SIP Server Platform

Programmer Guide

NOTICE

© 2002-2008 RADVISION Ltd. All intellectual property rights in this publication are owned by RADVISION Ltd. and are protected by United States copyright laws, other applicable copyright laws and international treaty provisions. RADVISION Ltd. retains all rights not expressly granted.

No part of this publication may be reproduced in any form whatsoever or used to make any derivative work without prior written approval by RADVISION Ltd.

No representation of warranties for fitness for any purpose other than what is specifically mentioned in this guide is made either by RADVISION Ltd. or its agents.

RADVISION Ltd. reserves the right to revise this publication and make changes without obligation to notify any person of such revisions or changes. RADVISION Ltd. may make improvements or changes in the product(s) and/or the program(s) described in this documentation at any time.

If there is any software on removable media described in this publication, it is furnished under a license agreement included with the product as a separate document. If you are unable to locate a copy, please contact RADVISION Ltd. and a copy will be provided to you.

Unless otherwise indicated, RADVISION registered trademarks are registered in the United States and other territories. All registered trademarks recognized.

For further information contact RADVISION or your local distributor or reseller.

RADVISION SIP Server Platform version 3.5, March 2008

Publication 10

http://www.radvision.com

CONTENTS

About this Manual

1 *Technical Overview*

Introduction 1
What is a SIP Server? 1
Registrar Server 3
Redirect Server 3
Proxy Server 5
- Stateful and Stateless Proxies 6
- Request Validation 10
- Address Resolution 11
- Determining the Target-set 11
- DNS Resolution 12
- Stateful Message Forwarding 12
- Symmetric Record-Route 17
- Loose-Routing 18
- Forking 19
- Authentication 22
- Loop detection and Max Forwards 23
- Message Spiraling 24
- Outbound Proxy 24
- Policy 25

B2BAF 26
Events Server 27
- Event Package Name 28
- Subscribe Duration 28
- Authentication 28

Presence Server 29
SIP Servers Infrastructure Components 32
Database Access 32
Accounting 32

2 *RADVISION SIP Server Platform*

SIP Server Development Challenges 33
Minimizing Development Time and Cost 34
RADVISION SIP Server Platform 34
Standards Compliance 35
Interoperability 36
Robustness 36
Extensibility 37
Flexibility and Customization 38
Performance and Capacity 38
Maintainability 38
Portability 39
SIP Server Platform Capabilities 39
Standards compliance 39
General Capabilities 40
Proxy Capabilities 41
Registrar Capabilities 42
Redirect Capabilities 42
Advanced B2BUA Application Framework 42
SDP Stack 43
Events Server 43
Presence Server 43
Winfo Module 44
Non-SIP Infrastructure 44
SIP Server Platform Architecture 46
SIP Server Platform Coding 47
SIP Server Platform Package 47
Operating System Support 48
SIP Sever Application 48

3 *Using the SIP Server*

Introduction 51
SIP Server Contents 51
API Conventions 52
- Data Types 52
- Function Parameters 52
- API Functions 53
- Return Status Codes 53

Memory Management 53
Logging 53
Creating an Application 54
- Initialization 54
- Configuration 57
- Event Processing 59
- Getting Module Handles 60
- Registering Callbacks 61

SIP Server Working Mode 62
Threading model 63
- Working in Multithreaded Mode 63

4 *Working with the Proxy Core*

Introduction 65
Proxy Core Entities 66
- Proxy Policy Manager 66
- Proxy Core Object 66
- Proxy Core Transaction 66
- Proxy Transmitter 67

Request in Transaction Stateful Mode 67
Working with Handles 68
Proxy Core Policy Manager API 69
- Proxy Core Policy Manager Parameters 69
- Proxy Core Policy Manager Control 69

Proxy Core Object API 70
- Proxy Core Object Parameters 70

Proxy Core Object Control 73
Proxy Core Object Events 75
Proxy Core Transaction API 81
Proxy Core Transaction Parameters 81
Proxy Core Transaction Control 84
Proxy Core Transmitter API 85
Proxy Core Transmitter Parameters 86
Proxy Core Transmitter Control 88
Proxy Core Transmitter Events 89
Proxy Core Object State Machine 90
Proxy Transaction State Machine 101
Proxy Transmitter State Machine 102
Workflow Diagrams 106
Sample Code 112

5 *Working with the Register Server*

Introduction 117
Register Server Entities 117
Register Server Handles 118
Register Server Manager 119
Callback Function Structure 120
Register Server API 120
Register Server Parameters 120
Register Server Control 123
Register Server Events 123
Register Server Object State Machine 125
State Machine 127
Workflow Diagrams 128
Sample Code 132

6 *Working with SIP Messages*

Introduction 137
Working with Handles 138

Message Manager Object 138
Message Objects 139
Header Objects 140
SIP Stack Message API 141
Working with Headers 141
Working with SIP Messages 143
Reading and Modifying SIP Messages 143
Encoding and Parsing 144
Adding New Headers to a Message 147
Stand-alone Headers 148
Creating a Stand-alone Header 149
Setting a Stand-alone Header in a Message 149
Removing Headers from a Message 150
Creating a New SIP Message 151
Using Compact Form 153
“Set” Compact Form Functionality 153
“Get” Compact Form Functionality 153
Forcing Compact Form on the Entire Message 154
Memory Management 154
Handling Messages with Syntax Errors 155
Bad Syntax Parameter 155
Bad Syntax Header 155
Bad Syntax Start Line 156
Handling Bad Syntax Messages 156
Fixing Bad Syntax Messages 159
Bad Syntax Events (Callbacks) 159
Bad Syntax API 160
Working With Multipart MIME 165
Message Multipart Body Structure 166
Content-Type Header 168
Headers in the Body Part object 169
Parsing a Multipart Body 170
Creating a Multipart Body 172
Encoding a Multipart Body 174
Body String 174

7 *Working with the Transport Layer*

Introduction 177
Persistent Connection Handling 177
- Connection Hash 178
- Connection Owner 178
- Connection Termination Rule 178
- Connection Capacity Percent 179
- Application Connections 179
- Persistency Levels 179

Working with Connections 185
- Connections 185
- Transport Manager 186
- Working with Handles 186

Connection API 186
- Connection Parameters 186
- Connection Creation and Initialization 187
- Connection Control Functions 188
- Connection Events 189
- Connection States 190
- Client Connection State Machine 193

Server Connections 195
- Server Connection API 196
- Server Connection Events 196
- Server Connection States 197
- Server Connection State Machine 198
- Closing Server Connections 198

Server Connection Reuse 200
- Client Side 201
- Server Side 203
- Server Connection Reuse API functions and Events 203
- Authorizing a Server Connection 204

Using TCP Transport 205
Using TLS Transport 205
- TLS Connection Establishment 205
- TLS and SIP 206

RADVISION SIP Stack and TLS 206

TLS Stack Objects 206
TLS Engine 206
TLS Engine API 207
TLS Connection 210
TLS Connection API 210
TLS Connection Events 211
TLS Connection States 213
Configuration Parameters 216
Working with IPv6 Addresses 217
IPv6 Address Syntax 217
Scope ID 218
Compiling the SIP Stack with IPv6 218
IPv6 Addresses and SIP 218
Initializing the SIP Stack with IPv6 218
IPv6 Addresses and the Transport Address Structure 218
Transport Layer Raw Buffer and Message Monitoring 219
Raw Buffer Events 219
Dynamic Local Address 220
DLA API 221

8 *Working with DNS*

Introduction 223
Configuring DNS Parameters 223
DNS/SRV Tree 224
DNS/ENUM Record 225
RADVISION SIP Stack Implementation 225
State Machine and API Functions 227
Manipulating the DNS List Object 230
DNS List API 230
DNS Support for Call-legs, Subscriptions, Register-Clients, and Publish-Clients 234
Call-leg Layer 234
Subscriptions Layer 234
Register-Client Layer 235
Publish-Client Layer 235

Changing DNS List before Message is Sent 235
DNS Caching 236
DNS Caching Compilation Flags 236

9 *Location Database Server Component*

Introduction 239
Location Database Interface 239
Replacing the Location Database Implementation 240
Work Flow Diagram 243
Location Database Server Component API 244
Configuration Parameters 246

10 *Security Server Component*

Introduction 249
Server Authentication Object 250
Server Authentication Object API 250
Server Authentication Object Control 251
Server Authentication State Machine 252
Security Component Interface 255
Creating an Application Security Component 259
SIP Server Security Component Implementation 263
API Functions 263
Initializing the SIP Server Security Component 264
Configuration Parameters 266

11 *SIP Server STD Layer*

Introduction 269
XML Library 269
XML Manager Object 270
XML Document Object 270
XML Tag Object 270
XML Attribute Object 271

XML XPath Object 271
XML Library Handles 271
XML Library API 272
STD Allocator Interface 276
STD List 276
STD List API 276

12 *Memory Pool*

Introduction 281
Definitions 281
API Functions 281
Using the Memory Pool 282
Copying a Page into a Buffer 282
Copying a Buffer to a Page 283
Constructing and Destructing a Memory Pool 284

13 *Advanced Features*

Working with DNS 287
Handling sending failures 287
DNS related functions and Events 288
Working with IPv6 289
Multihomed Host 289
Receiving Messages 290
Sending Messages 290
Working with a Multihomed-host Proxy 291
General URL Scheme Support 294
Sending Requests to General URL Addresses 294
Handling Requests Received with General URL Addresses 295
Handling General URL with "Pres" Scheme 296
Session-Timer 297
Proxy Behavior 297
Configuration Parameters 297
API Functions 298

14 *SIP Server Configuration*

Introduction 299
Initialization 299
Configuration Structure 300
Setting Addresses within RvSipServerCfg 300
Configuration File 301
Configuration Parameters 301
SIP Server Configuration Parameters 301
Proxy Configuration Parameters 304
Register Configuration Parameters 310
Network Configuration Parameters 312
TLS Configuration Parameters 317
Multithreading Configuration Parameters 321
Server Dialog Configuration Parameters 322
B2B Configuration Parameters 323
Events Configuration Parameters 326
Log Configuration Parameters 334
Stack Configuration Parameters 342
LDAP Parameters 350
DNS Configuration Parameters 351
XML Manager Configuration 354
B2BAF Configuration Parameters 355
B2BAF Service Library Configuration Parameters 358

15 *Working with the Mid-Layer*

Introduction 359
Threading considerations 359
Mid-layer Management API 360
Mid-layer Manager Handle 360
Initializing, Constructing and Destructing of the Mid-layer 360
Mid-layer Timer API 362
Threading Considerations 362
Timer Handle 363
Timer control functions 363
Events 363

Timer Utility Functions 363
Mid-layer Select API 364
API Functions 364
Events 365
Running the select() Loop at Application Level 366
Mid-Layer Thread API 367
API Functions 367

16 *SIP Server Log*

Introduction 369
Source Identifiers 370
Log File 373
Log Messages 373
Message Structure 374
Log Configuration 374
Log Filters 375
Initializing the Log 375
Changing Log Configuration During Runtime 376
Controlling the Log 376
Compilation Log Control 378

17 *SIP Server Test Application*

Introduction 379
TCL/TK 379
Installation 379
ServerCommonApp Library 380
Configuration 380
User Interface 381
Test Application Main Window 382
Menu Bar and Toolbar 382
Tools Menu 382
Toolbar Options 413
Remotes Menu 413

Components Menu 439
Help Menu 439
Objects Window 440
Tab Area 441
General Tab 442
Lists Tab 446
Log Tab 448
E-DNS Tab 449
Setting Identifier Tab 451
Advance Tab 455
Body Tab 457
Configuration Tab 460
Application log Area 465
Server Components 467
Security Component 467
Location Database Server Component 468

18 *SIP Server Sample Applications*

Sample Applications 471
Sample Code 473

Index 475

About this Manual

The RADVISION SIP Server Platform is a comprehensive software platform for development of SIP Server applications. The SIP Server Platform includes an implementation of fully-standard server functionality that is controllable through multi-level APIs. SIP Server developers can quickly and effectively add SIP Server capabilities to their application by integrating the SIP Server library.

The *RADVISION SIP Server Platform Programmer Guide* describes how to use the SIP Server libraries for developing SIP Server applications.

