A Replaces header can be placed in an INVITE message or it can be part of the Refer-To header in a REFER message. When a User Agent receives an INVITE request with a Replaces header, it will search for a call that has the same Call-ID, From tag and To tag, and if necessary, that is in an early state. If the User Agent finds one, it will disconnect this call and accept the INVITE request.

When an application receives a REFER with Replaces in the Refer-To header, it will send the new INVITE, triggered by this REFER with a Replaces header, in the INVITE message.

SENDING A MESSAGE WITH REPLACES HEADER

The application can use the Replaces header in one of the following situations:

- When sending a REFER message
- When sending an INVITE message

SENDING REFER WITH REPLACES INFORMATION

If the application wants to send a REFER message with Replaces header in the Refer-To header, it has to create a subscription and call RvSipSubsReferInit() or RvSipSubsReferInitStr() with the Replaces header that it wants to add to the Refer-To header.

SENDING INVITE WITH REPLACES HEADER

The application can set the Replaces header in the *call-leg* before sending an initial INVITE request, by calling the function, RvSipCallLegSetReplacesHeader(). If the *call-leg* was created as a result of an accepted REFER, and the Refer-To header in the REFER message contained the Replaces header, the *call-leg* will insert the Replaces header that was in the REFER into the INVITE message automatically. If the application wants a

different Replaces header to be inserted into the INVITE, or that the INVITE will not contain any Replaces header, it can set the Replaces header of the *call-leg* before connecting the call.

Receiving an Invite Message with Replaces header

When an initial INVITE message that contains a Replaces header is received, the *call-leg* calls the evStateChanged callback with the RVSIP_CALL_LEG_STATE_OFFERING state and the RVSIP_CALL_LEG_REASON_REMOTE_INVITING_REPLACES reason.

The application should do the following:

1. Check if it has a *call-leg* that matches the Replaces header in the INVITE message. The application can check it with the RvSipCallLegReplacesGetMatchedCallExt() function.
2. If the application did not find a matched *call-leg*, it should reject the INVITE.
3. If the application found a matched *call-leg*, it should disconnect this *call-leg* and accept the INVITE.

Note If an application wants to work with Replaces, it must add the “replaces” option tag to the Supported header.

Sample Code

The following sample code demonstrates the State Changed callback.

```
/*=========================================================================================*/
void RVCALLCONV AppCallLegStateChangedEv(
                                        IN RvSipCallLegHandle hCallLeg,
                                        IN RvSipAppCallLegHandle hAppCallLeg,
                                        IN RvSipCallLegState eState,
                                        IN RvSipCallLegStateChangeReason eReason)
{
    RvInt16 responseCode = 0;

    switch (eState)
    {
    case RVSIP_CALL_LEG_STATE_OFFERING:
        if(eReason == RVSIP_CALL_LEG_REASON_REMOTE_INVITING_REPLACES)
        {
            RvSipCallLegHandle          hMatchedCallLeg = NULL;
            RvSipCallLegReplacesReason  eReplacesReason =
                            RVSIP_CALL_LEG_REPLACES_REASON_UNDEFINED;

            /*Looks for the call-leg to be replaced.*/
            RvSipCallLegReplacesGetMatchedCallExt(hCallLeg,
                                                  &eReplacesReason,
                                                     &hMatchedCallLeg);
            if(hMatchedCallLeg != NULL &&
               eReplacesReason == RVSIP_CALL_LEG_REPLACES_REASON_DIALOG_FOUND_OK)
            {
                /* found the correct call-leg to replace.
                   1. Accepts the new call-leg
                   2. Disconnect the replaced call-leg. */
                RvSipCallLegAccept(hCallLeg);
                RvSipCallLegDisconnect(hMatchedCallLeg);
                return;
            }

            /* Replaced call-leg was not found. Decide on the correct response */
            switch(eReplacesReason)
            {
            case RVSIP_CALL_LEG_REPLACES_REASON_FOUND_CONFIRMED_DIALOG:
                responseCode = 486;
```

```
                break;
            case RVSIP_CALL_LEG_REPLACES_REASON_FOUND_TERMINATED_DIALOG:
                responseCode = 603;
                break;
            case RVSIP_CALL_LEG_REPLACES_REASON_DIALOG_NOT_FOUND:
            case RVSIP_CALL_LEG_REPLACES_REASON_FOUND_NON_INVITE_DIALOG:
            case RVSIP_CALL_LEG_REPLACES_REASON_FOUND_INCOMING_EARLY_DIALOG:
            default:
                responseCode = 481;
                break;
            }

            /* Reject the new call-leg. */
            RvSipCallLegReject(hCallLeg, 481);
        }
        break;
    default:
        break;
    }
}
/*============================================================================================*/
```

SIP SESSION TIMER

The Session Timer extension enables determining the duration time of a call. A Session-Expires header included in the 2xx response of an initial INVITE determines the session duration. Periodic refresh enables extending the duration of the call and allows both User Agents (UAs) and proxies to determine if the SIP session is still active. The refresh of a SIP session is done through a re-INVITE or UPDATE.

The Session-Expires header in the 2xx response also indicates which side will be the refresher, in other words, which side will be responsible for generating the refresh request. The refresher can be the UAC or UAS. The refresher should generate a refresh request (using re-INVITE or UPDATE) before the session expires. If no refresh request is sent or received before the session expires, or if the refresh request is rejected, both parties should send BYE.

The Session Timer feature defines two new headers—Session-Expires and Min-SE—and a new response code of 422. The Session-Expires header conveys the lifetime of the session, the Min-SE header conveys the minimum value for the Session Timer, and the 422 response code indicates that the Session Timer duration is too small. A 422 response leads to the termination of the response receiving *call-leg*. Consequently, to recover a 422 response, a new outgoing *call-leg* has to be created according to section 7.3 of RFC 4028. This new *call-leg* **must** be created only after the original 422-rejected *call-leg* is terminated. If the Min-SE header is missing, the minimum value for the Session Timer is 90 seconds by default, according to the Session Timer RFC.

CONFIGURATION PARAMETERS

The configuration parameters of the Session Timer feature are as follows:

supportedExtensionList

List of supported option-tags, separated by commas, which are supported by the SIP Stack. The list will be added to a Supported header for outgoing messages. In order to support the session timer feature, the application needs to add the “timer” option tag to the list.

sessionExpires

The time at which an element will consider the call timed out, if no successful INVITE *transaction* or UPDATE *transaction* occurs beforehand. This value is inserted into every INVITE and UPDATE *transaction* in the Session-Expires header unless it was configured to zero. A zero sessionExpires means that the sessionTimer feature is turned off. If the “timer” option tag is not part of the supported list, the sessionExpires value will be ignored.

Default: 1800 seconds

minSE

The minimum value for the session interval that the application is willing to accept. If the application does not set this parameter, the minSE value is set to the default value of 90 seconds according to the Session Timer RFC. Also, the Min-SE header will not be present in the sent requests (except for a request, following a 422 response). However, if the application set this parameter to 90 or any other value, the Min-SE header will appear in any sent request.

Default: –1

The *sessionExpires* and *minSE* parameters can also be determined for a specific *call-leg* or *call-leg transaction* using the RvSipCallLegSessionTimerSetPreferenceParams() and RvSipCallLegTranscSessionTimerSetPreferenceParams() functions defined below.

TIMERS

When the SIP Stack is configured to support the timer extension, the Initial INVITE request and every refresh will automatically include the Session-Expires header with the configured sessionExpires value, unless the sessionExpires is configured to zero. If you configured a minSE value, a Min-SE header will also be included. After a 2xx response is received with the final Session-Expires value and the call is connected, the SIP Stack sets a timer for the session. A timer is set for each party of the call. On the refresher side, an alert timer is set to a default of the final sessionExpires/2. This timer alerts the *call-leg* to send a refresh request. You can change the alert time default value using the RvSipCallLegSessionTimerSetAlertTime() function. The SIP Stack sets another timer to the end of the session in both sides (the refresher side and the non-refresher side). This timer is set to:
sessionExpires–min(32,sessionExpires/3)
according to the SessionTimer RFC. When this timer expires, a BYE is sent.

MODE OF OPERATION

The SIP Session Timer feature can be operated in manual mode only. When the Session Timer Alert timer is expired, the application on the refresher side will be notified about it using the CallLegSessionTimerRefreshAlertEv() callback.

As mentioned in the above Timers section the SIP Stack sets another timer to the end of the session on both sides (the refresher side and its remote side). When this timer is expired, the application will be notified about it using the CallLegSessionTimerNotificationEv() callback.

CALL-LEG SESSION TIMER PARAMETERS

The following set of *call-leg* parameters relates to the session-timer feature.

Preference and Negotiation Parameters

The preference and negotiation parameters are temporary parameters used before the final session timers parameters were determined. The preference parameters are the parameters that the local party prefers for this session. You can only set the preference parameters. Each setting of preference parameters is valid for a single session timer refresh within a dialog (either an initial or subsequent refresh). The negotiation parameters are the parameters that are negotiated with the remote party. You can only get the negotiation parameters.

Before a UA sends a refresh request or response to a refresh request, the UA can set the preference parameters using the RvSipCallLegSessionTimerSetPreferenceParams() function. Upon receiving a refresh request or a 2xx response to a refresh request, the UA can learn about the remote party requested parameters using the RvSipCallLegSessionTimerGetNegotiationParams() function.

Session Expires, Min-SE, Refresher Type

The final session timer parameters that were determined in the last negotiation process.

Alert Time

The time in which the refresh request will be sent. For more information, see Timers.

API Functions

The SIP Stack provides the API functions listed below to support the Session Timer mechanism.

Get and Set Functions

The following Get and Set functions are provided for session timer support:

RvSipCallLegSessionTimerSetPreferenceParams()

Sets the preferred Session Timer parameters associated with this call. These parameters may not be equal to the Session Timer parameters of the call in the end of the negotiation. The preference parameters setting is valid for a single session refresh request or response. If an UNDEFINED value is set as the minSE, it is configured as the default (90 seconds) according to the Session Timer RFC and the Min-SE header will not appear in the upcoming sent messages.

If the sessionExpires parameter is set to 0, the Session-Timer mechanism is turned off immediately in the current *call-leg*. Moreover, the Session-Timer mechanism can be turned on by:

- Calling this function with non-zero sessionExpires value in the middle of a call.
- Calling one of the Session-Timer Call-leg API functions for refreshing the current session (RvSipCallLegTranscSessionTimerGeneralRefresh() or RvSipCallLegSessionTimerInviteRefresh()).

RvSipCallLegSessionTimerGetNegotiationParams()

Gets the negotiation Session Timer parameters associated with this call. These parameters may not be equal to the Session Timer parameters of the call in the end of the negotiation.

RvSipCallLegSessionTimerGetSessionExpiresValue()

Returns the Session-Expires value associated with the call.

RvSipCallLegSessionTimerGetMinSEValue()

Returns the Min-SE value associated with the call.