Documentation

See the *SIP Server Platform Release Notes* for a complete list of documentation for the SIP Server Platform and SIP Server Platform Add-on Modules.

Additional Products

The SIP Server Platform is part of the RADVISION SIP product family which also includes these products:

RADVISION IMS SIP Toolkit (Stack)

The RADVISION IMS SIP Toolkit consists of SIP, SDP and RTP/RTCP Stacks, a suite of APIs, sample code and user documentation for developers who wish to build SIP applications. The Toolkit includes IMS support. For more information, see the *RADVISION SIP Stack Programmer Guide* and the *RADVISION IMS SIP Add-on Module Programmer and Reference Guide*.

PROLAB™ SIP TEST MANAGER

The ProLab™ SIP Test Manager is a highly scalable, feature-rich testing system that is suitable for use in different stages of the product development cycle. The ProLab™ SIP Test Manager can emulate a group of synchronized SIP entities, such as SIP user agents (UAs) and SIP Servers. For more information, see the *RADVISION ProLab™ SIP Test Manager User Guide.*

MULTI-MEDIA TERMINAL FRAMEWORK

The RADVISION Multi-media Terminal Framework provides VoIP application developers with an extensible framework for developing IP-based videophones, IP phones and IADs. It shields the application from the myriad tasks that a terminal must implement. It also enables application developers to concentrate on value-added services rather than on the intricacies of VoIP signaling, security, call state handling, and standard supplementary services. A complete terminal solution can be achieved by integrating the RADVISION Multi-media Terminal Framework components with third-party DSP services and telephony hardware. For more information, see the *RADVISION Multi-media Terminal Framework Programmer Guide*.

1

TECHNICAL OVERVIEW

INTRODUCTION

SIP Servers are essential network elements that enable SIP endpoints to exchange messages, register user location, and seamlessly move between networks. SIP Servers enable network operators to operate routing and security policies, authenticate users and manage user locations.

SIP Server applications may take many forms, but the SIP standard defines a few general types of server functionality that apply to all—Proxy, B2BUA, Events, Redirect and Registrar servers. These standard functionalities can be used according to the needs of the specific implementation. In addition, since SIP defines other event packages and templates, such as Winfo and Register, a SIP Server may function as an Events Server for handling the various SIP events, packages and templates.

With the advance of SIP, server logic has become increasingly complex. SIP Servers need to deal with varying network topologies (such as public Internet networks, cellular networks, broadband residential networks), complex routing policies, security and SIP extensions. SIP Servers often need to handle high message/transaction rates and yield real-time performance and scalability, high throughput, and low delay. This chapter discusses the protocol aspects of SIP Server behavior and the usage of the RADVISION SIP Server Platform to address the challenges of effective SIP Server development.

WHAT IS A SIP SERVER?

The SIP baseline specification RFC 3261 (previously RFC 2543bis) divides SIP Server functionality into the following parts:

- SIP Registrar Server—handles location registration messages.
- SIP Redirect Server—returns “contact this address” responses.
- SIP Proxy Server—forwards SIP requests and responses.

In addition to the functionality of the above three SIP Servers, the SIP specification and its related drafts also refers to other types of SIP Server:

- Back-to-Back User Agent (B2BUA)—acts as UA server to one side and as UA client to the other side.
- Presence Server—provides a Watcher with presence information of a Presentity, as defined in RFC 2778 and RFC 3856. Also provides Watcher information to Presentities, as defined in RFC 3857, RFC 3862 and RFC 3863.
- Events Server—in this document, the term "Events Server" is used for the network element, acting as Notifier, which receives SUBSCRIBE messages for different SIP Event types, such as Registration and MWI (Message Waiting Indication). This type of general Notifier functionality is defined in RFC 3265: SIP Specific Event Notification.

Note The document version number may change as newer versions are published.

The authors of SIP have made this functional separation for ease of understanding, and not necessarily as a guideline for implementers. In other words, it is acceptable to have one "box" do more than just one type of server functionality. For example, a SIP Server may benefit from having a routing capability to forward some SIP messages as a proxy and redirect others as a redirect server. There are existing SIP Server implementations that have a few types of functionalities in one product running in the same process space, while others prefer to distribute functionality, with different machines performing different server functions. In fact, the SIP standard does not prohibit a SIP UA (endpoint) from having server capabilities, since UAs may redirect requests or process registrations. It is important to understand that proxy, redirect and registrar are functionalities that may be put to use in any way that benefits the application.

The behavior of SIP Servers is discussed in RFC 3261, but you can find more detailed examples of message flows in "SIP Call Flow Examples" in RFC 3665 found at the IETF site or one of its mirror sites. (http://www.ietf.org/)

Note The document version number may change as newer versions are published.

Registrar Server

The SIP standard defines a registrar server as “a server that accepts REGISTER requests and places the information it receives in those requests into the location service for the domain it handles”. REGISTER requests are generated by clients in order to establish or remove a mapping between their externally known SIP address(es) and the address(es) they wish to be contacted at. The REGISTER request can also be used to retrieve all the existing mappings saved for a specific address.

The Registrar processes the REGISTER request for a specific set of domains. It uses a “location service”—an abstract location database—in order to store and retrieve location information. The location service may run on a remote machine and may be contacted using any appropriate protocol (such as LDAP). The SIP standard leaves this decision to the implementation. Some implementations may co-locate the location service and the registrar server on the same machine.

A registrar server may authenticate incoming REGISTER requests using the 401 (Unauthorized) response.

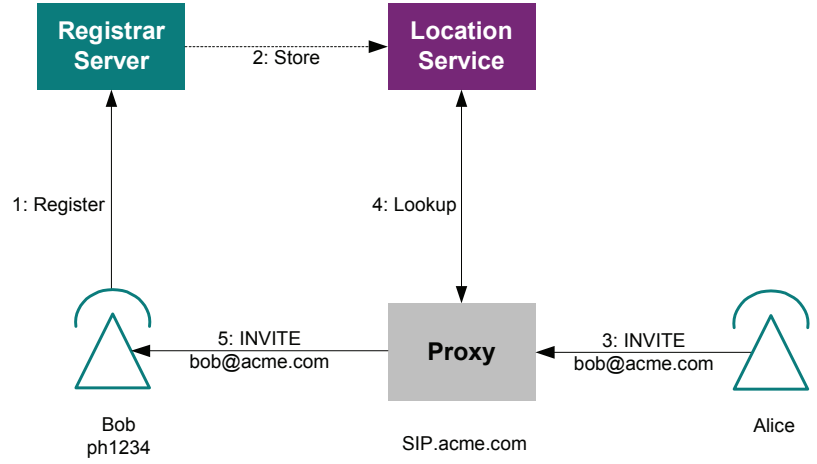


Figure 1-1 *Registration Process*

Redirect Server

Redirect server functionality is the simplest of the three functionalities. A redirect server receives SIP requests and responds with 3xx (redirection) responses, directing the client to contact an alternate set of SIP addresses. The alternate addresses are returned as Contact headers in the response message.

Table 1-1 shows the 3xx responses that are currently defined by SIP.

Table 1-1 *3xx Responses*

Response	Meaning
300 Multiple Choices	The address in the request was resolved to several choices, each with its own specific location, and the user (or UA) can select a preferred communication end point and redirect its request to that location. This status response is appropriate if the callee can be reached at several different locations and the server cannot, or prefers not, to proxy the request **Note:** 301 and 302 responses can also contain multiple Contact address. The difference is that they convey a more specific reason for the redirection.
301 Moved Permanently	The user can no longer be found at the address specified in the Request-URI (the destination address in the request), and the requesting client should retry at the new address given by the Contact header field.
302 Moved Temporarily	The user is temporarily available at a different address(es). The duration of validity of these addresses may be expressed in the Contact header.
305 Use Proxy	The requested destination address must be accessed through the proxy specified in the Contact field.
380 Alternative Service	The call was not successful, but alternative services are possible. The alternative services are described in the message body of the response. The use of this response code is still not defined in SIP and is for future use.

In most cases, 301 and 302 responses are used by redirect servers. 300 can also be used, although it is more ambiguous to the calling client.

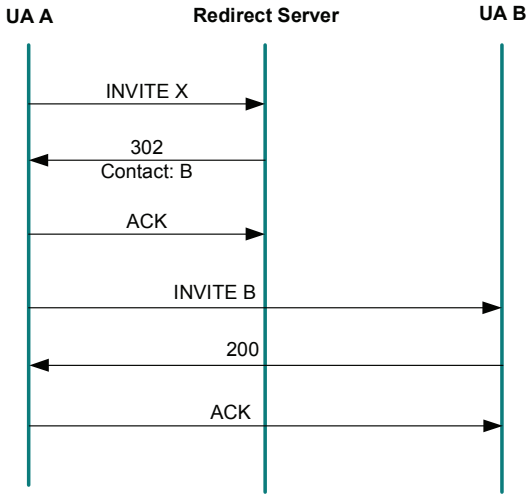


Figure 1-2 *Request Redirection*

The scenario in Figure 1-2 illustrates a redirection scenario. Note that the second INVITE request is generated with the same dialog identifiers, Call-ID, and To and From headers as the first INVITE request, but with a different CSeq value.

Redirection allows servers to push back routing information for a request in a response to the client, thereby aiding in locating the target of the request, while taking themselves out of the loop of further messaging for this transaction. Redirect servers typically are not aware of the state of dialogs (calls, subscriptions), only of the state of the individual transactions they are handling, making them transaction-stateful elements. Redirection is designed as a simple and quick procedure, allowing for redirect servers to be highly scalable and to yield high-performance. Redirect servers are sometimes used as a load balancing devices.

Redirect servers may request user authentication with the use of the 401 response (Proxy Authentication Required) as explained below.

PROXY SERVER

The SIP standard defines SIP proxies as “elements that route SIP requests to User Agent Servers (UAS) and SIP responses to User Agent Clients (UAC). A request may traverse several proxies on its way to a UAS. Each will make

routing decisions, modifying the request before forwarding it to the next element. Responses will route through the same set of proxies traversed by the request in the reverse order."

It is useful to view Proxy Servers as SIP-level routers that forward SIP requests and responses. However SIP proxies employ routing logic that is typically more sophisticated than just automatically forwarding messages based on a routing table. Routing decisions can be based on various criteria, such as the time of day, originator and termination.

The SIP standard allows proxies to perform actions such as validate requests, authenticate users, fork requests, resolve addresses, cancel pending calls, Record-Route and Loose-Route, and detect and handle loops. The versatility of SIP proxies allows the operator/system administrator to use the proxies for different purposes and in different locations in the network (such as edge proxy, core proxy and enterprise proxy). This versatility also allows for the creation of a variety of proxy policies, such as routing calls only for authenticated users that have no standing debt to the network service provider operating the proxy. Proxies can be placed at the network of the service provider or at the enterprise or SOHO premises. The 3GPP IMS architecture, for example, uses proxies known as Call Session Control Functions (CSCF in 3GPP terminology) of different kinds for various purposes—as the first hop server that communicates with the handset; as the element that accesses location repositories and application servers and triggers services; and as the interface element that communicates with proxies in foreign networks.

A proxy server is designed to be mostly transparent to UAs. Proxy servers are allowed to change messages only in specific and limited ways. For example, a proxy is not allowed to modify the SDP body of an INVITE. Apart from a few exceptions, proxies cannot generate requests at their own initiative. Therefore a proxy cannot terminate an existing call by generating a BYE request.

Stateful and Stateless Proxies

The SIP specification defines two types of SIP proxies:

- Stateful proxy
- Stateless proxy

Stateless Proxy

A stateless proxy is a "simple message forwarder", as described in the SIP standard. When receiving a request, the stateless proxy processes the request much like a stateful proxy, however the stateless proxy forwards the message in a stateless fashion—without saving any transaction context. This means that

once the message is forwarded the proxy "forgets" ever handling this message. Stateless forwarding allows for improved performance and scalability, but has some consequences:

- A stateless proxy cannot associate responses with forwarded requests because it retains no knowledge of the requests it has forwarded. Therefore, the proxy application cannot know if a transaction was successful or not.
- A stateless proxy cannot associate retransmissions of requests and responses with the previous instance of these messages. It processes retransmissions exactly as if this is the first copy of the message it received.

Note Stateless proxies need to use a routing algorithm that always routes copies of the same message to the same destination. If this rule is not enforced, a retransmission of an ACK message, for example, may be routed to a different UA than the one that returned the 200 response.

- If the message is lost, the proxy will not retransmit it. Retransmission is the responsibility of stateful UAs or proxies

Because of their high-throughput capabilities, stateless proxies are often used at the core of carrier and service provider networks assisting in forwarding SIP messages on the network. Stateless proxies may also be used as Load Balancers.

Note that once a proxy decides to return a non-2xx response, the proxy cannot do this statelessly and has to retain a transaction state.

STATEFUL PROXY

When stateful, the proxy processes transactions rather than individual messages. The proxy manages two types of transactions—server transactions to receive requests and return responses, and client transactions to send requests and receive responses. An incoming request is processed by a server transaction and then forwarded downstream by one or more client transactions (there may be more than one in the case of parallel forking, for example). An incoming response is received by the matching client transaction and forwarded back to the server transaction.

For example, in the RADVISION SIP Proxy implementation, associating between client and server transactions and managing the overall state of this request is the responsibility of the Proxy Core Object (*ProxyCoreObj*) of the RADVISION SIP Server Platform. The *ProxyCoreObj* chooses the destination address(es) and instantiates one or more client transaction objects accordingly. The *ProxyCoreObj* also collects the responses from the different client transactions and chooses the response(s) that will be sent upstream via the server transaction.

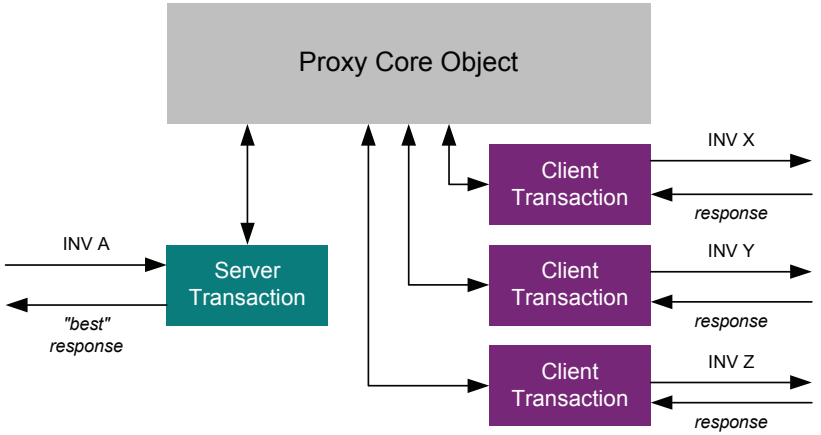


Figure 1-3 *RADVISION SIP Server Platform Stateful Proxy Model*

A stateful proxy is aware of the state of transactions and message history, and can therefore perform better-informed processing on incoming messages. For example, a stateful proxy can identify a retransmission of an incoming message and forward the message only in situations that require retransmission forwarding, whereas a stateless proxy cannot identify retransmissions and has to forward every message it encounters. A stateful proxy can also generate retransmissions in cases of message loss. In addition, a stateful proxy can locally process incoming CANCEL requests and generate CANCEL requests as needed. Forking is also more natural for a stateful proxy (especially sequential forking).

Statefulness, however, has the following drawbacks:

- **Memory consumption**

  The stateful proxy retains more memory per processed message than a stateless proxy, and for a longer duration. This has a negative impact on the maximal capacity of the proxy and limits the number of concurrent calls/transactions it can handle. Certain code optimizations that are SIP-specific can compensate for this and bring memory consumption to the level required by high-capacity proxies.

- **Throughput**

  A stateful proxy has to spend more CPU cycles on message processing—mapping messages to transactions, managing transaction state machines, processing transaction timers and associating client and server transactions. This extra processing reduces proxy capacity in terms of performance (maximal number of processed requests per second). For these reasons, developing a high-performance stateful proxy has proved to be non-trivial, and requires special optimizations in the design of the proxy. The ability to customize and to fine tune the proxy through flexible configuration is also crucial for achieving high performance.

- **Implementation complexity**

  A stateful proxy does more than just forward requests. Certain logic needs to be employed in order to deal with actions such as parallel forking (for example, choosing the best response), CANCEL, recursion on 3xx responses, and handling ACK for 2xx responses. The evolution of the SIP standard introduced numerous “hard-to-deal-with” special cases for proxies that require the use of a number of special techniques, such as storing hashed context information in certain request message fields and later retrieving the information from responses. All of these factors make the implementation of the proxy non-trivial and add special cases that have to be tested.

- **Underlying SIP Stack complexity**

  A proxy requires certain flexibilities from the underlying SIP Stack that a UA does not. This is especially true for the Transport and Transaction layers. This requires the SIP Stack to be more modular and export more layers of API than a UA-oriented stack.

REQUEST VALIDATION

Before routing a request, a SIP Server (proxy or redirect) needs to validate the request to make sure it can actually proceed with processing this message. The message has to pass the following validity checks:

- **Reasonable syntax check**

  The request must be well-formed enough to be handled by the server. However, this applies only to specific fields in the message which the server must process. All other parts should not be checked or fixed by the proxy.

- **URI scheme check**

  The URI scheme (for example, “pres” in pres:radvision.com) must be URI scheme the proxy understands and knows how to route. If not, the proxy must return a 416 (Unsupported URI Scheme) response.

- **Max-Forwards check**

  Max-Forwards is a message field that indicates how many more hops the message is allowed to traverse. Each proxy that handles the message decrements this number by one (similar to the TTL field in certain protocols). If the message contains a Max-Forwards value of zero, the proxy must return a 483 (Too many hops) response. This mechanism allows preventing a message from going into an endless loop between a set of proxies.

- **Loop Detection (Optional)**

  A proxy may check for loops by executing a loop detection algorithm on the Via list contained in the message. The proxy checks that it did not previously handle this message. If it did previously handle the message, it verifies that the message contains different values in the fields that influence routing decisions (such as Request-URI, From and To). If the proxy identifies a loop condition, it rejects the message with a 482 response code (Loop Detected). This check is now optional because the Max-Forwards field has become mandatory.

- **Proxy-Require**

  The client may indicate certain SIP extensions in the Proxy-Require fields that the proxy must support in order to successfully handle this request. The proxy must inspect this field and verify that it supports all the extensions listed in the field.

- **Authentication**

  If the SIP Server determines it has to authenticate the originator of the message, it has to make sure the message contains credentials that authenticate the user. If the message does not contain credentials or the credentials failed to authenticate the user, the proxy may return a 407 response containing a challenge. The authentication procedure is explained in more detail in the Authentication chapter.

Address Resolution

Once a proxy has validated an incoming request and decided to forward it, it must determine the destination(s) to which the message is to be forwarded before sending the messages. The proxy does two types of address resolution:

- Determining the target-set—the proxy resolves the request SIP destination address (Request URI) to a set of SIP addresses that are mapped to it in some way.
- DNS resolution—the proxy resolves each of the SIP destination addresses to a transport address of the form:

  {transport_protocol, transport address, transport address parameters—such as port}

Note A SIP request may be forwarded to more than one destination address. An example of such a case is a user that is simultaneously registered at several locations. In this case, the proxy may decide to fork the request either sequentially or in parallel.

Determining the Target-Set

The first process in address resolution, known as obtaining a target-set in the SIP specification, results in producing a set of SIP addresses. Essentially this stage maps from SIP address to SIP addresses. (Some addresses in the target-set may express explicit transport addresses by using the format sip:user@ip_address;transport=xyz.)

A target-set is obtained in one of two ways:

- **Predefined target-set**

  This is the simpler case, where the destination address of the request (Request-URI) is such that the proxy must automatically forward to the destination address without trying to resolve to other addresses. One such case is where the Request-URI is in a domain for which the SIP Server is not responsible. For example, a proxy sip:proxy1.acme.com which

is responsible for the domain acme.com receiving a request for sip:bob@example.com must proxy the request to sip:bob@example.com.

- **Target-set determined by proxy**

  If the target-set is not dictated by the message, the proxy may employ whatever mechanism it may wish to determine the target-set. Some options are:

  - Accessing a location service updated by a SIP registrar
  - Reading from a database
  - Consulting a presence server
  - Using other protocols
  - Performing algorithmic substitutions on the destination address

While the Request-URI is an important factor in determining the target set, the proxy may also choose to route based on other message fields, or on external parameters, such as time of day, network and server load.

DNS Resolution

Before forwarding a message, the proxy must resolve the message to concrete transport addresses which it can use in sending the message. Proxies, as other SIP entities, use the DNS mechanism described in RFC 3263 (SIP: Locating SIP Servers). Essentially, this is an algorithm that selectively uses SRV, NAPTR, A and AAAA DNS queries to map a given SIP address to a prioritized set of transport addresses of the form:

{transport_protocol, IP_address, port}

Using this advanced DNS scheme is recommended, since it allows building highly available, load-balanced SIP networks with the possibility of dynamic adjustment through DNS tables.

Stateful Message Forwarding

The message flows below illustrate the way a stateful SIP proxy forwards different types of messages. The term “downstream” means in the direction of the server (request direction); “upstream” means in the direction of the client (response direction). For the sake of simplicity, the scenarios show only non-forking proxies that Record-Route.

Non-INVITE Request

The scenario in Figure 1-4 shows Non-INVITE Request-Response message flow through multiple proxies.

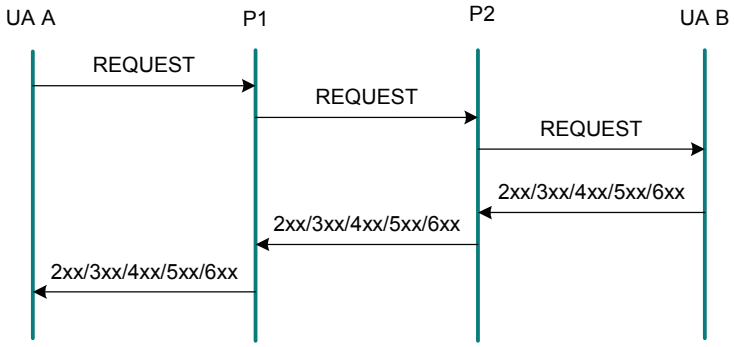


Figure 1-4 *Non-INVITE Request-Response Message Flow*

In the case of non-INVITE requests, such as BYE and REGISTER, proxy functionality is to forward requests and responses as they arrive. The proxy processes all final responses (2xx-6xx) the same way. Retransmitted requests are not forwarded by the proxy, but the proxy may retransmit requests based on its own retransmission timers. Retransmitted responses are forwarded by the proxy.

INVITE-ACK

When processing an INVITE request, a proxy typically responds with a 100 (Trying) response to stop INVITE retransmissions at the previous hop. All received 1xx (provisional) responses except 100 are forwarded to the previous hop. If the proxy does not receive a 100, it may retransmit the INVITE request as necessary.

The scenario in Figure 1-5 shows INVITE-non-2xx-ACK message flow through multiple proxies.

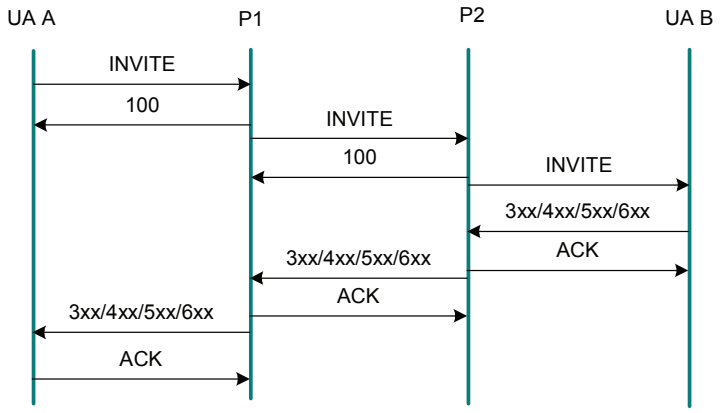


Figure 1-5 *INVITE-non-2xx-ACK Message Flow*

If a non-2xx response is received, the proxy generates the ACK request and forwards the response upstream. Upon reception of a retransmission of the response, the proxy will retransmit the ACK request.