RvSipCallLegSessionTimerGetRefresherType()

Returns whether the refresher of the call is UAC or UAS. The value of refresher type can be different from refresher preference that was requested by the application.

RvSipCallLegSessionTimerSetAlertTime()

Allows the application to modify the time in which the CallLegSessionTimerRefreshAlertEv() callback occurs. (The default time is sessionExpires/2).

RvSipCallLegSessionTimerGetAlertTime()

Returns the time in which the CallLegSessionTimerRefreshAlertEv() callback will occur after the end of the Session-Timer negotiation.

RvSipCallLegTranscSessionTimerSetPreferenceParams()

Sets the preferred Session Timer parameters associated with this transaction. The only general *transaction* allowed is "UPDATE". These parameters may not be equal to the Session Timer parameters of the call at the end of the negotiation. If an UNDEFINED value is set as the minSE, it is configured as the default (90 seconds) according to the Session Timer RFC and the Min-SE header will not appear in the upcoming sent message that is related to this transaction.

If the sessionExpires parameter is set to 0, the Session-Timer mechanism is turned off immediately in the current *call-leg*. Moreover, the Session-Timer mechanism can be turned on by:

- Calling this function with non-zero sessionExpires value in the middle of a call.
- Calling one of the Session-Timer Call-leg API functions for refreshing the current session (RvSipCallLegTranscSessionTimerGeneralRefresh() or RvSipCallLegSessionTimerInviteRefresh()).

RvSipCallLegTranscSessionTimerGetNegotiationParams()

Gets the negotiation Session Timer parameters associated with this transaction. These parameters may not be equal to the Session Timer parameters of the call in the end of the negotiation.

CONTROL FUNCTIONS

The following Control functions are provided for session time support:

RvSipCallLegSessionTimerRefresh()

Causes a re-INVITE to be sent in order to refresh the session duration. This function can be called only in the CONNECTED state when there is no other pending re-INVITE transaction. The response of the remote party to the re-INVITE will be given in the RvSipCallLegModifyStateChangedEv() callback.

RvSipCallLegTranscSessionTimerGeneralRefresh()

Creates a *transaction* related to the *call-leg* and sends a Request message with the given method in order to refresh the call. The only general *transaction* allowed is "UPDATE". The request will have the To, From and Call-ID of the *call-leg* and will be sent with a correct CSeq step. It will be record routed if needed. The request will contain all the parameters related to the Session Timer.

CALLBACKS

The Call-leg Session Timer API defines the following callback functions:

RvSipCallLegSessionTimerRefreshAlertEv()

Notifies that the alert time (the time in which the application needs to send a re-INVITE or UPDATE to refresh the call) is expired. The application needs to send a refresh using the RvSipCallLegSessionTimeRefresh() or RvSipCallLegTranscSessionTimerGeneralRefresh() functions. (This callback is called only on the refresher side).

RvSipCallLegSessionTimerNotificationEv()

Notifies the application about events that are related to the Session Timer feature. Note the following:

- When the callback is called with the RVSIP_CALL_LEG_SESSION_TIMER_SESSION_EXPIRES reason, it notifies the application that the session time is about to expire. It is the responsibility of the application to decide whether to send BYE or to do something else.
- When the callback is called with the RVSIP_CALL_LEG_SESSION_TIMER_NOTIFY_REASON_422_RECEIVED reason, it notifies the application that a 422 response was received over the current *call-leg*. According to RFC 4028 section 7.3, in order to retry the Session Timer mechanism a new request has to be sent with same Call-ID, To, and From of the previous request, but the CSeq should contain a new sequence number that is one higher than the previously rejected request. Thus, in order to retry the mechanism, a new *call-leg* should be created using RvSipCallLegMgrCreateCallLeg(). Moreover, this new *call-leg* details has to be set by the various API functions of the SIP Stack, such as RvSipCallLegSetFromHeader(), RvSipCallLegSetCSeq(), RvSipCallLegSetCallId() and so on, according to the rejected *call-leg*.
  Note: The newly created *call-leg* cannot be connected until the original, rejected *call-leg* is completely terminated.

RvSipCallLegSessionTimeNegotiationFaultEv()

Notifies the application about negotiation problems while defining the Session Timer parameters for a call.

When the callback is called with the RVSIP_CALLLEG_SESSION_TIMER_DEST_NOT_SUPPORTED reason, it notifies the local party that a 2xx final response was received, but the server does not support the Session Timer. The application should return synchronously whether or not it wants to execute Session Timer.

The Session Timer mechanism can be operated as long as one of the two UAs in the call leg supports the extension. If the application decides to operate the Session Timer, that side will send the refresh. The other side will see the refreshes as repetitive re-INVITEs. The default behavior is to execute the Session Timer mechanism for the call.

When the callback is called with the RVSIP_CALLLEG_SESSION_TIMER_APP_REFRESH_REQUEST_REJECT reason, it notifies the application that the refresher preference did not match the call refresher. The application should return synchronously whether or not it wants to execute the Session Timer. If the application decides to operate the Session Timer mechanism, the refresher will be different from the application request. The default behavior is to execute the Session Timer mechanism for the call.

SAMPLE CODE

The sample code below demonstrates an implantation of the CallLegSessionTimerRefreshAlertEv() callback. This callback notifies that the alert time (the time in which the application needs to send re-INVITE or UPDATE to refresh the call) is expired.

```
/*-------------------------------------------------------------------------------------*/
  RvStatus RVCALLCONV CallLegSessionTimerRefreshAlertEv(
              IN  RvSipCallLegHandle    hCallLeg,
              IN  RvSipAppCallLegHandle hAppCallLeg)
  {
     RvStatus           status = RV_OK;
     RvInt32            SE = 500;
     RvInt32            minSE = 100;

      /*The local party wishes to be the refresher.*/
      RvSipCallLegSessionTimerRefresherPreference eRefreshPref
          = RVSIP_CALL_LEG_SESSION_TIMER_REFRESHER_LOCAL;

      /*Sets the session timer parameters.*/
      status = RvSipCallLegSessionTimerSetPreferenceParams(
                                 hCallLeg,SE,minSE,eRefreshPref);
      if(status != RV_OK)
      {
          return status;
      }

      /*Sends a Refresh using re-INVITE.*/
      status= RvSipCallLegSessionTimerRefresh(hCallLeg);
      if(status != RV_OK )
      {
          return status;
      }

      return status;
  }
/*-------------------------------------------------------------------------------------*/
```

General URI Scheme Support

SIP defines and uses different URI schemes, such as “sip”, “sips”, “im”, “pres” and “tel”. In addition, some implementations define proprietary URI schemes. A general framework in the SIP Stack provides support for sending any type of URI scheme. In each of the SIP Stack layers, a new callback function was added with the following form: RvSipXXXOtherURLAddressFoundEv().

Whenever a SIP Stack object tries to send a request and the destination address is not a sip or a sips URI, this callback is called. The application is required to replace the non-SIP URI address with a sip URI address which the SIP Stack knows how to handle. Once the application converts the non-sip URI to a sip URI address, the SIP Stack continues with its regular behavior and tries to send the message according to the new address. The message however remains unchanged and will include the non-sip URI address.

The following callbacks are supplied:

RvSipTransactionOtherURLAddressFoundEv()

For *transaction* requests.

RvSipTranscMgrOtherURLAddressFoundEv()

For requests sent by the *TransactionMgr* in proxy implementations.

RvSipSubsOtherURLAddressFoundEv()

For subscription requests.

RvSipCallLegOtherURLAddressFoundEv()

For *call-leg* requests.

RvSipTransmitterOtherURLAddressFoundEv()

For *transmitter* requests.

Other URI Extension Support

The Stack provides extension support for “im” and “pres” URI schemes. You can compile the Stack with the RV_SIP_OTHER_URI_SUPPORT compilation flag and, as a result, addresses with im and pres schemes will be parsed as im/pres-addresses (a regular address object with type = RVSIP_ADDRTYPE_IM or RVSIP_ADDRTYPE_PRES), and not as an absolute URI (RVSIP_ADDRTYPE_ABS).

When working with Other URI support, whenever a SIP Stack object tries to send a request and the destination address scheme is im or pres, the RvSipXXXOtherURLAddressFoundEv() callback will **not** be called, and the SIP Stack will manage to handle the address.

TEL URI Extension Support

The SIP Stack also provides extension support for the "tel" URI scheme. You can compile the SIP Stack with the RV_SIP_TEL_URI_SUPPORT compilation flag, and, as a result, addresses with tel schemes will be parsed as a tel-address (a regular address object with type = RVSIP_ADDRTYPE_TEL), and not as an absolute URI (RVSIP_ADDRTYPE_ABS).

When working with tel URI-support, whenever a SIP Stack object tries to send a request and the destination address scheme is tel, the SIP Stack acts according to its *bresolveTelUrls* configuration parameter. If it was set to TRUE, the SIP Stack will handle the tel address, Otherwise, RvSipXXXOtherURLAddressFoundEv() will be called.

Type of Service (TOS)

The SIP Stack provides an API that enables setting and getting the IP_TOS option for sockets that serve local addresses and *connections*, which the SIP Stack opened. The API can be called at any moment of the SIP Stack life cycle. Incoming *connections* inherit the IP_TOS option from the local addresses, on which the *connections* are accepted.

The value of the TOS byte that the API set depends on the RFC implemented by the OS, since different RFCs declare different rules for TOS values. Some OS compilations do not support TOS. In this case, the OS typically supplies the DSCP mechanism (see RFC 2474). Note that the SIP Stack does not provide an API for DSCP.

The SIP Stack provides the following API functions for support of the TOS option:

RvSipTransportMgrLocalAddressSetIpTosSockOption()

Sets the option value into the opened local address. All the UDP packets that the SIP Stack sends from this address will be marked with the value in the TOS byte of the IP header. Also, all the packets (TCP) that the SIP Stack sends on the *connection* that were accepted on this address will be marked with the set value in the TOS byte of the IP header.

RvSipTransportMgrLocalAddressGetIpTosSockOption()

Gets the option value that was set for the local address.

RvSipTransportConnectionSetIpTosSockOption()

Sets the option value into the local address, from which the outgoing *connection* is opened. All the packets (TCP) that are sent on this *connection* will be marked with the set value in the TOS byte of the IP header. This function can be called from RvSipXXXNewConnInUseEv() callback.

RvSipTransportConnectionGetIpTosSockOption()

Gets the option value that was set for the outgoing *connection*.

SAMPLE CODE

The following code goes over all local addresses that the SIP Stack opened and sets the IP_TOS option for their sockets. The function illustrated in the sample can be called after SIP Stack initialization.

```
/*=========================================================================*/
RvStatus SetTOSForAllLocalAddresses(
                            IN RvSipTransportMgrHandle hTransportMgr,
                            IN RvInt32                 tos,
                            IN RvSipTransport          eTransportType)
{
    RvStatus rv;
    RvSipTransportLocalAddrHandle hLocalAddr = NULL;

    rv = RvSipTransportMgrLocalAddressGetFirst(
                            hTransportMgr, eTransportType, &hLocalAddr);
    if (RV_OK!=rv  &&  RV_ERROR_NOT_FOUND!=rv)
    {
        printf("RvSipTransportMgrLocalAddressGetFirst() failed (rv=%d)\n",rv);
        return rv;
    }
    while (NULL != hLocalAddr)
    {
        rv = RvSipTransportMgrLocalAddressSetIpTosSockOption(hLocalAddr,tos);
        if (RV_OK != rv)
        {
            printf("SetTOSForAllLocalAddresses() failed (hLocalAddr=0x%x, rv=%d)
            \n",hLocalAddr,rv);
        }
        else
        {
            printf("TOS=%d was set successfully for hLocalAddr=0x%x\n",tos,hLocalAddr);
        }
        rv = RvSipTransportMgrLocalAddressGetNext(hLocalAddr, &hLocalAddr);
        if (RV_OK!=rv  &&  RV_ERROR_NOT_FOUND!=rv)
        {
            printf("RvSipTransportMgrLocalAddressGetFirst() failed (rv=%d)\n",rv);
            return rv;
        }
    }
    return RV_OK;
}
/*=========================================================================*/
```

Changing the Top Via Header of the Message

The SIP Stack automatically handles the top Via header of its outgoing requests. After the destination address is resolved and the local address that will be used is known, the *transmitter* responsible for sending the request places this local address in the "sent-by" parameter of the top-most Via header. In some cases, the application does not want to reveal the address from where a request was sent. Since this information is part of the "sent-by" parameter of the Via header, the application must change this parameter. This task can be accomplished only in the RvSipXXXFinalDestResolvedEv() callback implementation that can be found on each of the SIP Stack layers. The application can override the "sent-by" information that the *transmitter* placed and this new information will be sent to the remote party. Note that the application must not change the Via header branch or transport parameters.

Timer Configuration

The SIP Stack provides API functions to control the timer values of the *transaction* and the number of retransmissions during runtime. You can use these API functions to set timer values to a *transaction* different than the one that was configured. To set new timer values, you should fill the RvSipTimers structure and set it to the object that holds the *transaction* you wish to control. You may set values in the RvSipTimers structure to –1, so the configured value will be taken instead.

The following API functions are supplied:

RvSipTransactionSetTimers()

Sets the timers in a *transaction*. You can use this function to set timers to an outgoing *transaction* before sending the request, and to set timers to an incoming *transaction* in the RvSipTransactionCreatedEv() callback.

RvSipCallLegSetTranscTimers()

Sets the timer values in a *call-leg*. Setting timers in a *call-leg* effects all the *transactions* that this *call-leg* handles. If you want the timers to influence all *call-leg transactions*, you should set them on *call-leg* creation (after calling RvSipCallLegMgrCreateCallLeg() for an outgoing *call-leg*, and in the RvSipCallLegCreateEv() callback for an incoming *call-leg*).

RvSipRegClientSetTranscTimers()

Sets the timers in a reg-client object. Setting timers in a reg-client effects all the *transactions* that this object handles.

Note For all objects, you can also reset the timer back to the configuration values with the same API function, by supplying NULL to the set function instead of the RvSipTimers pointer.

ENUM Resolution Support

According to RFC 3824, ENUM (E.164 Number Mapping) is a system that uses DNS (Domain Name Service) to translate certain telephone numbers, such as +12025332600, into URIs (Uniform Resource Identifiers). The SIP protocol handles telephone numbers that cannot be routed in the SIP context by using ENUM.

URIs, which include PSTN phone numbers in E.164 format, must include the "tel:" schema. This way, the SIP Stack will refer to them as TEL URIs. Once the SIP Stack encounters a TEL URI, it will try to resolve it to a SIP URI using an ENUM NAPTR query. The result of the NAPTR query is a regular expression, and applying this regular expression on the original TEL URI phrase may result in a SIP URI. The SIP Stack, however, does not evaluate regular expressions and asks the application to resolve the regular expression using a callback. If the resolution of a TEL URI fails at any stage, the SIP Stack will try to resolve it as a regular Other URI. For more information, see the Other URI Extension Support chapter.

Initiating an ENUM Query

The two basic ways of initiating an ENUM NAPTR query are as follows:

- Using the RvSipResolverResolve() function with the RVSIP_RESOLVER_MODE_FIND_URI_BY_NAPTR. resolution mode. For more information, see the Working with Resolvers chapter.
- Sending a message to a TEL URI destination over one of the SIP Stack objects (such as *call-leg*, *transmitter* and so on). The SIP Stack supplies a new API function in the Message layer for creating TEL URIs. For more information, see the *SIP Stack Message Layer Reference Guide*.

Retrieving an ENUM Result

According to RFC 3764, generally there is no need to have more than one ENUM NAPTR record under a single telephone number (section 4). Thus, the SIP Stack keeps only a single entry of an ENUM query result element within a

DNS list object. The single query result can be retrieved by using one of the Transport API functions for manipulating DNS list objects. For more information, see the Working with DNS chapter.

ENUM Event

The Transmitter API supplies an ENUM event in the form of a callback function, to which your application may listen and react. To listen to this event, your application should pass the event handler pointer to the *TransmitterMgr* object using RvSipTransmitterMgrSetEvHandlers(). When an event occurs, the *transmitter* calls the event handler function using the pointer. The event callback is as follows:

RvSipTransmitterMgrRegExpResolutionNeededEv()

This function will pop at any time the SIP Stack requires the resolution of a regular expression. The RvSipTransmitterRegExpResolutionParams structure will contain all the information needed for resolving the regular expression, including the string to evaluate the regular expression and an array in which to store the matches. By implementing this callback, the application is able to provide the *transmitter* with the required resolution analysis.

Configuration

The SIP Stack provides an easy way to use the ENUM methodology by using the following configuration parameters found in the configuration structure, RvSipStackCfg:

strDialPlanSufix

The algorithm to create an NAPTR query string is described in RFC 3761. In section 2.4.4 the algorithm uses “e164.arpa” as a concatenated phrase to the NAPTR query. In the strDialPlanSufix parameter you can specify the dial plan suffix you need to use (in most cases, it will be “e164.arpa”).

bResolveTelUrls

If set to RV_FALSE, the SIP Stack will treat a TEL URI as any non-SIP URI. If set to RV_TRUE, ENUM will be used for TEL URI resolution.

Compilation

In order to enable TEL URI parsing, you must compile the SIP Stack with RV_SIP_TEL_URI_SUPPORT. You can remove RV_SIP_TEL_URI_SUPPORT and the SIP Stack will remove both the TEL URI parsing support and the special treatment for those URIs.

Sample Code

The following code sample illustrates an implementation of the ENUM callback using an abstract function that which performs regular expression parsing, called AppRegComp.

```
/*=========================================================================*/
RvStatus AppRegExpResolutionNeeded EvHandler (
              IN    RvSipTransmitterMgrHandle                hTrxMgr,
              IN    void*                                    pAppTrxMgr,
              IN    RvSipTransmitterHandle                   hTrx,
              IN    RvSipAppTransmitterHandle                hAppTrx,
              INOUT RvSipTransmitterRegExpResolutionParams*  pRegExpParams)
{
    RvStatus rv = RV_OK;
    RvInt32 index;
    RvSipTransmitterRegExpMatch *pCurrMatch;

    rv = AppRegComp(
        pRegexpParams->strRegExp,              /* A regular expression from NAPTR query */
        pRegexpParams->strString,    /* The string to apply the regexp on */
        pRegexpParams->matchSize,    /* The size of the pMatches array */
        pRegexpParams->eFlags,       /* regexp parsing flags */
        pRegexpParams->pMatches);    /* Array of matching substrings offsets, any unused*/
                                     /* array elements contain -1 offset */
        for (index =0, pCurrMatch = pRegexpParams->pMatches;
            index < pRegexpParams->matchSize && pCurrMatch-> startOffSet != -1;
            index++, pCurrMatch++)
       {
                printf("Found match from %d to %d",
                    pCurrMatch-> startOffSet, pCurrMatch-> endOffSet);
       }

        return  rv;
}
/*=========================================================================*/
```

20

CONFIGURATION

INTRODUCTION

This chapter describes the configuration of the SIP Stack. You will find information about the SIP Stack configuration parameters and how to set the parameters to suit your specific application requirements.

INITIALIZATION

SIP Stack configuration and memory allocation is performed upon initialization. You initialize the SIP Stack using the function, RvSipStackConstruct(). The RvSipStackConstruct() function is defined as follows:

```
/*=========================================================================*/
RvStatus RvSipStackConstruct(
        IN     RvInt32            cfgStructSize,
        INOUT  RvSipStackCfg      *pStackCfg,
        OUT    RvSipStackHandle   *hStack);
/*=========================================================================*/
```

RVSIPSTACKCFG CONFIGURATION STRUCTURE

The RvSipStackCfg structure that is used in the RvSipStackConstruct function contains the entire SIP Stack configuration. The configuration parameters of the RvSipStackCfg can be divided into two groups:

- Group A—parameters that the SIP Stack cannot figure by itself and the application must supply. For example, maxCallLegs.
- Group B—parameters that the SIP Stack can calculate according the values of other configuration parameters. For example, maxTransactions. When the application does not set the maxTransactions parameter, the SIP Stack will use the following formula: maxTransactions = maxCallLegs + maxSubscriptions.

The following rules apply to parameters of the RvSipStackCfg structure:

1. If you set a parameter from group A to –1, the SIP Stack will use a default hard coded value. For example, maxCallLegs will be set to 10.
2. If you set a parameter from group B to –1, the SIP Stack will calculate the value of this parameter using the values of other parameters.
3. If you set a parameter from group A to an invalid value, rule 1 will apply.
4. If you set a parameter from group B to an invalid value, rule 2 will apply.

The RvSipStackInitCfg() function can help you to initialize the configuration structure and you should call it before you construct the stack.

The function performs the following:

1. Sets the default hard coded value to parameters of group A.
2. Sets –1 to parameters of group B.

You can then change the values of the group A parameters and the rest of the parameters will be changed accordingly. The RvSipStackCfg is an INOUT parameter of the RvSipStackConstruct function. The final calculated configuration values that the SIP Stack will use are set back in the structure by the SIP Stack.

Configuration Parameters

The SIP Stack configuration parameters and default values are as follows:

Stack Object Allocation

maxCallLegs

The maximum number of *call-legs* the SIP Stack allocates. You should set this value to the maximum number of calls you expect the SIP Stack to handle simultaneously.

Default Value: 10

Remarks: Group A parameter

maxTransactions

The maximum number of *transactions* the SIP Stack allocates. You should set this value to the maximum number of *transactions* you expect the SIP Stack to handle simultaneously.

Default Value: –1

Remarks:

- Group B parameter: maxTransaction = maxCallLegs + maxSubscriptions
- If maxCallLeg = 0, then maxTransactions = 10 + maxSubscriptions

maxRegClients

The maximum number of *register-clients* the SIP Stack allocates. You should set this value to the maximum number of Register-Clients you expect the SIP Stack to handle simultaneously.