INVITE-200-ACK

The scenario in Figure 1-6 shows INVITE-200-ACK message flow through multiple proxies.

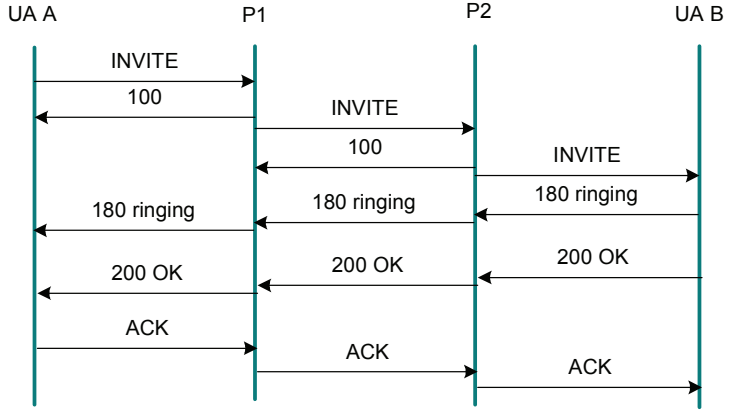


Figure 1-6 *INVITE-200-ACK Message flow*

A 2xx response to an INVITE request presents a special case. For purposes of call-setup robustness, it was decided that reliability (retransmissions) of 200 and ACK messages is to be handled end-to-end, rather than hop-by-hop. This means that upon receiving a 2xx response for an INVITE, the proxy forwards the message (and possibly any retransmission that follows) in a stateless fashion. It does not change state in any of its transactions and it does not generate an ACK request. Only the calling User Agent (UA A in the figure above) is allowed to ACK a 2xx response. The proxies forward the ACK (and possible retransmission) also in a stateless fashion. If either the 2xx or the ACK messages are lost along the way, it is the responsibility of the callee (UA B in the drawing) to retransmit the 2xx until it receives an ACK. This procedure assures that a call is established (and media can start flowing) only when both UAs have completed the handshake.

CANCEL

The scenario in Figure 1-7 shows CANCEL processing.

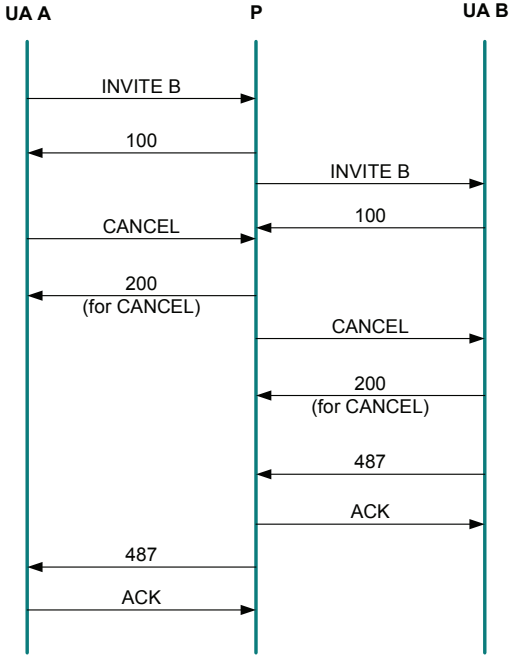


Figure 1-7 *CANCEL Processing*

A stateful proxy may choose to generate a CANCEL request for any pending INVITE request it has previously forwarded (subject to the rules of CANCEL as defined by the SIP standard). This capability is put to use in parallel forking as explained later in Parallel Forking on page 20.

A proxy receiving a CANCEL request must try and match it to an existing INVITE context (*ProxyCoreObj*) and cancel any pending client transactions associated with this INVITE, as illustrated in Figure 1-7 above. If an INVITE context is not found, the proxy must statelessly forward the CANCEL request (it may have statelessly forwarded the associated INVITE previously).

RECORD-ROUTING

The scenario in Figure 1-8 shows message proxying without Record-Routing.

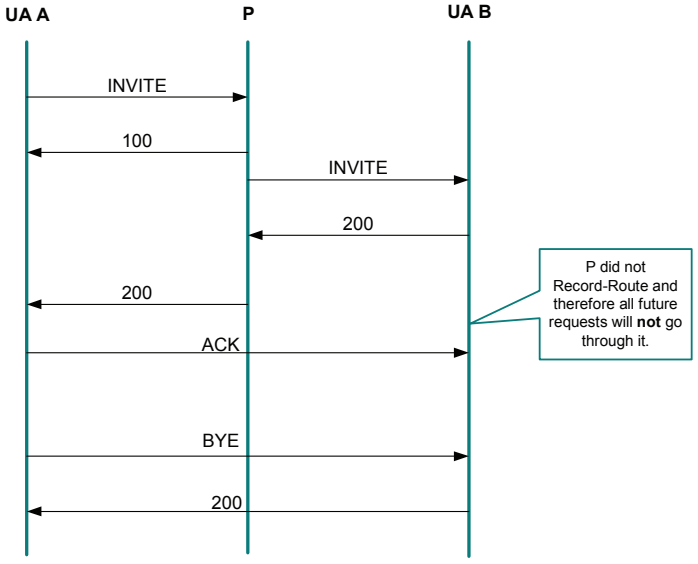


Figure 1-8 *Message Proxying without Record-Routing*

Record-Routing is a SIP mechanism that allows SIP proxies to request being in the signaling path of all future requests that belong to this dialog. A proxy Record-Routes by entering the Record-Route header into the initial dialog-establishing request (currently, INVITE and SUBSCRIBE). The UAS and UAC build route-lists based on the Record-Route headers they find in the request and send each subsequent request with the route list as a set of Route headers.

A proxy that does not Record-Route an INVITE message should not expect to receive any of the following requests sent as part of the *Stack Call-leg*, including the ACK request (as shown in Figure 1-8). Similarly, a proxy that does not Record-Route a SUBSCRIBE request should not expect to see any of the NOTIFY requests sent in the context of the subscription. Exceptions to these rules may arise from the use of outbound proxy or from loose-routing. For more information, see Loose-Routing and Outbound Proxy.

Proxies should normally Record-Route only requests that set up a dialog (currently INVITE and SUBSCRIBE). However, a proxy may add a Record-Route header to any SIP request if it so wishes. SIP UAs do not change their route-lists based on Record-Route headers in requests other than the initial INVITE or SUBSCRIBE. It is recommended that proxies avoid adding Record-Route headers to every request for reasons of processing time and message size.

Selective Record-Routing is highly important because it allows proxies to keep track of some dialogs for their entire duration, while assisting others only in their initial setup phase and then removing themselves from loop. This helps proxies avoid spending resources on routing requests for dialogs that are of no interest to them.

The Record-Routing mechanism was enhanced and fine-tuned numerous times during the evolution of SIP in order to cope with various difficult cases. Today a Record-Routing proxy is required to implement the following functionality:

- Route information pre-processing
- Route information post-processing
- Rewriting Record-Route headers in responses (For example, the case of a proxy that is routing between different domain names.)
- Symmetric Record-Route
- Loose-Routing

Symmetric Record-Route

In some cases, a request is received in the Proxy on network interface X and needs to be sent on network interface Y. In such cases, it is recommended that the proxy adds two Record-Route headers, one for each interface. As result of such behavior of the Proxy, each of the UAs builds the same Route-List (in the opposite order), and each of the UAs will send further requests of the session to the corresponding interface of the Proxy. This also means that the Proxy does not need to analyze the Record-Route in the response.

LOOSE-ROUTING

A Loose-Routing proxy (also known as Loose-Router) is one that follows the procedures defined for Record-Routing in RFC 3261. These procedures allow a proxy to introduce more hops into the route-list regardless of the final destination of the message. This enables a proxy to route a message through a pre-defined set of other proxies before reaching its final destination. Loose-Routing was added to the SIP specification due to requirements set by wireless 3G standard bodies, such as the 3GPP, in order to allow for "intelligent" routing of messages.

RECURSION ON 3XX RESPONSES

The scenario in Figure 1-9 shows recursion on 3xx Response

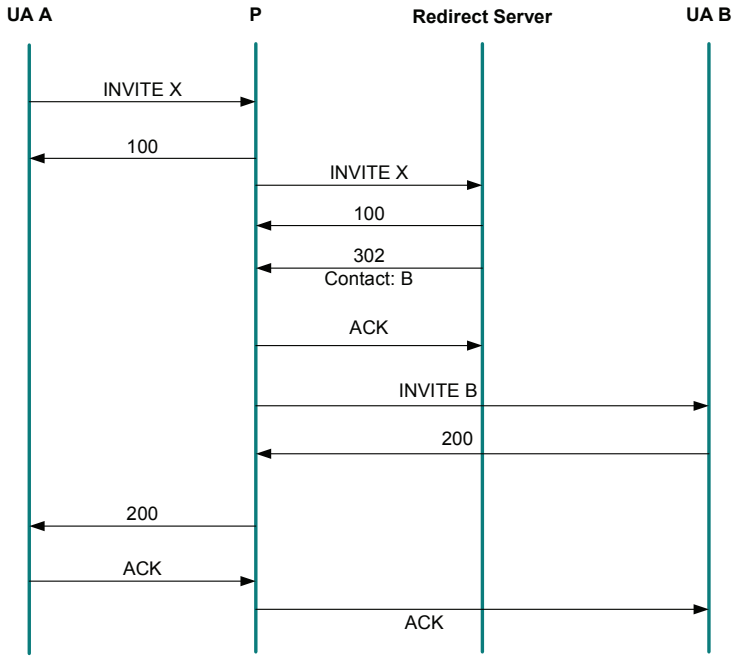


Figure 1-9 *Recursion on 3xx Response*

A proxy, upon receiving a 3xx (Redirection) response, may choose to add the contact address(es) provided in the 3xx response to the target set and possibly generate a new copy of the request to this address(es), as illustrated in Figure 1-9. This process is called recursion on 3xx responses. A proxy may do recursion on 3xx responses only if the Request-URI of the original request indicates an address for which the proxy is responsible.

FORKING

After processing an incoming request and building a target-set for it, the proxy may choose to forward the request to multiple addresses. This process is called forking and a proxy that supports it is called a forking proxy. Forking allows the implementation of features such as simultaneously searching (ringing) for a user in multiple UA devices (desktop phone and cellular phone, for example), finding the first available agent in a call-center, and Forward-on-Busy.

A proxy may choose to fork in several ways:

- Parallel forking—the proxy forwards copies of the request to multiple destinations simultaneously.
- Sequential forking—the proxy forwards the request to one target address at a time, waiting for a final response (failure) before moving on to the next address.
- Mixed forking—the proxy may choose to forward requests to some target addresses in parallel while doing sequential forwarding for others.

PARALLEL FORKING

The scenario in Figure 1-10 (below) shows parallel forking.

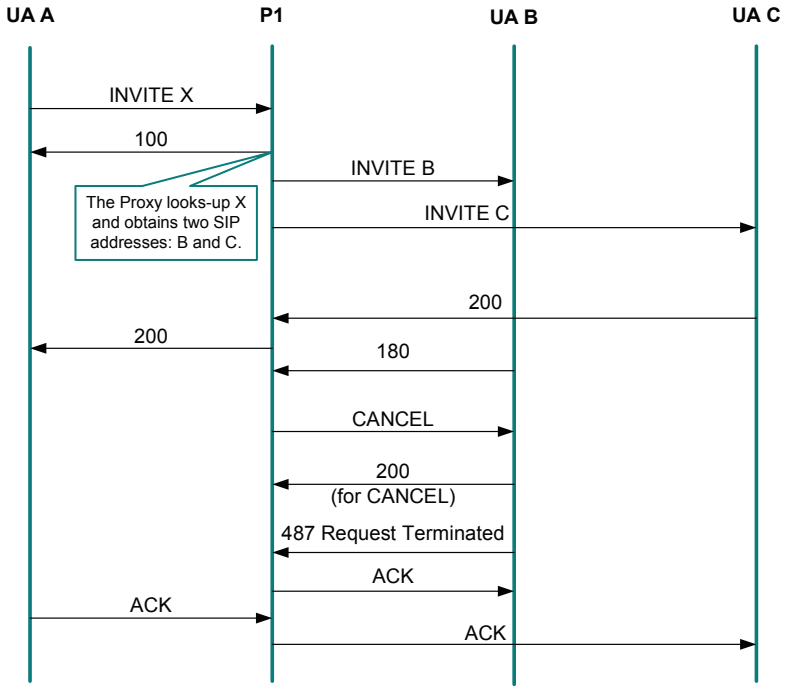


Figure 1-10 *Parallel Forking*

Parallel forking, as illustrated in Figure 1-10, is a more time-effective way to search for a user. This is because parallel forking attempts to reach the user in multiple locations simultaneously. However, parallel forking introduces additional complexity into the work of the proxy. A parallel forking proxy has to handle multiple concurrent client transactions and possibly collect multiple final responses from them. The proxy has to choose the "best" final response and forward it upstream. Picking the best response is done according to an algorithm provided in the SIP standard. Some responses have higher preference then others, and some responses should never be forwarded upstream. For example, a 200 (OK) response is "better" than a 404 (Not Found) response. Upon receiving a 2xx (Success) or 6xx (Global Failure) response at one of the client transactions, the proxy has to cancel the remainder of the pending requests (using CANCEL) and forward the final response upstream.

When parallel forking, a proxy also may have to do aggregation of challenges if it receives multiple 401 (Unauthorized) or 407 (Proxy Authentication Required) responses. In this case, the proxy must collect the challenges from all responses and forward them upstream.

A parallel forking proxy may have to do redirection response aggregation if multiple 3xx responses are received.

Sequential Forking

The scenario in Figure 1-11 shows sequential forking.

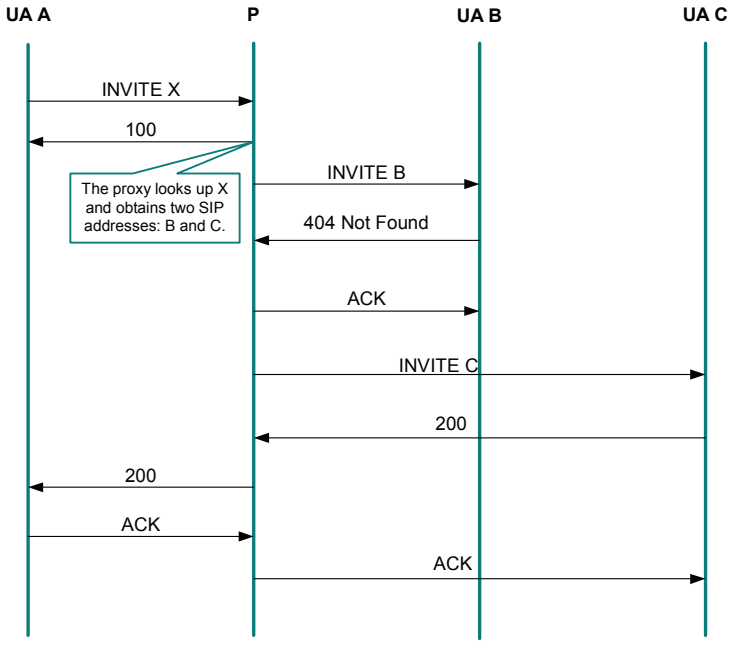


Figure 1-11 *Sequential Forking*

AUTHENTICATION

The scenario in Figure 1-12 shows authentication.

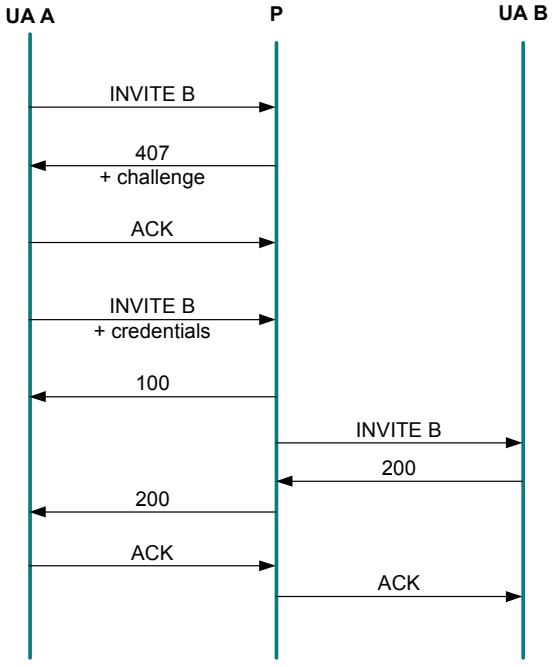


Figure 1-12 *Authentication*

When a UAC sends a request to a proxy server, the proxy server may decide to authenticate the originator before the request is processed. The proxy can challenge the originator to by returning a 407 response (Proxy Authentication Required) with a Proxy-Authenticate header containing the challenge. The client can re-send the request with a Proxy-Authorization header which provides the credentials that match the challenge. A client may provide the credentials also before being challenged in order to avoid the delay and extra processing of the 407 response (the credentials may be built according to cached challenges). Both challenge and credentials are built using a cryptographic hash so that certain values, such as password, are not sent in the clear.

Authentication by proxy is useful for the following:

- Verifying that the originator of the request is indeed an authorized user entitled to receive services (avoiding service theft)

- Asserting that certain message fields were not altered by a third party

SIP authentication also allows for multi-level authentication by different proxies along the signaling path.

Loop Detection and Max Forwards

Figure 1-13 shows an example of an illegal loop.

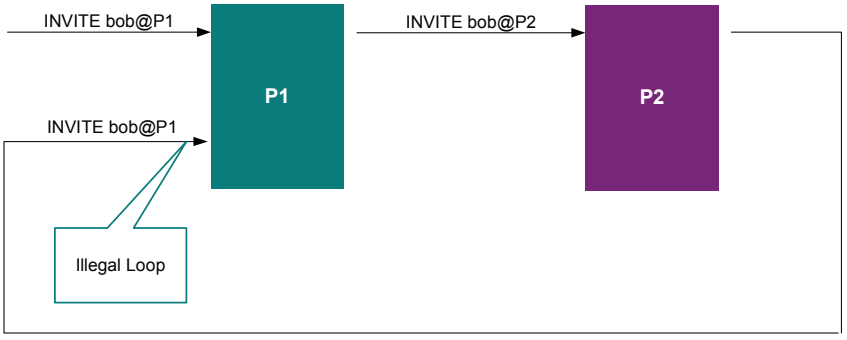


Figure 1-13 *Illegal Loop*

A loop is a situation where a request that arrives at a proxy is forwarded, and later arrives back at the same proxy. An illegal loop occurs if the second time it arrives, the message has the same values in fields that affect the routing decision (Request-URI and any other fields the proxy may take into account). In this case the proxy will forward the request the same way it did the first time and every other proxy along the loop path may do the same. This causes an error condition where the message is looped through a set of proxies for an undefined number of times, consuming network and proxy resources, but never reaching its final destination.

The SIP specification handles loops in two ways:

- Max Forwards (mandatory)
- Loop detection (optional)

Max Forwards

The Max-Forwards header field serves to limit the number of hops a request can transit on the way to its destination. It consists of an integer that is decremented by one at each hop. If the Max-Forwards value reaches 0 before the request reaches its destination, it will be rejected with a 483 (Too Many Hops) error response. The default initial value for Max-Forwards is 70.

This solution requires minimal processing by proxies and UACs, but has the disadvantage of stopping the loop only after the message has been forwarded enough times to exhaust the Max-Forwards value (70 hops by default).

Loop Detection

A proxy can optionally check for loops by employing a special loop detection algorithm. The algorithm affects the way the proxy builds the Via-branch field and mandates the proxy to do certain validations of the Via list in incoming requests. Loop detection requires some extra processing per message, but guarantees immediate detection of the loop as the proxy receives the message for the second time.

Message Spiraling

Figure 1-14 shows an example of message spiraling (Legal Loops).

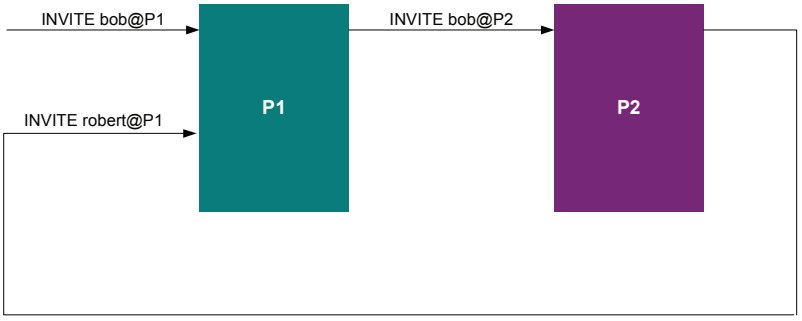


Figure 1-14 *Spiral Example*

A spiral is a SIP request that is routed to a proxy, forwarded onwards, and arrives once again at that proxy, but with a different set of values in the fields that affect the routing decision. The proxy will route the spiraled request to different target addresses the first and second time it processes it, therefore an endless loop is not created. A spiral is not an error condition, unlike a loop. Identifying spirals and telling them apart from loops presents a challenge for loop-detecting proxies and requires the use of special techniques.

Outbound Proxy

Outbound proxy is a proxy that receives requests from a client regardless of the destination of the messages (Request-URI) the client is sending. Simple clients may choose to send every outbound message via an outbound proxy. Typically, a UA is manually configured with an outbound proxy, or can learn about one through auto-configuration protocols.

Outbound proxies are important because they allow the creation of simple UAs that do not have to be concerned with making routing decisions and DNS queries. This is especially important in wireless devices which typically have limited capabilities and resources but may roam to foreign networks.

A proxy that is suitable for use as an outbound proxy is one that is willing to accept requests even though they are not intended for its domain and that do not belong to a dialog that it has Record-Routed.

POLICY

As explained above, SIP Servers have a wide array of possible ways to handle incoming messages. A SIP Server has the freedom to make the following decisions:

- Where to route the request and based on which location information source (location service/database/presence info/ other)
- Proxy/Redirect/Reject
- Stateful/Stateless forwarding
- Record-routing
- Forking—parallel/sequential/none
- Authentication
- Loop detection

These decisions can be made based on many factors, such as:

- Message destination address (Request-URI)
- Domain(s) managed by this server
- Type of request (method)
- Other message fields: From, To, Date, Priority …
- External parameters, such as time of day

In this document, the collective set of rules that govern proxy routing decision making is called Server Policy. Server Policies are typically set by the service providers or administrators that install and configure the SIP Servers.

Because of the large number of input variables and output decisions that are available for Server Policies, SIP Servers are highly versatile and flexible elements. This is of key importance for deployment of SIP-based networks. Server Policy can be as complex or as simple as required by network topology, user profiles, traffic load, security risk levels, and other factors specific to the environment in which the server operates.

Below are examples of some simple policies that may be employed by proxies:

- Forward all requests targeted to domain acme.com based on location service records. Redirect all other requests to sip:routing_proxy.sp.com
- Authenticate all incoming INVITE messages except those intended for 911 emergency services
- Statefully forward (and fork if necessary) requests with destination address sip:bob_smith@acme.com, unless they are from sip:alice_smith@worldcom.com, in which case forward to sip:bob_smith@voicemail.acme.com.
- Switch from stateful to stateless forwarding modes when system resources are low.

B2BAF

A Back-to-Back User Agent (B2BUA) is a server type that is defined by the SIP standard, RFC 3261. The B2BUA is a dialog logical entity that receives a request and processes it as a User Agent Server (UAS). To determine how the request should be answered, it acts as a User Agent Client (UAC) and generates a request. Unlike a Proxy server, the B2BUA maintains the dialog state and participates in all requests sent on the dialog that was established. The B2BUA can also intervene in the session establishment and change the session parameters according to its policy.

The SIP Server Platform Advanced Back-to-Back Application Framework (B2BAF) allows implementing multi-level Back-to-Back (B2B) applications using predefined building blocks to quickly create powerful custom applications. SIP developers can quickly and effectively add B2BUA capabilities to their application by integrating the SIP Server B2BAF Add-on library and Service Library package.

The RADVISION SIP Server Platform supplies the application with two logical layers, the B2BAF Framework layer and the B2BAF Service Library layer. The B2BAF Framework layer provides a predefined building block for development of SIP call-based services. It is written on top of the SIP Stack Call-Leg layer and exports an API for implementing customizable SIP services. The B2BAF Service Library uses the B2BAF Framework to supply a set of SIP call-based services, such as transparent connect, disconnect and modify, among others.