Default Value: 2

Remarks: Group A parameter

maxTransmitters

The number of Transmitters that the SIP Stack allocates. You should set this value to the maximum number of transmitters you expect the SIP Stack to handle simultaneously. A *transmitter* is used for sending a message that is not related to *transactions* such as ACK on 2xx for forked INVITE, or responses for messages with syntax errors that fail to create a transaction. The application may also use *transmitter* for sending stand-alone messages.

Note Each SIP Stack *transaction* also uses a *transmitter* but the transmitters for the *transactions* are allocated by the SIP Stack automatically and you do not have to add them to the maxTransmitters parameters.

Default value: –1

Remarks:

- Group B parameter: maxTransmitters = 10 + maxCallLegs + maxTransactions
- If forking is enabled, the value will be increased by maxCallLegs/2.

Memory Pool Allocation

A memory pool consists of a set of memory pages used by the SIP Stack for storage. The SIP Stack uses three different pools:

- Message Pool
- General Pool

- Element Pool

MESSAGE POOL

Used to hold and process all incoming and outgoing messages in the form of encoded messages or *message objects*. It is recommended that you configure the page size to the average message size your system is expected to manage.

messagePoolNumofPages

The message pool page number.

Default Value: –1

Remarks: Group B parameter: messagePoolNumOfPages = max (2.5*maxTransactions, 4)

messagePoolPageSize

The message pool page size.

Default Value: 1536

Remarks: Group A parameter

GENERAL POOL

Used by SIP Stack objects, such as *call-legs* and *transactions*, to store the internal fields. For example, the call-leg object will store the To, From and Call-ID headers and the local and remote contact addresses on the general pool pages. The general pool is also used for other activities that demand memory allocation.

generalPoolNumofPages

The number of pages in the general pool.

Default Value: –1

Remarks:

- Group B parameter: generalPoolNumOfPages = 1.5*maxCallLegs + 1.5*maxTransactions + maxRegClients + 5 + maxTransmitters (that are allocated by the application)
- This parameter has a minimum value of 3.

generalPoolPageSize

The size of a page in the general pool.

Default Value: 1024

Remarks:

- Group A parameter
- This parameter has a minimum value of 512.

Element Pool

Used by the SIP Stack to hold small pieces of information. For example, the *call-leg* holds its remote contact on an element pool page; a TLS session holds the hostname on such a page; and the *transmitter* uses this type of page in the DNS procedure.

elementPoolNumofPages

The number of pages in the element pool.

Default value: –1

Remarks:

- Group B parameter: maxCallLegs*2 + 0.5*maxSubscriptions + maxConnections
- The parameter value must be greater than zero

elementPoolPageSize

The size of a page in the element pool.

Default value: 250—for holding an element such as a stand-alone header.

Remarks: Group A parameter

Network Parameters

The network parameters are grouped as follows:

- General configuration
- UDP configuration
- Outbound Proxy
- Connection-oriented configuration
- TCP configuration
- TLS configuration

General Configuration

sendReceiveBufferSize

The buffer size used by the SIP Stack for receiving and sending SIP messages.

Default Value: 2048

Remarks: Group A parameter

Note The SIP Stack can accumulate several SIP messages before starting to process them. These messages are placed in a queue and each message uses a different buffer.

---

maxNumOfLocalAddresses

The maximum number of local addresses that the SIP Stack may use simultaneously. Since the application can add and remove local addresses at runtime, the SIP Stack must know the maximum number of concurrently used addresses on initialization.

Default value: –1

Remarks: Group B parameter: maxNumOfLocalAddresses = 1 + numOfExtraUdpAddresses + 1 + numOfExtraTcpAddresses +

bDLAEnabled

Indicates whether or not the DLA feature is enabled. The DLA (Dynamic Local Addresses) feature enables the application to add and remove local addresses at runtime. If the feature is enabled the local address list is protected by a lock.

Default value: RV_FALSE

Remarks: Group A parameter

numOfReadBuffers

The number of buffers that will be allocated for receiving SIP messages. When the SIP Stack works with several processing threads, several incoming messages can be processed concurrently. To enable this, the SIP Stack should allocate several read buffers.

Default value: –1

Remarks:

- Group B parameter: numOfReadBuffers = processingQueueSize/2

processingQueueSize

The maximum length of the processing queue. The processing queue holds events such as network events. If the SIP Stack is running on a single thread, this thread will pop and process events from the queue. If the SIP Stack is running in multithreaded mode, the processing threads will process the events.

Default value: –1

Remarks: Group B parameter. processingQueueSize = maxCallLegs + maxTransactions + maxRegClients + maxSubscriptions + maxConnections + 10.

bIgnoreLocalAddresses

By default, the SIP Stack is initialized with one UDP and one TCP local address, and the application can configure extra local addresses of any transport type. When this parameter is set to RV_TRUE, all local address information from the configuration is ignored and the SIP Stack is constructed with no local addresses and listening sockets. The application can then use the DLA functionality to add local addresses with listening sockets.

Default value: RV_FALSE

Remarks: Group A parameter

UDP Configuration

localUdpAddress

The local UDP IP of the SIP Stack.

Default Value: 0.0.0.0 (Indicates the local IP address.)

Remarks: Group A parameter

localUdpPort

The local UDP port on which the SIP Stack listens.

Default Value: 5060

Remarks: Group A parameter

numOfExtraUdpAddresses

The size of the localUdpAddresses and localUdpPorts arrays. Applications that wish to listen to more than one UDP address must allocate the localUdpAddresses and the localUdpPort arrays and set the extra addresses in them. The numOfExtraUdpAddresses parameter indicates the size of each array.

Default value: 0 (no extra addresses)

Remarks: Group A parameter

localUdpAddresses
localUdpPorts

Additional local UDP addresses to which the SIP Stack listens. The extra addresses will be used in multihomed host applications.

Note: The addresses and ports array must be allocated according to the size given in numOfExtraUdpAddresses.

Each of the entries of localUdpAddresses must be allocated as well, in order to contain the requested IP address.

Default value: NULL (no extra addresses)

Remarks: Group A parameter

Outbound Proxy Configuration

outboundProxyIpAddress

The IP address of an outbound Proxy the SIP Stack uses.

Default Value: 0—no outbound Proxy

Remarks: Group A parameter

outboundProxyHostName

The host name of an outbound proxy that the SIP Stack uses. For each outgoing request, the DNS will be queried for this host IP address.

Default value: NULL

Remarks:

- Group A parameter
- If you set the outboundProxyIPAddress parameter, the outboundProxyHostName parameter will be ignored.

outboundProxyTransport

Indicates the transport of the outbound proxy that the SIP Stack uses.

Default Value: RVSIP_TRANSPORT_UNDEFINED

Remarks: Group A parameter

outboundProxyPort

The port of the outbound Proxy the SIP Stack uses.

Default Value: 5060

Remarks:

- Group A parameter
- If you set the outbound Proxy port to –1, it will be discovered using the procedures defined in RFC 3263.

Connection Oriented Configuration

tcpEnabled

Indicates whether the TCP is enabled. If set to RV_FALSE, no *connection* will be allocated, and the SIP Stack will not support TCP.

Default Value: RV_FALSE

Remarks: Group A parameter

maxConnections

The number of *connection* sockets to be allocated by the SIP Stack.

Default Value: –1

Remarks:

- Group B parameter: maxconnections = (maxTransactions/2) +1
- If TCP is disabled, no *connection* is allocated.

ePersistencyLevel

The persistency level to be used by the SIP Stack objects.

Default value: –1 (RVSIP_TRANSPORT_PERSISTENCY_LEVEL_UNDEFINED—not using persistent *connections*).

Remarks: Group A parameter

serverConnectionTimeout

Specifies the time duration a server *connection* is kept open. By default, the SIP Stack does not close server *connections* when the *connection* has no more owners. The SIP Stack waits for the remote party to close server *connections*. However you can use the serverConnectionTimeout to change the default behavior of the SIP Stack. If you set serverConnectionTimeout to a value bigger than 0, the SIP Stack will set a timer for each server *connection* once it has no more owners, and close the *connection* when this timer expires. If you set the serverConnectionTimeout to 0, each server *connection* will be closed immediately after the last owner detaches from it.

Default value: –1—the SIP Stack will not close server *connections*.

Remarks: Group A parameter

connectionCapacityPercent

Determines the recommended percentage of opened *connections* that the SIP Stack is allowed to hold at any given moment from its pool of *connections*. RFC 3261 recommends that *connections* are kept open for some period of time after the last message was exchanged over the *connection*. However, the exact time period to leave the *connection* open is implementation-defined. Using the *connectionCapacityPercent* configuration parameter the SIP Stack allows the application to leave client *connections* open even after the *connections* are no longer in use. When the connectionCapacityPercent parameter is greater than 0, the SIP Stack will not close *connections* that are no longer in use. Such *connections* will be kept in a separate list and will remain open as long as their resources are not required. Once the percentage of opened *connections* exceeds the allowed connectionCapacityPercent, the SIP Stack will start closing the *connections* from this list each time it is required to open a new *connection*.

Default value: 0.

Remarks: Group A parameter

TCP Configuration

numOfExtraTcpAddresses

The size of the localTcpAddresses and localTcpPorts arrays. Applications that wish to listen to more than one TCP address must allocate the localTcpAddresses and the localTcpPorts arrays and set the extra addresses in them. The numOfExtraTcpAddresses parameter indicates the size of each array.

Default value: 0 (no extra addresses)

Remarks: Group A parameter

localTcpAddresses
localTcpPorts

Additional local TCP to which the SIP Stack listens. The extra addresses will be used in multihomed host applications.

Default value: NULL (no extra addresses)

Remarks:

- Group A parameter
- The addresses and ports array must be allocated according to the size given in numOfExtraTcpAddresses.

- Each of the entries of localTcpAddresses must be allocated as well, in order to contain the requested IP address.

localTcpAddress

The local TCP address (IP) to which the SIP Stack listens.

Default Value: 0.0.0.0 (Indicates the local IP address.)

Remarks: Group A parameter

localTcpPort

The local TCP port on which the SIP Stack listens.

Default Value: 5060

Remarks: Group A parameter

TLS Configuration

numOfTlsAddresses

The number of TLS addresses on which the application wishes to listen. Setting this number to 0 means that the application does not want to listen to any TLS addresses. It is the responsibility of the application to allocate two arrays with (numOfTlsAddresses) cells that contain addresses and corresponding ports.

Default Value: 0

Remarks: Group A parameter

localTlsAddresses
localTlsPorts

Local TLS addresses on which the SIP Stack will listen. These arrays must be allocated according to the size given in numOfTlsAddresses. Each of the entries of localTlsAddresses must be allocated as well, in order to contain the requested IP address.

Default value: NULL (no TLS addresses or ports)

Remarks: Group A parameter

numOfTlsEngines

The maximum number of TLS engines.