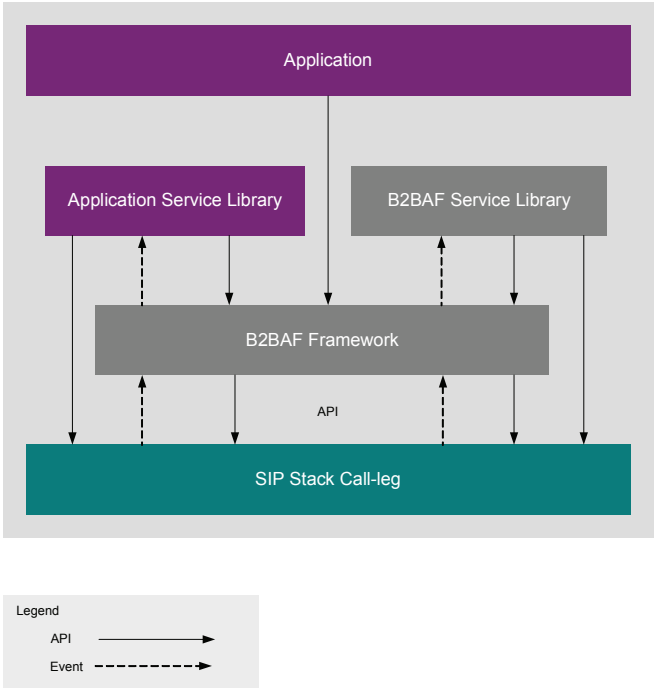


Figure 1-15 *B2BAF Add-on Layers*

EVENTS SERVER

An Events Server is a general implementation of specific event notification, as discussed in RFC 3265, and provides a complete implementation of a Notifier. This RFC defines a general, SIP-extensible framework that allows an entity to subscribe for notifications on the state change of other entities. The Notifier in this architecture is responsible for receiving the SIP SUBSCRIBE request, validating it, and creating a subscription object that also includes the Event package name and the event header ID parameter if one is present. Additionally, the Notifier is responsible for collecting the resource state and sending NOTIFY messages to the Subscribers. The Notifier may involve other entities (some not

in the scope of SIP) to collect resource state information. Validation of a SUBSCRIBE request should be an automatic function that does not require user input, since non-INVITE transactions are limited by Timer F with a duration of 64*T1 (default T1 is 500ms according to RFC 3261). The validation of a SUSCRIBE request may include the following:

Event Package Name

The Notifier may return a 489 (bad event) response if it does not understand the event package that was specified in the Event header value, or is not capable of processing this event package. The Notifier may also create a subscription and return 200 (OK) or 202 (accepted) according to its policy if it cannot immediately determine its ability to process the event package. (For example, when user input is required or other entities need to be involved in such processing).

Subscribe Duration

The Notifier may check that the Expires header value is not too short. If it is too short, it may return a 423 (interval too small) response according to the rules specified in RFC 3265. This response will contain a Min-Expires header to notify the subscriber about the minimum expected duration of the subscription.

Authentication

The Notifier may also choose to perform authentication and authorization according to the application policy. This is an action usually recommended, given the information that is provided in the NOTIFY body.

After the subscription was validated and a subscription object was created, the Notifier must send a NOTIFY message. The message may contain a body expressing the current state of the resources, according to the event package that was processed and according to the application policy. SIP uses this mechanism for the implementation of various functionalities and defines various event packages in related drafts and RFCs. Examples of such event packages are:

- Presence—provides information about the willingness and ability of a user to communicate with other users on the network.
- Winfo—the set of users subscribed to a particular resource for a specific event package, and the state of their subscriptions is called Watcher information (Winfo). A user can subscribe to be notified about the subscription state of such users. This package

is a template package and can be implemented with any event package of the *foo.winfo* template (where "*foo*" can be any SIP event package) including *presence.winfo*, or *winfo.winfo*.

For example, a Presentity that subscribes to its Presence Watcher information will receive notifications about Watchers subscribing to their Presence status and when their subscription status changes.

- Reg-event—a 3GPP event package used for receiving information about a terminal registration state.

Using this general framework of an Events Server, servers can be built to handle specific event packages, such as a Presence Server. In 3G networks, for example, event packages are considered to be the infrastructure of all advanced services.

PRESENCE SERVER

Presence is a service that allows a party to know the ability and willingness of the other party to participate in a call even before the call attempt is made. A user interested in receiving presence information (a Watcher) for another user (Presentity) can subscribe to his/her presence status and receive Presence status notifications from the Presence system.

The IETF SIPMPLE WG is developing a set of specifications for the implementation of a Presence system using SIP. They are working within a general IETF requirements framework for Presence and IM, which is called Common Presence and Instant Messaging (CPIM).

According to SIMPLE, the Presence system is composed of these components:

- One or more Presence User Agents (PUA)—these are the client devices/software the Presentity uses and that collects Presence data (for example, phone, cell-phone, PDA, soft PC client, and geo-location system).
- One or more Presence Agents (PA)—the PA is responsible for:
  - Receiving and handling Presence subscriptions from Watchers.
  - Receiving and aggregating Presence data from the PUAs (either through SIP or via other means).
  - Composing the Presence status from the fragments of presence data.
  - Notifying all subscribed Watchers when the Presence status of a Presentity in which they are interested changes.

- SIP Proxies—forward Subscription requests and notifications between the Watcher and Presence Agent (basically regular proxies that are not aware of Presence).
- Presence Server—a physical entity that processes SUBSCRIBE requests either as a presence agent (and handles them locally) or as a proxy (and forwards them to other entities).

SIMPLE defines a general type of server called Presence Server (PS) which can act either as a PA or a Proxy, as needed. Essentially this is a Proxy which also has PA capability.

Figure 1-16 illustrates a common Presence implementation, Presence entities and the relation between them in the network.

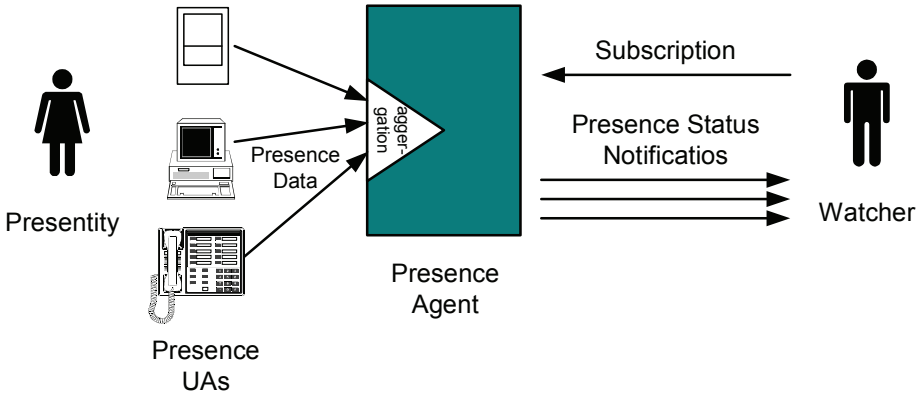


Figure 1-16 *Presence Topography*

Figure 1-17 illustrates a common scenario between Watchers, Presentities and the Presence Agent.

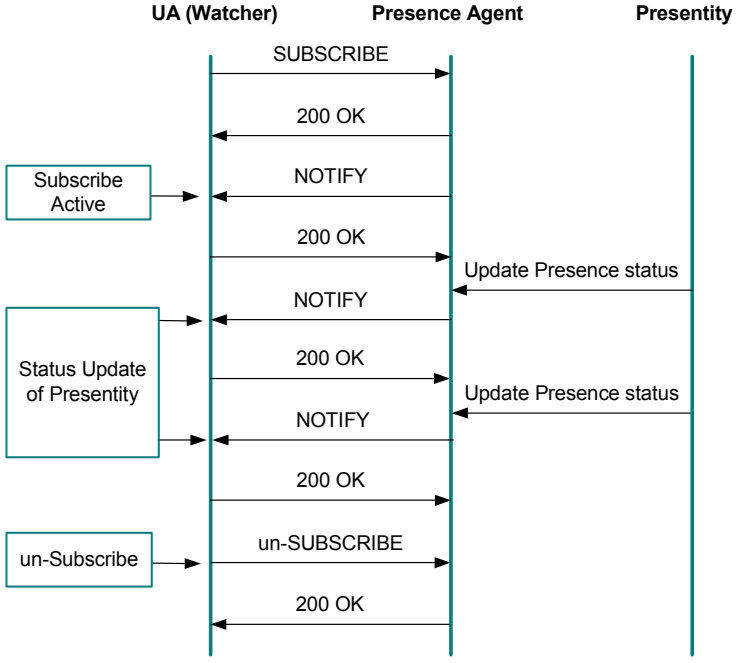


Figure 1-17 *Watchers, Presentities and Presence Agent*

Multiple Watchers can subscribe to receive updates of the status of the Presentity. Each subscription will result in a NOTIFY message generated by the Presence Agent to report an active subscription. The Presence Agent will generate the status information of the Presentity from different sources, some of which are SIP-based and some of which are not. Examples of SIP-based sources are PUBLISH messages and REGISTER messages sent by the Presentity. A change in the status of the Presentity will result in the generation of a NOTIFY message by the Presence Agent to all Watchers according to application policy.

Both a Subscriber (Watcher) and Presence Agent can terminate (Unsubscribe) a subscription. A Subscriber can send a SUBSCRIBE within the dialog with an Expires header of zero. (An Expires header indicates the duration of the subscription). This causes an immediate termination of the subscription. The

Presence Agent then generates a NOTIFY request with the most recent state. This, in fact, is a way of retrieving Presentity status without actually subscribing to the Presence Agent, and is referred to in the Standard as a Fetcher or Poller.

The Presence Agent can terminate the subscription by sending a NOTIFY request with a Subscription-State header, indicating that the subscription has been terminated.

SIP Servers Infrastructure Components

This sections describes the following:

- Database Access
- Accounting

Database Access

Many SIP Servers need realtime access to databases. The most common are the Location Database and the Security Database. Databases can be server-internal or server-external. When using an external database, database access protocols are used, such as LDAP and DIAMETER. SIP Servers should support this capability.

Accounting

There is a need for many SIP Server applications to collect accounting or Call Detail Record (CDR) information. The challenge in VoIP and SIP systems is that accounting information can include a wide set of data elements, as opposed to traditional systems, where the main information was call time and duration. On SIP systems, accounting information is expanded and includes the following accounting elements:

- Type of service—call, instant messaging, presence, and so on.
- Type of media—codec, bitrate.
- Mid-call events—media update, information messages, and so on.

Accounting standards are defined in 3GPP (TS.32.225).

2

RADVISION SIP SERVER PLATFORM

SIP SERVER DEVELOPMENT CHALLENGES

With the world-wide adoption of SIP, numerous vendors, research organizations and academic institutions are choosing to develop their own SIP servers. While no two SIP server projects are exactly alike, the set of requirements put forward by customers are often the same. RADVISION has identified the following as key technical requirements for a SIP server project:

- Fast Implementation
- Standards compliance
- Interoperability
- Robustness
- Extensibility
- Flexibility and customization
- High performance and capacity
- Maintainability
- Portability

Often SIP servers replace or supplement telephony switches and private branch exchange (PBX) systems so customer requirements are especially strict. Failing to fulfill some or all of these requirements presents a major risk to the commercial success of the product. For example, shipping a non-standard or non-interoperable SIP server may result in lost accounts or in having to do an overall recall or upgrade.

Naturally the regular project constraints also apply when developing SIP servers, namely:

- Minimizing development time

- Minimizing cost of development

Minimizing Development Time and Cost

The need for a fast and effective development process is clear. Often development time has a direct affect on time-to-market and the ability to meet customer deadlines. Development of a SIP Server application typically involves implementation of these layers:

- SIP Stack layer—has to have extra flexibility and customizability in the way transactions are processed because of the special needs of the proxy and the SIP Server in general.
- SIP Server layer—implements the standard SIP Server functionality.
- Application layer—implements all other aspects of the application (for example, service engine, billing module and database access).

Developing all three layers from scratch requires multiple man-years even for a basic type of SIP Server. Using an existing SIP Stack may save some time, but only if the SIP Stack was already enhanced to support proxy implementation. Testing and verifying standards compliance and interoperability are also resource consuming activities and often require purchasing third-party equipment for lab testing.

The RADVISION SIP Server Platform was proven to radically shorten development time and cost. The SIP Server Platform provides the following functionality with a minimal amount of extra coding:

- Complete Proxy, Redirect and Registrar servers
- B2BAF
- Events Server (for the implementation of Presence and other SIP events)
- Accounting Module
- LDAP Module

For example, developing a full-featured Proxy, Registrar or Redirect server using the SIP Server Platform requires a few man-weeks to complete. The SIP Server Platform allows developers to focus on the differentiating features of the application they are developing and thus reduce both the time and cost of development and improve the quality of the product.

RADVISION SIP Server Platform

The RADVISION SIP Server Platform is a comprehensive framework for the development of any type of SIP Server application. The SIP Server Platform includes an implementation of fully-standard server functionality that is

controllable through multi-level APIs. SIP Server developers can easily and seamlessly integrate the SIP Server Platform into their application and customize it according to project needs, thus adding SIP Server capabilities to their implementation quickly and effectively.

The SIP Server Platform was designed to specifically address the requirements listed above. This section explains how.

STANDARDS COMPLIANCE

Standards compliance is a key requirement for any communication element. Without it the chances of interoperating with other vendors' equipment are small. Assuring compliance with the SIP standards is complex for the following reasons:

- **Extensive standardization activity**

  The SIP related working groups (SIP, SIPPING, SIMPLE, BEHAVE, XCON and so on) within the IETF are extremely active. Each year they discuss many proposed changes and enhancements to the baseline standard and to its extensions. Researching and keeping up-to-date with this activity is time consuming and requires extra resources.

- **SIP still on the move**

  There are two main reasons for this continuous change:

  - SIP is being adapted to new environments such as 3rd-Generation (3GPP) wireless and cable TV networks
  - "Broken" protocol behavior is being fixed.

    Changes to fulfill these requirements are not always backwards compatible. For example, RFC 3515.

- **Complexity**

  The functionally defined in the SIP standard for Proxy server, B2BUA server, and Events server is extensive in scope and not trivial in implementation. A standard SIP server is anything but simple. Some of this functionality is optional, but the bulk of it is not. Building a SIP Server from scratch (even with a SIP Stack in place) is a task that requires careful design and many man-years in implementation.

RADVISION is a leading VoIP technology vendor and is very active in researching and shaping the protocols. The SIP Server Platform, as all other technology products, is kept closely in-tune with the standard, and product updates are regularly shipped with SIP standard updates.

INTEROPERABILITY

Interoperability is a top requirement, especially interoperability with market leaders equipment. However being standards-compliant alone does not guarantee interoperability, since different vendors may interpret the standard differently. Interoperability is mostly achieved through:

- **Implementation philosophy**—leniency when processing incoming messages and strictness when encoding outgoing messages.
- **Extensive interoperability testing**—one of the most important SIP interoperabilty venues is the SIP Interoperability Testing (SIPit) and SIMPLEt events that take place a number of times each year. Other events are led by IMTC. In-lab interoperabilty testing is also crucial.

The RADVISION SIP Server Platform is tested on a regular basis in SIPit and SIMPLEt events where interoperability is verified against dozens of vendors at a time, including market leaders. In addition the SIP Server is tested for interoperability in the RADVISION QA lab with equipment bought from select endpoint and server vendors.

ROBUSTNESS

SIP servers are often required to display the same high-level of robustness and fault tolerance required from high-end telecom and datacom equipment. SIP servers are expected to run continuously for long periods of time with no restart. They are expected not to crash even under extreme loads or error conditions. Robustness is typically achieved through correct design and exhaustive testing. A SIP server typically needs to undergo different cycles of testing such as load, stress, and error-condition testing in order to minimize the chances of crashes or faults at the customer site.

The RADVISION SIP Server Platform is a robust standard implementation. The robustness of the SIP Server Platform is verified in the following ways:

- **Extensive testing and QA**—each new version of the SIP Server Platform undergoes extensive testing and QA before being released to the market.
  - **Torture tests**—the SIP Server Platform is tested with the most advanced torture testing suits.
- **Based on the RADVISION SIP Stack**—the SIP Server Platform is built using the award winning and widely used RADVISION SIP Stack. The SIP Stack is sold separately as part of the RADVISION SIP Toolkit package, where it

undergoes independent testing. This dual testing (at Stack level and at SIP Server level) helps further assure robustness in the SIP Server Platform.

- **Interoperability testing**—when testing interoperability the robustness of the product is also tested since many crashes occur due to erroneous protocol implementation by other vendors. Also, in the SIPit and SIMPLEt events, the SIP Server Platform typically takes part in advanced server testing which further aids in detecting and fixing bugs.
- **Used by multiple implementors world wide**—the SIP Server Platform is sold to developers world wide, who typically do their own system testing in various environments. This helps to further assure that the SIP Server Platform is fully robust and stable in any condition.

EXTENSIBILITY

SIP is designed for extensibility—the protocol can be extended in many different ways. The IETF already defined multiple extensions to the baseline protocol and others are sure to come. The implications on SIP applications are that any type of SIP element, endpoint or server, needs to be designed with extensibility in mind. The direct implications on proxy and redirect servers are:

- The servers must be able to process unknown types of messages even without understanding the request type (method), response code, certain headers in the message, or the body of the message.
- The servers must be able to process and respond to the Proxy-Require header.
- Server design and implementation should allow for future extensions without requiring major code re-writes. This typically requires a modular multi-layer design and implementation.

The SIP Server Platform was conceived and implemented as a development tool for any type of SIP server. As a result, it can be extended in many ways including message processing, message structure, state changes, policy and authentication. Nearly every aspect of the behavior of the server is overridable and customizable by the application to allow for maximum extensibility.

Flexibility and Customization

SIP servers may be deployed in different environments which have varying constraints. The ability to write a server that can be tailored and configured to match the constraints of a network without code changes is a key advantage. In many cases network administrators will want to do some of the customization and fine-tuning on their own at the customer site.

The SIP Server Platform accommodates this requirement through its extensive multi-level API, its rich set of configuration parameters, and its open and modular design. Many aspects of server behavior are customizable, such as Advanced B2BUA services, LDAP schema, domain names, DNS usage, location database access, memory consumption, and automatic behavior when processing messages. The SIP Server Platform enables the application to decide on any aspect of message processing, such as routing decisions, forking, and stateless/stateful authentication. The SIP Server Platform features multiple levels of APIs and callbacks that enable the application developer to have fine control even over the most low-level behavior, such as the order of headers in an outgoing message.

Performance and Capacity

High performance and capacity are prime requirements for carrier-class and large enterprise types of equipment.

SIP server performance is typically measured in calls per second (CPS), transactions per second (TPS) or messages per second (MPS). Capacity is typically measured in maximum number of concurrent transactions (sometimes called “logical ports”).

The SIP Server Platform is optimized for both types of requirements. It yields extremely high CPS, TPS and MPS rates (benchmarking documents are available). This is due to the high performance capabilities of the underlying Stack as well as to the use of highly efficient ANSI C code in the message and transaction processing engines inside the SIP Server Platform.

The SIP Server Platform achieves high capacity through tight control over resource usage, especially memory consumption. The memory consumed by a transaction in progress is minimal, thus the number of concurrent transactions that a machine can sustain is high and can be further up scaled by installing more memory.

Maintainability

Maintenance is an important (and often costly) phase in the life cycle of any SIP server application. This is typically due to:

- Standard changes, which frequently still occur in SIP.

- Customer requirement changes—most SIP networks today are still “1st generation” and therefore the customers often do not have full advanced knowledge of their requirements.
- Defects and interoperability issues detected in the product.

Using the SIP Server Platform can improve the overall maintainability of the project and can reduce the time and resources spent on maintenance for the following reasons:

- RADVISION does its own maintenance of the SIP Server Platform including alignment with the latest standards and bug fixes. These versions are made available to customers under the maintenance agreement and free them from the need to update their code with SIP standard changes.
- The SIP Server Platform is provided entirely in source code for enhanced readability and has extensive and customizable logging capabilities for error tracking and debugging. This makes detecting and fixing bugs related to use of the SIP Server Platform easier and faster.
- The SIP Server Platform introduces a layered and object oriented design that improves modularity and maintainability.

Portability

In some cases, the platform on which the SIP server application runs may change over time. Being able to port your code to a new operating system or platform in a fast and straight-forward way is an advantage.

The RADVISION SIP Server Platform is built in a cross-platform way so that all its code is entirely operating system-agnostic except for an internal operating system abstraction layer. The SIP Server Platform was already ported to multiple operating systems (including embedded/RTOS) and is expected to be ported to others in the future. The API that the SIP Server Platform exposes to the application is the same regardless of the underlying operating system. Therefore, migrating to a new operating system does not necessitate any code changes.

SIP Server Platform Capabilities

The following sections present the capabilities of the RADVISION SIP Server Platform.

Standards compliance

The SIP Server Platform complies with many of the SIP IETF specifications. For example:

- **RFC 3261**—SIP: Session Initiation Protocol (baseline SIP spec)
- **RFC 3263**—locating SIP Servers
- **RFC 2778**—a model for Presence and IM
- **RFC 3856**—Presence Event Package
- **RFC 3857**—Winfo Event Template Package
- **RFC 3858**—Winfo format
- **RFC 3862**—CPIM
- **RFC 3863**—Presence format (PIDF)
- **RFC 4028**—Session-Timer
- Partial Presence complies with the following drafts:
  - **draft-ietf-simple-partial-pidf-format-06.txt**—Partial PIDF format
  - **draft-ietf-simple-xml-patch-ops-02.txt**—Patch-ops framework
  - **draft-ietf-simple-partial-notify-07.txt**—Partial Notify
  - **draft-ietf-simple-partial-publish-04.txt**—Partial Publish

General Capabilities

The general capabilities of the SIP Server Platform are as follows:

- TLS—a security protocol that 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.
- Persistent Connection—in many cases, a SIP Server can handle multiple messages, transactions, and dialogs with a single SIP Server or Gateway and therefore, a single TCP connection can be reused avoiding TCP handshake overhead (and even more so in TLS connections).
- Multi-homed Host—the SIP Server Platform can be configured to listen to multiple IP addresses and to send a message via a specific socket. This feature can be used to enable the support of heterogeneous networks, such as combinations of IPv4 and IPv6 and different network segments.

- Symmetric Record-Route—in some cases, a request is received in the Proxy on network interface X and needs to be sent on network interface Y. In such cases, it is recommended that the proxy adds two Record-Route headers, one for each interface. As result of such behavior of the Proxy, each of the UAs builds the same Route-List (in the opposite order), and each of the UAs will send further requests of the session to the corresponding interface of the Proxy. This also means that the Proxy does not need to analyze the Record-Route in the response.
- Internal Multithreading (Configurable)
- DNS—the SIP Server Platform supports both basic (A records) and advanced (SRV and NAPTR records) DNS queries as defined in RFC 3263 (Locating SIP Servers).
- IPv6 support—the SIP Server Platform can listen on IPv6 sockets, accept TCP connections, and send and receive IPv6 packets (UDP and TCP).[1]
- High availability—saving active calls and the state of the subscriptions.
- Authentication—the SIP Server Platform support digest-MD5 (HTTP) authentication as specified in the standard. This can be extended to support other kinds of authentication.
- Address resolution—the SIP Server Platform supports the full procedure for address resolution including:
  - Determining a target-set, both:
    - Predefined
    - Defined by proxy based on location service or any other means
  - DNS resolution

Proxy Capabilities

The SIP Server Platform proxy features complete SIP proxy functionality with the following capabilities:

- Stateful forwarding
- Stateless forwarding

1. This capability is not recommended for operating systems that do not have native support for IPv6. Also in some cases, using IPv6 may prohibit full use of advanced DNS.