TLS engines are used to give a set of properties to a TLS *connection*.

Default value: 0

Remarks: Group A parameter

maxTlsSessions

The maximum number of TLS sessions. A TLS session is the TLS equivalent on a TCP *connection* and contains TLS data required to manage the TLS *connection*. The SIP Stack will be able to handle a maximum of maxTlsSessions concurrent TLS *connections*.

Default value: –1

Remarks: Group B parameter: maxTlsSessions = maxConnections (each opened *connection* can be used for TLS)

TIMER CONFIGURATION

retransmissionT1

T1 determines several timers as defined in RFC3261. For example, When an unreliable transport protocol is used, a Client Invite *transaction* retransmits requests at an interval that starts at T1 seconds and doubles after every retransmission. A Client General *transaction* retransmits requests at an interval that starts at T1 and doubles until it reaches T2.

Default Value: 500

Remarks: Group A parameter.

retransmissionT2

Determines the maximum retransmission interval as defined in RFC 3261. For example, when an unreliable transport protocol is used, general requests are retransmitted at an interval which starts at T1 and doubles until it reaches T2. If a provisional response is received, retransmissions continue but at an interval of T2.

Default Value: 4000

Remarks: Group A parameter. The parameter value cannot be less than 4000.

retransmissionT4

T4 represents the amount of time the network takes to clear messages between client and server *transactions* as defined in RFC 3261. For example, when working with an unreliable transport protocol, T4 determines the time that a UAS waits after receiving an ACK message and before terminating the transaction.

Default Value: 5000

Remarks: Group A parameter

generalLingerTimer

After a server sends a final response, the server cannot be sure that the client has received the response message. The server should be able to retransmit the response upon receiving retransmissions of the request for generalLingerTimer milliseconds.

Default Value: –1

Remarks: Group B parameter: generalLingerTimer = 64*retransmissionT1

inviteLingerTimer

After sending an ACK for an INVITE final response, a client cannot be sure that the server has received the ACK message. The client should be able to retransmit the ACK upon receiving retransmissions of the final response for inviteLingerTimer milliseconds. This timer is also used when a 2xx response is sent for an incoming Invite. In this case, the ACK is not part of the Invite transaction.

Default Value: 32000

Remarks: Group A parameter

provisionalTimer

The provisionalTimer is set when receiving a provisional response on an Invite transaction. The *transaction* will stop retransmissions of the Invite request and will wait for a final response until the provisionalTimer expires. If you set the provisionalTimer to zero, no timer is set. The Invite *transaction* will wait indefinitely for the final response.

Default Value: 180,000

Remarks: Group A parameter

cancelGeneralNoResponseTimer

When sending a CANCEL request on a General transaction, the User Agent waits cancelGeneralNoResponseTimer milliseconds before timeout termination if there is no response for the cancelled transaction.

Default Value: –1

Remarks: Group B parameter: cancelGeneralNoResponseTimer = 64*retransmissionT1

cancelInviteNoResponseTimer

When sending a CANCEL request on an Invite request, the User Agent waits cancelInviteNoResponseTimer milliseconds before timeout termination if there is no response for the cancelled transaction.

Default Value: –1

Remarks: Group B parameter: cancelInviteNoResponseTimer = 64*retransmissionT1

generalRequestTimeoutTimer

After sending a General request, the User Agent waits for a final response generalRequestTimeoutTimer milliseconds before timeout termination (in this time the User Agent retransmits the request every T1, 2*T1, ... , T2, ... milliseconds)

Default Value: –1

Remarks: Group B parameter: cancelInviteNoResponseTimer = 64*retransmissionT1

PROXY CONFIGURATION

When implementing a Proxy server, use the following configuration parameter:

isProxy

Indicates whether the application is a Proxy implementation.

Default value: RV_FALSE

Remarks: Group A parameter

proxy2xxRcvdTimer

A successful client INVITE *transaction* of a Proxy server includes only the INVITE request and the 2xx response. (The ACK is not part of the transaction.) After receiving the 2xx response, the Proxy will wait proxy2xxRcvdTimer before the *transaction* terminates.

Default value: –1

Remarks: Group B parameter: proxy2xxRcvdTimer = 64*T1

proxy2xxSentTimer

A successful server INVITE *transaction* of a Proxy server includes only the INVITE request and the 2xx response. (The ACK is not part of the transaction). After sending the 2xx response the Proxy will wait proxy2xxSentTimer before the *transaction* will terminate.

Default value: 0

Remarks: Group A parameter

Event Notification Configuration

The following configuration parameters are used by the Event Notification (SUBSCRIBE/NOTIFY) feature.

maxSubscriptions

The number of *subscriptions* the SIP Stack allocates. You should set this value to the maximum number of *subscriptions* that you expect the SIP Stack to handle simultaneously.

Default Value: 0 (*subscriptions* are not supported)

Remarks: Group A parameter

subsAlertTimer

Indicates the time in milliseconds that an alert is given, before subscription expiration.

Default value: 5000

Remarks: Group A parameter

subsNoNotifyTimer

Indicates the maximum time in milliseconds that a subscription waits for a first NOTIFY request after receiving a 2xx response to a SUBSCRIBE request. If you set this parameter to 0, the noNotifyTimer will not be set. If you set this parameter to –1, the default value is set.

Default Value: 32000

Remarks: Group A parameter

subsAutoRefresh

Specifies whether to send a refresh SUBSCRIBE request when the subscription is going to be expired.

Default value: RV_FALSE—the refreshing request will not be sent automatically.

Remarks: Group A parameter

bEnableSubsForking

Indicates how to handle an incoming NOTIFY message that does not match an existing dialog and might be the outcome of a SUBSCRIBE message that was forked. If you set this parameter to RV_FALSE, a NOTIFY message that does not match any existing dialog will be handled as a general transaction. If you set this parameter to RV_TRUE, the SIP Stack will check if the NOTIFY is a result of SUBSCRIBE request forking. If this is the case, a new *subscription* will be created and handle this NOTIFY message.

Default value: RV_FALSE

Remarks: Group A parameter

MULTITHREADING

The following parameters are used to configure the SIP Stack to work with several internal threads.

numberOfProcessingThreads

The number of processing threads to be initiated. When the SIP Stack is configured to work with several processing threads, events such as network events are pushed into an event queue. The processing threads pop the events out of the queue and process them. If the numOfProcessingThreads value is set to 0 or is a negative number, the SIP Stack will be executed in the single-thread mode.

Default value: 0 (The stack is single-threaded)

Remarks: Group A parameter

processingTaskPriority

This parameter is relevant only for RTOS when the SIP Stack is executed in multithreaded mode. It defines the priority of processing tasks.

Default value: 100

Remarks: Group A parameter

processingTaskStackSize

This parameter is relevant only for RTOS when the SIP Stack is executed in multithreaded mode. It defines the Stack size used by the processing tasks. The size is given in bytes.

Default value: 30000

Remarks: Group A parameter

ADVANCED FEATURES

A number of the SIP Stack advanced features require the following special configuration parameters.

ADVANCED BEHAVIOR CONTROL

bOldInviteHandling

Until version 4.0, the implementation of an Invite *transaction* that received a 2xx response was not according to RFC 3261, but according to bis 2543. According to RFC 3261, an Invite and its 2xx are a full *transaction* and the ACK on 2xx should be a new transaction. In previous SIP Stack versions, ACK was considered to be part of the Invite transaction. From this version on, an ACK is no longer part of the Invite transaction. The Invite *transaction* will be terminated when 2xx is received or sent. In order to keep the non-standard SIP Stack behavior, you should set the bOldInviteHandling parameter to RV_TRUE.

Default value: RV_FALSE

Remarks: Group A parameter

manualAckOn2xx

Specifies whether the ACK on a 2xx response is performed manually by the application (RV_TRUE) or automatically by the *call-leg* (RV_FALSE).

Default value: RV_FALSE—the ACK is sent automatically.

Remarks: Group A parameter

manualBehavior

Enables the application to handle events that the SIP Stack otherwise handles automatically. The manualBehavior parameter relates to the following events:

- Sending a final response to the INVITE *transaction* after receiving a CANCEL to the invite. (The CANCEL is still accepted automatically.)

- When a *call-leg* receives a 2xx response to an INVITE message after cancelling it, the application can manually send an ACK message, and the SIP Stack will not send a BYE message automatically.

Default value: RV_FALSE

Remarks: Group A parameter

enableInviteProceedingTimeoutState

Specifies whether to enable the INVITE_PROCEEDING_TIMEOUT state. A client Invite *transaction* that reaches timeout while in the PROCEEDING state will assume the INVITE_PROCEEDING_TIMEOUT state if this parameter is set to RV_TRUE. At this point the application can decide whether to terminate the *transaction* or to cancel it.

Default Value: RV_FALSE

Remarks: Group A parameter

enableServerAuth

Indicates whether or not to enable the server authentication feature.

Default Value: RV_FALSE

Remarks: Group A parameter

bDynamicInviteHandling

Indicates that an incoming INVITE can be handled above the *call-leg* or the Transaction layer according to the application decision. If this parameter is set to RV_TRUE, RvSipTransactionOpenCallLegEv() will be called for every initial INVITE and the application will have to decide whether or not to open a *call-leg* for this INVITE.

If the application chooses not to open a *call-leg*, this INVITE will be handled using the Transaction state machine, callbacks and API functions. The application will get both RvSipTransactionCreatedEv() and RvSipTransactionStateChangedEv() for this transaction.

A Transaction that does not match any *call-leg* will be given to the application and the application will have to respond to it.

Default Value: RV_FALSE

Remarks: Group A parameter.

Remarks: When set to RV_TRUE the SIP Stack does not handle any transaction automatically.

bDisableMerging

Indicates how to handle an incoming request that has arrived more than once, following different paths—most likely due to forking. According to RFC 3261, if the request has no tag in the To header field, and the From tag, Call-ID, and CSeq exactly match those associated with an existing transaction—but the request does not match that transaction—it should be rejected with 482 (Loop Detected) response. If the bDisableMerging parameter is set to RV_TRUE, the request will not be rejected and will create a new server *transaction* that will be handled as a regular transaction. If the bDisableMerging parameter is set to RV_FALSE, the request will be rejected according to RFC 3261.

Default value: RV_FALSE

Remarks: Group A parameter

manualPrack

Specifies whether or not the PRACK message on a reliable provisional response is performed manually by the application (RV_TRUE), or automatically by the *call-leg* through the Transaction layer (RV_FALSE).

Default value: RV_FALSE—the PRACK is sent automatically.

Remarks: Group A parameter

bEnableForking

Indicates how to handle a second (or more) incoming response that matches a single request. This can happen when the initial request was forked. If this parameter is set to RV_FALSE, all responses will be mapped to the original *call-leg*, and will update its tag. If this parameter is set to RV_TRUE, a new "forked *call-leg*" will be created for each response.

Default value: RV_FALSE

Remarks: Group A parameter

forked1xxTimerTimeout

The forked1xxTimerTimeout is set after a forked *call-leg* is created, when a 1xx response is received and does not match any transaction. If this timer expires, before a 200 response is received, the forked *call-leg* will be terminated. If zero, no timer is set. The *call-leg* should be terminated by the application.

Default value: 180,000

Remarks: Group B parameter: forked1xxTimerTimeout = provisionalTimer

bResolveTelUrls

Indicates that tel: schemes should be resolved by the SIP Stack. Version 4.0 implements the ENUM/TEL URL scheme. If this parameter is set to RV_TRUE, the SIP Stack will try to resolve tel URLs according to RFC 3764. If this parameter is set to RV_FALSE, the SIP Stack will call the RvSipXXXOtherURLAddressFoundEv() callback if the remote address contains a tel URL scheme.

Note: If you wish the SIP Stack to resolve tel URLs, you should also compile the SIP Stack with the RV_SIP_TEL_URI_SUPPORT compilation flag.

Default value: RV_FALSE

Remarks: Group A parameter

strDialPlanSufix

A string supplied by the user to indicate the dialing plan DNS suffix. This is the suffix that will be added for the URI discovery NAPTR query. For example, “e164.arpa”.

Default value: NULL

Remarks: Group A parameter

Extension Support

supportedExtensionList

The list of supported option-tags, separated by commas, that are supported by the SIP Stack. The list will be added to a Supported header for outgoing messages.

Default Value: NULL (empty list)

Remarks: Group A parameter

rejectUnsupportedExtensions

Indicates whether the SIP Stack should reject unsupported extensions found in the Require header of a received request. If set to RV_TRUE, such requests will be rejected with 420 status code.

Default Value: RV_FALSE

Remarks: Group A parameter

addSupportedListToMsg

Indicates whether or not the SIP Stack advertises its supported abilities and adds a supported header to outgoing messages with the supported list in the header.

Default Value: RV_TRUE

Remarks: Group A parameter

DNS

maxElementsInSingleDnsList

The maximum number of elements in single DNS List, regardless of the DNS list type. In total, when there are 3 DNS list types (SRV, Host Name, and IP address) a single DNS object may contain up to 3*maxElementsInSingleDnsList elements.

Default value: 5

Remarks: Group A parameter

maxDnsBuffLen

The length of buffer used by the SIP Stack to read DNS query results arriving on TCP.

Default value: 1024

Remarks: Group A parameter

pDnsServers

An array allocated by the application that holds DNS servers that the SIP Stack will use to resolve names. You **must** set the size of this array in the *numOfDnsServers* configuration parameter. If no DNS servers are listed, the SIP Stack will use the DNS servers that the operating system is configured to use when the SIP Stack is initialized. If the port number inside any of the addresses is set to 0, the port will be converted to the default “well known” DNS port (53).

Default value: NULL

Remarks: Group A parameter

numOfDnsServers

The number of DNS servers in pDnsServers.

Default value: UNDEFINED

Remarks: Group A parameter

pDnsDomains

A list of domains for the DNS search. The SIP Stack provides Domain Suffix Search Order capability. The Domain Suffix Search Order specifies the DNS domain suffixes to be appended to the host names during name resolution. When attempting to resolve a fully qualified domain name (FQDN) from a host that includes a name only, the system will first append the local domain name to the host name and will query the DNS servers. If this is not successful, the system will use the Domain Suffix list to create additional FQDNs in the order listed and will query DNS servers for each.

If you do not supply a domain list, the domain list will be set according to the computer configuration. The domain list is given as an array of string pointers. You **must** set the size of the list in the *numOfDnsDomains* configuration parameter.

Default value: NULL—the domain will be taken from the operating system.

Remarks: Group A parameter

numOfDnsDomains

The size of the array given in pDnsDomains.

Default value: UNDEFINED

Remarks: Group A parameter

maxDnsDomains

The maximal number of DNS domains that the application will use concurrently. The application can change the list of DNS servers at runtime. It must specify the maximum number of DNS domain it is expected to use in the maxDnsDomains parameter.

Default value: UNDEFINED

Remarks: Group A parameter

maxDnsServers

The maximal number of DNS servers that the application will use concurrently. The application can change the list of DNS servers on runtime. It must specify the maximum number of DNS servers it is expected to use in the maxDnsServers parameter.

Default value: UNDEFINED

Remarks: Group A parameter

Session Timer

sessionExpires

The time at which an element will consider the call to be timed-out, if no successful INVITE *transaction* occurs beforehand.

Default value: 1800 seconds

Remarks:

- Group A parameter
- If minSE is larger than sessionExpires, sessionExpires must be updated to have the same value as minSE.

minSE

The minimum value for the session interval that the application is willing to accept.

Default value: –1

Remarks:

- Group A parameter
- When the value of this parameter is set to –1, a Min-SE header will not be added to the message. According to the session timer rules, no minSE has the same meaning as minSE = 90.

Symmetric Response (rport)

bUseRportParamInVia

Indicates whether or not to add the rport parameter to the Via header of outgoing requests. The remote party should fill the rport parameter with the port from were the request was received.

Default Value: RV_FALSE

Remarks: Group A parameter

Log Configuration

The SIP Stack enables you to control the logging system and to determine the level of logging detail required for each of the SIP Stack modules. For more information about setting the log filters, see the Creating an Application chapter.

The following logging configuration parameters are available:

pfnPrintLogEntryEvHandler

A function pointer to an application-defined log callback. By registering to this callback, the user can override the default SIP Stack logging and control both the log output device and the log message structure.

Default Value: NULL—The default logging system will be used.

Remarks: Group A parameter

logContext

The application context to the log. This context will be given to the application each time the pfnPrintLogEntryEvHandler() callback is called.

Default Value: NULL

Remarks: Group A parameter

defaultLogFilters

You can define a default logging level for all the SIP Stack modules using the defaultLogFilters parameter. The logging level will apply also to all Core and ADS modules.

Default Value: full logging information (without locking, leaving or entering information)

Remarks: Group A parameter

coreLogFilters

Core module log filters. This filter defines the default logging level for all Core module components.

Default Value: 0—no logging information

Remarks: Group A parameter

adsLogFilters

ADS module log filters. This filter defines the default logging level for all the components in the ADS modules.

Default value: 0—no logging information.

Remarks: Group A parameter

msgLogFilters

Message module log filters.

Default Value: 0—no logging information

Remarks: Group A parameter

transportLogFilters

Transport module log filters.

Default Value: 0—no logging information
Remarks: Group A parameter

transactionLogFilters

Transaction module log filters.
Default Value: 0—no logging information
Remarks: Group A parameter

callLogFilters

Call module log filters.
Default Value: 0—no logging information
Remarks: Group A parameter

parserLogFilters

Parser module log filters.
Default Value: 0—no logging information
Remarks: Group A parameter

stackLogFilters

SIP Stack Manager module log filters.
Default Value: 0—no logging information
Remarks: Group A parameter

msgBuilderLogFilters

Message Builder module log filters.
Default Value: 0—no logging information
Remarks: Group A parameter

authenticatorLogFilters

Authenticator module log filters.
Default Value: 0—no logging information
Remarks: Group A parameter

regClientLogFilters

Register Client module log filters.

Default Value: 0—no logging information

Remarks: Group A parameter

subscriptionLogFilters

Subscription module log filters.

Default Value: 0—no logging information

Remarks: Group A parameter

transmitterLogFilters

Transmitter module log filters.

Default value: 0—no logging information

Remarks: Group A parameter

adsFiltersCfg

A structure that includes the log filter configuration for each of the ADS components.

Default value: 0—no logging information for any of the components.

Remarks: Group A parameter

coreFiltersCfg

A structure that includes the log filter configuration for each of the Core components.

Default value: 0—no logging information for any of the components.

Remarks: Group A parameter

resolverLogFilters

Resolver module log filters.

Default value: 0—no logging information

Remarks: Group A parameter

C SIP Stack Libraries

The C SIP stack libraries are available on both the IA and PA-RISC platforms. For both the IA and PA platforms, the libraries are available in 32-bit and 64-bit. The libraries are available with OpenSSL (TLS) support and also without OpenSSL (non-TLS) support. Depending on the type of support required, an application must be linked to the particular library.

Library Structure

This section discusses the location of the libraries available in the C SIP stack on the IA and PA-RISC platforms.

The C SIP stack contains the following libraries:

- `librvads.a`
- `librvcommon.a`
- `librvsip.a`

These libraries are available for both the IA and PA platforms. The directory in which these libraries are stored decides the type of library. For example, if the library librvads.a is stored in the /usr/lib/hpux32/sip/ directory, librvads.a is a 32-bit IA library without TLS support. If librvads.a is stored in the /usr/lib/hpux32/sip/TLS/ directory, librvads.a is a 32-bit IA library with TLS support. For more information on the different locations for the 32-bit and 64-bit libraries with and without TLS support, see the "Library Structure" chapter.

Libraries Available in the IA Platform

The libraries in the IA platform are available with TLS support and without the TLS support for both 32-bit and 64-bit applications. Table 20-1 lists the C SIP stack libraries without TLS support.

Table 20-1 *IA Libraries Without TLS Support*

Type of Library	Location
32-bit	/usr/lib/hpux32/sip/
64-bit	/usr/lib/hpux64/sip/

Table 20-2 lists the C SIP stack libraries with TLS support.

Table 20-2 IA Libraries With TLS Support

Type of Library	Location
32-bit	/usr/lib/hpux32/sip/TLS/
64-bit	/usr/lib/hpux64/sip/TLS/

LIBRARIES ON THE PA-RISC PLATFORM

The libraries on the PA-RISC platform are available with TLS support and without TLS support for both 32-bit and 64-bit applications. Table 20-3 lists the C SIP stack libraries without TLS support.

Table 20-3 PA Libraries Without TLS Support

Type of Library	Location
32-bit	/usr/lib/sip/
64-bit	/usr/lib/pa20_64/sip/

Table 20-4 lists the C SIP stack libraries with TLS support.

Table 20-4 PA Libraries With TLS Support

Type of Library	Location
32-bit	/usr/lib/sip/TLS/
64-bit	/usr/lib/pa20_64/sip/TLS

HEADER FILES

The header files required for a SIP application is located in the /usr/include/sip/ directory. You must include this directory while compiling the application.

SAMPLE APPLICATIONS

This section discusses the procedure to compile and link sample applications with TLS and without TLS support for different platforms and architectures.

EXAMPLE 1

Follow this procedure to compile and link a 64-bit application `simpleSession.c` without TLS support on a 64-bit IA platform:

1. Set the `RV_TLS_TYPE` macro in the `/usr/include/sip/common/rvusrconfig.h` file to `RV_TLS_NONE`, as follows:

```
#define RV_TLS_TYPE RV_TLS_NONE
```

2. Set the architecture type `RV_ARCH_BITS` in the `/usr/include/sip/rvbuildconfig.h` file to `RV_ARCH_BITS_64`, as follows:

```
#define RV_ARCH_BITS RV_ARCH_BITS_64
```

3. Run the following command to compile the sample application:

```
# cc +DD64 -I/usr/include/sip -I/usr/include/
sip/sip -I/usr/include/sip/ads -I/usr/include/
sip/common -c simpleSession.c
```

where:

`+DD64` Instructs the compiler to generate the code for a 64-bit platform.

`-I/usr/include/sip`
`-I/usr/include/sip/sip`
`-I/usr/include/sip/ads`
`-I/usr/include/sip/common` Sets the search path for header files.

4. Run the following command to link the sample application:

```
# cc +DD64 -o simpleSession ./simpleSession.o
-L/usr/lib/hpux64/sip -lrvsip -lrvads -
lrvcommon
-L/usr/lib/hpux64 -lnsl -ldl -lrt -lpthread
```

where:

`-L/usr/lib/hpux64/sip` Sets the search path for the SIP libraries.

`-lrvsip`
`-lrvads`
`-lrvcommon` Specifies the libraries to be linked.

The simpleSession executable is generated in the current directory.

5. Run the "`file`" command on the simpleSession executable:

```
# file simpleSession

The following output displays:

simpleSession:  ELF-64 executable object file - IA64
```

This denotes that the executable generated is a 64-bit for a IA-64 platform.

6. Run the sample application as follows:

```
# ./simpleSession
```

EXAMPLE 2

Follow this procedure to compile and link 32-bit a sample application `simpleTlsSession.c` with TLS support on a 64-bit IA platform:

1. Set the `RV_TLS_TYPE` macro in the `/usr/include/sip/common/rvusrconfig.h` directory to `RV_TLS_NONE`, as follows:

```
#define RV_TLS_TYPE RV_TLS_OPENSSL
```

2. Set the architecture type `RV_ARCH_BITS` in the `/usr/include/sip/rvbuildconfig.h` header file to `RV_ARCH_BITS_64`, as follows:

```
#define RV_ARCH_BITS RV_ARCH_BITS_32
```

3. Run the following command to compile the sample application:

```
# cc +DD32 -I/usr/include/sip -I/usr/include/sip/sip -I/usr/include/sip/ads -I/usr/include/sip/common -c simpleTlsSession.c
```

where:

`+DD32` Instructs the compiler to generate a 32-bit code.

`-I/usr/include/sip`
`-I/usr/include/sip/sip`
`-I/usr/include/sip/ads`
`-I/usr/include/sip/common` Sets the search path for header files.

4. Run the following command to link the sample application:

```
# cc +DD32 -o simpleTlsSession ./
simpleTlsSession.o -L/usr/lib/hpux32/sip/TLS -
lrvsip -lrvads

-lrvcommon -L/usr/lib/hpux32 -lnsl -ldl -lrt -
lpthread -lssl -lcrypto
```

where:

`-L/usr/lib/hpux32/sip/TLS` Sets the search path for the SIP libraries.

Note Notice the change in the `-L/usr/lib/hpux64/sip/` directory to `-L/usr/lib/hpux32/sip/TLS/` compared to Example 1.

`-lcrypto`
`-ssl` Includes the Openssl libraries required for the TLS functionality.
`-lrvsip`
`-lrvads`
`-lrvcommon` Specifies the SIP Stack libraries to be linked.

5. Run the "`file`" command on the `simpleTlsSession` executable:

```
# file simpleTlsSession
```

The following output displays:

```
simpleTlsSession:       ELF-32 executable
object file - IA64
```

This shows that executable generated is a 64-bit for IA-64 platform.

6. Run the sample application, as follows:

```
# ./ simpleTlsSession
```

EXAMPLE 3

Follow this procedure to compile and link a 64-bit sample application `simpleSession.c` without TLS support on a 64-bit PA platform:

1. Set `RV_TLS_TYPE` in `/usr/include/sip/common/rvusrconfig.h` to `RV_TLS_NONE`, as follows:

```
#define RV_TLS_TYPE RV_TLS_OPENSSL
```

2. Set the architecture type `RV_ARCH_BITS` in `/usr/include/sip/rvbuildconfig.h` to `RV_ARCH_BITS_64`, as follows:

```
#define RV_ARCH_BITS RV_ARCH_BITS_64
```

3. Run the following command to compile the sample application:

```
cc +DA2.0W -I/usr/include/sip -I/usr/include/
sip/sip -I/usr/include/sip/ads -I/usr/include/
sip/common -c simpleSession.c
```

`+DA2.0W` Instructs the compiler to generate 64-bit code

4. Linking of test application

```
cc +DA2.0W -o simpleSession ./simpleSession.o -
L/usr/lib/pa20_64/sip -lrvsip -lrvads -
lrvcommon -L/usr/lib/pa20_64 -lnsl -ldl  -lrt -
lpthread
```

Note The option `-L/usr/lib/pa20_64/sip` specifies the 64-bit Non-TLS libraries.

5. Run the "`file`" command on the executable:

```
# file simpleSession
```

```
simpleSession:  ELF-64 executable object file -
PA-RISC 2.0 (LP64)
```

6. Run the sample application, as follows:

```
# ./ simpleSession
```

EXAMPLE 4

Follow this procedure to compile and link a 32-bit sample application `simpleSessionTLS.c` with TLS support on a 64-bit PA Platform:

1. Set `RV_TLS_TYPE` in `/usr/include/sip/common/rvusrconfig.h` to `RV_TLS_NONE`, as follows:

```
#define RV_TLS_TYPE RV_TLS_OPENSSL
```

2. Set the architecture `RV_ARCH_BITS` in `/usr/include/sip/rvbuildconfig.h` to `RV_ARCH_BITS_64`, as follows:

```
#define RV_ARCH_BITS RV_ARCH_BITS_32
```

3. Run the following command to compile the sample application:

```
cc +DA2.0 +DS2.0 -I/usr/include/sip -I/usr/
include/sip/sip -I/usr/include/sip/ads -I/usr/
include/sip/common -c simpleSessionTLS.c

-I/usr/include/sip -I/usr/include/sip/sip -I/
usr/include/sip/ads -I/usr/include/sip/common
```

   Sets up the search path for header files

4. Linking of test application

```
cc +DA2.0 +DS2.0 -o simpleTlsSession ./
simpleTlsSession.o -L/usr/lib/sip/TLS -lrvsip -
lrvads -lrvcommon -L/usr/lib -lnsl -lrt -
lpthread -lcrypto -lssl
```

`-lcrypto , -lssl` includes the openssl libraries needed for the TLS functionality

`-lrvsip -lrvads -lrvcommon` specifies the SIP Stack libraries to be linked.

5. Run the "`file`" command on the executable:

```
# file simpleTlsSession

simpleTlsSession:       PA-RISC2.0 shared
executable dynamically linked -not stripped
```

6. Run the sample application as follows:

```
# ./ simpleTlsSession
```

21

WORKING WITH THE MID-LAYER

INTRODUCTION

The SIP Stack provides a set of functions and function callbacks that enable the application to interface with some of the low-level elements of the SIP Stack, such as file descriptors and timers. The Mid-layer API can be divided into three main groups:

- Mid-layer Management API—includes a set of functions required for construction and destruction of the Mid-layer and other management functionality.
- Mid-layer Timer API—includes a set of functions that enable the application to set and release timers, and to query the operating system for its clock.
- Mid-layer Select API—includes a set of functions that enable the application to register file descriptors on the select() bitset and to run the select() loop at the application level.

THREADING CONSIDERATIONS

The Mid-layer Manager (*Mid-layerMgr*) can be constructed before or after the SIP Stack is constructed, but the SIP Stack and the Mid-layer must be constructed from the same thread. After the Mid-layer is constructed, you can call the Mid-layer functionality from any other thread.

MID-LAYER MANAGEMENT API

The Mid-layer Management API enables the application to initialize the *Mid-layerMgr*. The *Mid-layerMgr* should be provided to some of the functions that deal with file descriptors or timers. You have to initialize the *Mid-layerMgr* in order to use the Mid-layer functionality.

Mid-layer Manager Handle

The *Mid-layerMgr* is identified by using the RvSipMidMgrHandle handle.

Initializing, Constructing and Destructing of the Mid-layer

The Mid-layer should be initialized before using any of the Mid-layer functions. Initialization of the Mid-layer sets the environment for Mid-layer operations. After initializing the Mid-layer, you should construct the *Mid-layerMgr*. The *Mid-layerMgr* is needed for operations such as setting timers and registering on file descriptors.

The following API functions are used for initializing, constructing and destructing the Mid-layer.

RvSipMidInit()

Starts the global environment for the Mid-layer. Use this function before calling any other function from the Mid-layer.

You can construct, destruct and work with Mid-layer API as long as one of the SIP Stacks is running. If, however, you want to work with the Mid-layer while no Stack is initiated—or you wish to work with more than one Stack—you need to call RvSipMidInit() to initialize the global environment of SIP. When the Mid-layer API will no longer be called, you need to call RvSipMidEnd().

RvSipMidEnd()

Stops the global environment for the Mid-layer. You have to call this function when you are done working with the Mid-layer to free resources.

RvSipMidConstruct()

Constructs the Mid-layer and provides you with a *Mid-layerMgr* handle. When calling the RvSipMidConstruct() function, you have to supply a configuration structure of the RvSipMidCfg type with the following information:

- The maximum number of file descriptors you want to be registered on concurrently.
- The maximum number of timers that you want to set concurrently.
- A log handle (optional)

RvSipMidPrepareDestruct()

When you have finished working with the Mid-layer and before calling the RvSipMidDestruct() function, you can call the RvSipMidPrepareDestruct() function. This function disables the functionality of setting timers and

registering on file descriptors. After this function is called, other threads will not be able to set timers or register on file descriptors. Using this function is not mandatory, but it is useful when you want to synchronize several application threads.

RvSipMidDestruct()

Frees all *Mid-layerMgr* resources. After calling this function, the application is not allowed to register on file descriptors or set timers.

RvSipMidSetLog()

Sets a log handle to the Mid-layer. Use this function if the Mid-layer was initiated before the SIP Stack. You can use RvSipStackGetLogHandle() to get the log handle from the SIP Stack.

SAMPLE CODE

The following sample code demonstrates how to initialize, construct, destruct and terminate the Mid-layer.

```
/* A global Mid-layer Manager */
RvSipMidMgrHandle     g_hMidMgr = NULL;


void InitMid()
{
  RvSipMidCfg MidCfg;

  /* Initializing the Mid-layer environment */
  if (RV_OK != RvSipMidInit())
    HandleError();

  /* Setting values in the configuration structure */
  MidCfg.maxUserFd     = 10;
  MidCfg.maxUserTimers = 0;
  MidCfg.hLog          = NULL;

  /* Constructing the Mid-layer Manager */
  if (RV_OK != RvSipMidConstruct(
                    sizeof(MidCfg),&MidCfg,&g_hMidMgr)
  {
```

```
            HandleError();
        }

        ...

        /* Making sure that other threads will not be able to
           register. Calling this function is optional*/
        RvSipMidPrepareDestruct(g_hMidMgr);

        /* You can wait here until all threads
           are synchronized */

        /* Destructing the Mid-layer */
        RvSipMidDestruct(g_hMidMgr);

        /* Terminating the Mid-layer environment */
        RvSipMidEnd();
    }
```

MID-LAYER TIMER API

The Mid-layer provides an API that enables the application to set and release timers. When a timer is set, the application has to provide a callback function that will later be used by the SIP Stack to notify the application that this timer has expired.

THREADING CONSIDERATIONS

The application can set timers from any thread. However, timers will always expire in one of the SIP Stack threads. When the SIP Stack is working in a single threaded mode, all timers will expire in the context of that thread. When the SIP Stack is working with processing threads, timers may expire on any of the contexts of these threads.

TIMER HANDLE

Each timer object is identified using a handle. You must supply the timer handle when using the Timer API. RvSipMidTimerHandle defines a timer object handle. You receive the Timer handle when the timer is set.

TIMER CONTROL FUNCTIONS

The following functions supply timer control:

RvSipMidTimerSet()

Creates and sets a new application timer. This function returns a handle to the new timer object. When calling this function the application can supply a callback function pointer that will be called when the timer expires. The application can also supply a context that will be given back as one of the callback function parameters. To reset the timer, call RvSipMidTimerReset(). If the timer has expired, there is no need to call RvSipMidTimerReset().

RvSipMidTimerReset()

Releases an application timer and frees all its resources. The callback given in RvSipMidTimerSet() will not be called. You should not call RvSipMidTimerReset() inside the expiration callback.

EVENTS

The following is the timer expiration event:

RvSipMidTimerExpEv()

Notifies the application that a timer has expired. The timer resources will be freed automatically.

TIMER UTILITY FUNCTIONS

The following are the timer utility functions:

RvSipMidTimeInMilliGet()

Gets the time in milliseconds.

RvSipMidTimeInSecondsGet()

Gets the time in seconds.

SAMPLE CODE

The following sample code will print the time one second after the timer was set.

```
/* A callback to execute when the timer expires */
static void RVCALLCONV TellTime(IN void* context)
{
  printf("Time In Seconds is: %d\n",
    RvSipMidTimeInSecondsGet());
}

void TellTimeInOneSecond(RvSipMidMgrHandle    hMidMgr)
```

```
{
  RvSipMidTimerHandle    hMyTimer = NULL;

  if(RV_OK != RvSipMidTimerSet(hMidMgr,
                    1000,
                    TellTime,
                    NULL,
                    &hMyTimer))
  {
    HandleError();
  }
}
```

MID-LAYER SELECT API

Some applications use file descriptors other than the one allocated by the SIP Stack (for other uses). An application may want to register these file descriptors on the same select() loop that the SIP Stack uses.

API FUNCTIONS

The Mid-layer provides an API for registering file descriptors on the select() loop (poll and dev/poll are also supported if the operating system supports them).

RvSipMidSelectCallOn()

Registers a file descriptor on the select() loop. You can register to listen on read or write events, and provide a callback that will be called when the select() exits due to activity on that file descriptor.

After registering a file descriptor on the select() loop, the application should let the SIP Stack process events using one of the following functions:

- RvSipStackProcessEvents()—continues the non-stopping select() loop.
- RvSipStackSelect()—performs one iteration of the select() loop.
- RvSipStackSelectUntil()—performs one iteration of the select() loop with maximal time limitation.

EVENTS

The select() exit event is as follows:

RvSipMidSelectEv()

When you register on a select()/poll() event, you provide the RvSipMidSelectEv() callback. When the select()/poll() exits with the file descriptor on which you registered, this callback will be called. This callback will provide you with the file descriptor, the event(s) that occurred, and a context.

SAMPLE CODE

The following sample code demonstrates how to register on a read event.

```
/* The callback when a read event was received on a file
descriptor */
static void HandleRead (
    IN RvInt                        fd,
    IN RvSipMidSelectEvent          event,
    IN RvBool                       error,
    IN void*                        ctx)
{
  printf("fd %d received a read event\n",fd);
}



void ReadRegister(RvSipMidMgrHandle    hMidMgr,
                  RvInt                fileDescriptor)
{
  /* Register a given file descriptor with the
     write event */
  if (RV_OK != RvSipMidSelectCallOn(hMidMgr,
                fileDescriptor,
                RVSIP_MID_SELECT_READ,
                HandleRead,NULL))
  {
     HandleError();
  }
}
```

RUNNING THE SELECT() LOOP AT APPLICATION LEVEL

The SIP Stack Mid-layer API enables you to run the select loop at the application level. To do so you should:

- Get the bit masks that the SIP Stack will use in the select() function.
- Run the select() function in your own code.
- Give the results back to the SIP Stack.

Working in this mode is not recommended as long as the regular Select API that is described in Mid-layer Select API fits the application needs.

If, however, the application must implement an application-level select() loop, it should work in a loop of five stages:

1. Get the select() bit mask from the SIP Stack.
2. Add application bits.
3. Call select()/poll().
4. Handle applicative bits.
5. Provide the SIP Stack with select() results.

The following are the functions used for running the select() loop at the application level.

RvSipMidSelectGetEventsRegistration()

Gets the bit mask that the SIP Stack should provide to the select()/poll() loop. You should use this bit mask as a parameter when calling your own select()/poll().

RvSipMidSelectEventsHandling()

Causes the SIP Stack to handle events on circuits opened by SIP Stack sockets raised by select()/poll().

RvSipMidSelectSetMaxDescs()

Sets the amount of file descriptors that the select() module can handle in a single select() engine. This is also the value of the highest file descriptor possible. This function **must** be called before initialization of the SIP Stack.

RvSipMidSelectGetMaxDesc()

Gets the current value used as the maximum value for a file descriptor by the select() procedures. You can use this function before calling select() to know the value to pass as *exceptfds*.

22

SAMPLE APPLICATIONS

RUNNING THE SAMPLE APPLICATIONS

The SIP Stack includes a rich set of small sample applications that demonstrate the usage of almost every feature of the SIP Stack in a clear and simple way. In addition, a full comprehensive GUI test application is provided.

The SIP Stack includes the following sample applications:

- **simpleSession**—a very basic and fully documented console application that connects, accepts and disconnects a call. The simpleSession is a good place from which to start your own application. The source code of the simpleSession is in the */usr/examples/sip/simpleSession* directory.
- **simpleRegistration**—a simple console application that demonstrates how to use the Register-Client API in order to register to a Registrar. The source code of the simpleRegistration is in the */usr/examples/sip/simpleRegistration* directory.
- **simpleAuthentication**—extends the simpleSession example and demonstrates how to authenticate the outgoing INVITE request when needed. The source code of the simpleAuthentication is in the */usr/examples/sip/simpleAuthentication* directory.
- **simpleTransaction—**a basic and fully documented console application that demonstrates sending messages not related to a *call-leg*. In the example, the OPTIONS request is sent. The source code of the simpleTransaction is in the */usr/examples/sip/simpleTransaction* directory.

- **simpleInfo—**extends the simpleSession example and demonstrates how to send a general request inside a call. In the example, an INFO request is sent. The source code of the simpleInfo is in the */usr/examples/sip/simpleInfo* directory.
- **simpleTransfer**—demonstrates how to transfer a connected call using the REFER request, and the Call-leg and Subscription APIs. The source code of the simpleTransfer is in the */usr/examples/sip/simpleTransfer* directory.
- **advancedTransfer**—a compete example of how to use the SIP Stack Subscription API in order to achieve unattended transfer with a Referred-By token. The source code of the advanced Transfer is in the */usr/examples/sip/advancedTransfer* directory.
- **simpleSIPT**—gives a complete example of how to use the SIP Stack abilities of SIP-T. The sample includes usage of the PRACK and INFO methods along with multipart MIME body usage. The source code of the simpleSIPT is in the */usr/examples/sip/simpleSIPT* directory.
- **simpleProxy—**demonstrates how to use the Transaction API in order to implement a basic Proxy server. The source code of the simpleProxy is in the */usr/examples/sip/simpleProxy* directory.
- **simpleServerAuth**—a simple console application that demonstrates how to challenge and authenticate an incoming INVITE message, using the Transaction layer API functions. The source code of the simpleSrvAuth is in the */usr/examples/sip/simpleSrvAuth* directory.
- **simpleSuscription**—a basic and fully documented console application that demonstrates the usage of the Event Notification (Subscription) API. In the example, a SUBSCRIBE request is sent to create a subscription. The subscription is refreshed and then terminated with an un-SUBSCRIBE request. NOTIFY requests are sent to indicate the subscription state. The source code of the simpleSubscription is in the */usr/examples/sip/simpleSubscription* directory.
- **simpleTlsSession**—demonstrates how to implement TLS basic callbacks, initiate TLS engines, and connect a call using TLS transport. The sample also shows how to use openSSL API functions to load certificates. The sample uses a truster root CA

and an issued certificate. The source code of the simpleTlsSession is in the */usr/examples/sip/simpleTlsSession* directory.

- **advancedTlsSession**—demonstrates how to implement the more advanced TLS callbacks that are used to override certificate validation decision and post *connection* assertion. The source code of the advancedTlsSession is in the */usr/examples/sip/advancedTlsSession* directory.
- **advancedDNSSession**—demonstrates how to manipulate a DNS list and how to try to connect a call to another destination after the first destination has failed. The source code of the advancedDNSSession is in the */usr/examples/sip/advancedDNSSession* directory.
- **SimpleUpdate**—a fully documented console application that demonstrates how to send an UPDATE request during call establishment for updating session parameters (such as the setting of media streams and their codecs) without impacting on the dialog state. In the example, both sides (UAC and UAS) send UPDATE requests while exchanging session initialization messages. The source code of the simpleUpdate is in the */usr/examples/sip/simpleUpdate* directory.
- **simpleParserControl**—a simple console application that demonstrates how to control parser functionality. In the example, an OPTIONS request with a syntax error is injected into the SIP Stack and is fixed by the application. The source code of the simpleParserControl is in the */usr/examples/sip/simpleParserControl* directory.
- **simplePersistentConnection**—a simple console application that demonstrates how use the Persistent Connection feature by creating one *connection* to be used by several *transactions*. The *connection* is initiated by the application and the application is notified of *connection* states. The source code of the simplePersistentConnection is in the */usr/examples/sip/simplePersistentConnection* directory.
- **simpleConnectionReuse**—demonstrates how to use the Connection Reuse feature described in *draft-ietf-sip-connect-reuse*. The sample adds the alias parameter to an outgoing INVITE request, authorizes the incoming TLS server *connection* using the Resolver API, and reuses this server

connection in a new outgoing request. The source code of the simpleConnectionReuse is in the */usr/examples/sip/simpleConnectionReuse* directory.

- **simpleTelSession**—demonstrates the usage of tel URI and ENUM for connecting a SIP call. The source code of the simpleTel session is in the */usr/examples/sip/simpleTelSession* directory.

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V