- Forking—parallel/sequential/mixed
- Record-Routing
- Loose-Routing
- CANCEL processing and forwarding
- Recursion on 3xx responses
- Loop detection
- Max Forwards check
- Working as outbound proxy
- Message validation
- Authentication
- Out-of-band transactions

Registrar Capabilities

The SIP Server Platform supports all standard registrar functionality:

- Accept and Validate REGISTER messages
- Read location mappings from location service
- Apply registration logic
- Update location service
- Remove location mappings that have expired from the location service
- Authentication

Redirect Capabilities

The SIP Server Platform provides full redirect server functionality:

- Receiving and processing incoming responses
- Address resolution
- Returning 3xx response with one or more Contact addresses
- Authentication

Advanced B2BUA Application Framework

The Advanced Back-to-Back User Agent (B2BUA) is an optional add-on module that is sold separately. The B2B, as opposed to a Proxy, acts as a SIP endpoint for one or more SIP dialogs (call-legs). Each call-leg can be in a different state, and B2BUA services are based on these accumulative states. The RADVISION Advanced B2BUA is built in a modular fashion, with a separation between the association of call-legs that take place in the B2B service, each call-leg state, and the accumulative service (such as conferencing). The B2B application developer has the following options:

- Use existing implemented states

- Extend existing services
- Replace (override) existing services
- Design new services
- Request RADVISION to develop new services

With these options, B2BUA application developers can implement any desired SIP Service. In summery, the Advanced B2BUA enables full control over groups of associated SIP call-legs.

SDP Stack

The RADVISION SDP Stack is a stand-alone SDP Library that provides SDP message processing functionality. SDP is a tool for specifying media capabilities, not a communication protocol. The SDP Stack performs the following:

- Parsing and encoding of SDP messages or message parts, such as media descriptions and RTP maps
- Message creation, either from new or existing messages or message parts

Events Server

The Events Server is an optional add-on module which is sold separately. The Events Server is a general implementation of specific event notification, as discussed in RFC 3265. It provides a complete implementation of a Presence Agent and a Watcher information template applied on the Presence Event Package. This module provides a general SIP-extensible framework exposing APIs to the application that allow handling of any type of SIP event package (including user-defined packages) and building of a general purpose SIP Events Server. In addition to this general purpose framework, the add-on module provides the specific implementation of three SIP event packages—Presence Server, Winfo, and Presence Agent.

Presence Server

The Presence Server module allows a party to know the ability and willingness of other parties to participate in a call even before an attempt has been made. The Presence Server is responsible for handling Presence SUBSCRIBE requests with event package “presence” from Watchers, and enables the application to notify them about the Presence status of the Presentities. The Presence Server works in coordination with the Presence Agent, to receive updated Presentity status according to PUBLISH requests, and optionally REGISTER requests, that are monitored by the Presence Agent. The Presence Server API also enables sending Presence Notification messages with Presence documents for active Presence subscriptions. The Presence documents are sent as the body of the

Notification message, and include Presence information about the requested Presentity. The Presence Agent module is implemented above the Presence Server layer.

As part of the RADVISION IMS strategy in the SIP Server Platform v.3, partial presence Publications and Notifications were implemented. The partial Publications and Notifications include only differences of a "presentity" state as opposed to the previous full state Publications and Notifications. This new functionality is much more efficient from the network perspective.

WINFO MODULE

The Winfo module handles subscriptions of the winfo template. The Winfo module can handle an incoming SUBSCRIBE request with the winfo template and any event package. For example, the *presence.winfo* and *foo.winfo* Event headers are both with the winfo template, and can be handled by the Winfo module (*presence* and *foo* are the event packages). The Winfo module offers the application a high-level interface for handling incoming Winfo SUBSCRIBE requests and for initiating NOTIFY messages with Winfo documents. The Winfo module uses the XML Encoder library for creating Winfo documents according to RFC 3858. For more information, see the *XML Encoder* chapter located in the *SIP Events Server Programmer and Reference Guide*.

The Winfo module is implemented above the Events layer. The Winfo API allows the application to decide how to handle a specific Winfo SUBSCRIBE request according to its policy, to control NOTIFY requests sent to the Winfo subscribers, and to create Winfo documents.

NON-SIP INFRASTRUCTURE

The SIP Server Platform includes some non-SIP functionality in the form of Server Components.

Currently the following functionality is supported:

ACCOUNTING ADD-ON MODULE

The RADVISION Accounting Add-on Module is designed to collect accounting information from other SIP Server modules. The accounting module will log accounting records according to triggers from the other SIP Server modules, which are totally configurable and flexible. The set of possible triggers is in compliance with 3G-IMS definitions (TS.32.225). SIP Server Platform 3.0 provides the accounting module in two parts. The first part is a protocol agnostic accounting data collection unit. This unit collects accounting data and stores it as well-formatted data. The second part is a transparent logger that saves the data to a file or to a network connection. The second part could be easily replaced by a specific protocol logger, such as IMS-CDR formatted in ASN.1, Diameter, or others.

Access to External Databases

Using the SIP Server LDAP Add-on Module, other SIP Server add-on modules can access external databases. This module is designed mainly for Location and Security databases. However, developers can define any proprietary LDAP scheme to access external databases.

SIP Server Platform Architecture

Figure 2-1 illustrates the SIP Server Platform architecture.

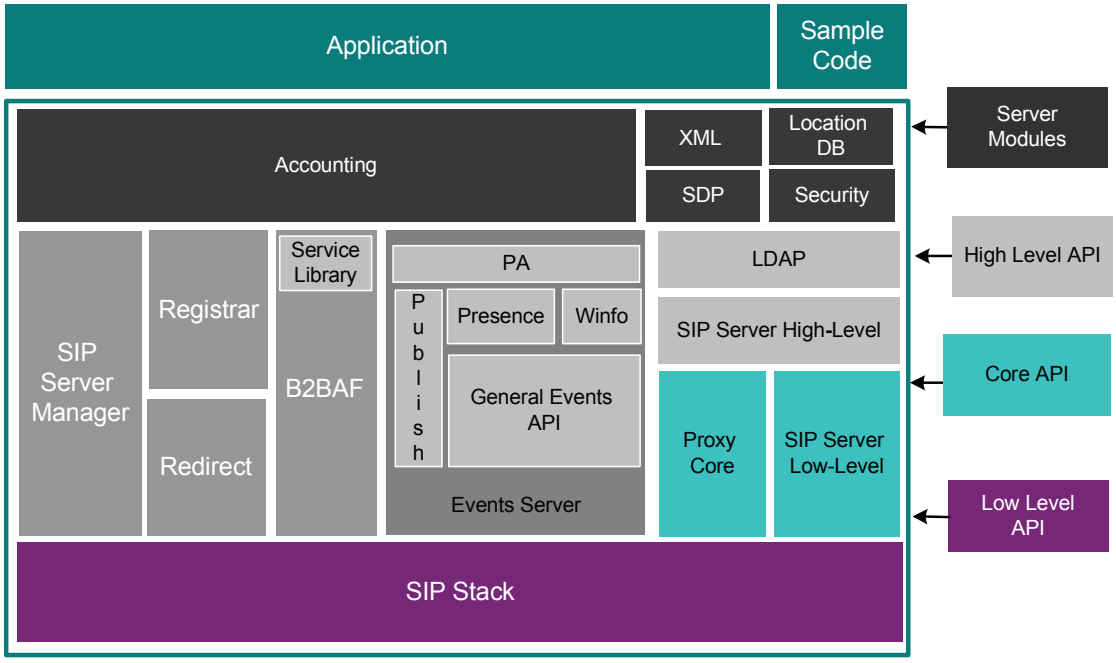


Figure 2-1 *Platform Architecture*

The SIP Server Platform has multiple internal layers. This allows for enhanced modularity and flexibility. Each layer exposes its API functions. Application developers can mix and match the level of API they choose to use according to application needs. For more information on these modules, see the relevant Programmer and Reference Guides.

The following API layers are exposed:

- **Component API**—server components also expose an API that enables the application to use them. This is not a SIP API.

- **High Level API**—the High Level API provides higher abstraction on top of the low level modules. This level contains such objects as Registration and allows configuring, initializing and shutting-down the entire SIP Server Platform.
- **Core API**—the Core API exposes the *ProxyCoreObj* (PCO), which is the low level processing object that matches server and client transactions and executes forwarding. The Core API is also used for working with state machines.
- **Low level API**—this API partially exposes the underlying Stack. This is mostly for purposes of working with messages (the messaging layer is located in the underlying Stack).

Note The underlying Stack cannot be used by the application for purposes other than SIP Server development and only as specified in product documentation.

SIP Server Platform Coding

The coding guidelines used by the SIP Server Platform are as follows:

- The SIP Server Platform is provided entirely in source code.
- The SIP Server Platform was designed and coded based on object-oriented (OO) methodology.
- For improved readability and maintainability, the SIP Server Platform was written using strict coding conventions covering naming, internal comments, code structure and project layout. The code is carefully documented.
- The SIP Server Platform is implemented in ANSI C.

SIP Server Platform Package

The SIP Server Platform package includes the following:

- SIP Server basic platform libraries in source code
- Server components or components in source code
- SIP Server add-on modules' libraries and source code (optional)
- Documentation in PDF, HTML and CHM format (For a list of all the SIP Server Platform documentation, see the About this Manual chapter.)
- Fully functional Test Application with graphical user interface given in source code

- Sample code—short, internally-documented, simple and compliable programs that demonstrate the usage of the APIs in source code

Operating System Support

Currently the following operating systems are supported:

- Linux Redhat
- Windows XP, 2003
- VxWorks
- Solaris
- Free BSD

The SIP Server Platform will be ported by RADVISION to additional operating systems in the future.

SIP Sever Application

The application layer implements all the application-specific behavior. This is where innovative services, value-added functionality and differentiated features are implemented while utilizing the powerful and versatile SIP capabilities of the underlying layers.

The SIP Server Platform together with its default Server Components can provide extensive out-of-the-box SIP server functionality. Implementing any of the following servers can be done with minimal coding:

- All-in-one SIP proxy, registrar and redirect server with simple routing policy and co-located location service
- Load balancer
- B2BUA
- Key systems
- Servers for causing calls to be anonymous
- Service delivering platforms
- Application SIP Servers
- Session border control
- IP-PBX
- Presence/IM servers
- Application Level Gateway (ALG)
- Billing servers
- SIP Security servers
- Fixed-to-Mobile Convergence servers (FMC)

- Soft switches

3

USING THE SIP SERVER

INTRODUCTION

This chapter provides basic and necessary information for getting started with the RADVISION SIP Server, including:

- Contents of the RADVISION SIP Server
- API conventions
- Memory management
- Logging
- Creating an application
- Threading model and working in multithreaded mode

SIP SERVER CONTENTS

The RADVISION SIP Server package contains the following:

- Source Code—the package contains the source code of the SIP Server, including the SIP Stack.
- Proxy Test Application—source code for the Proxy Test Application of the SIP Server. Demonstrates the functionality and flexibility of the Sip Server. For more information, see the SIP Server Test Application chapter.

- Sample Applications—Demonstrates the creation and implementation of specific functionality using the SIP Server. For more information, see the SIP Server Sample Applications chapter.

Note For a detailed description of the package structure and compilation directions, see the Readme file.

API Conventions

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

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

- Basic Data types
- Function Parameters
- API functions
- Return Status Codes

Data Types

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

- RvStatus
- RvUint64
- RvUint32
- RvUint16
- RvUint8
- RvUint
- RvInt32
- RvInt16
- RvInt8
- RvInt
- RvChar
- RvBool—the Boolean data type 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.

API Functions

API function names are according to the following convention:

RvSipServer + ObjectName + Functionality Description

or

RvProxy + ObjectName + Functionality Description (API for proxy functionality.)

Example 1

The API function for initialize the SIP Server instance is RvSipServerMgrConstruct().

Example 2

The API function for proxying a request is RvProxyCoreObjProxyRequest().

Return 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 type RvStatus.

For more information about the various status codes and their meaning, see the *Status Codes* chapter in the *RADVISION SIP Server Platform Reference Guide*.

Memory Management

The SIP Server allocates all required memory upon initialization. This means that you must predetermine the resources that the SIP Server will require during operation. These resources include the number of *ProxyCoreObj*, *RegServerObj*, *ProxyTransc* and *ProxyServerAuthObj* objects; the transport buffer size; the Memory Pool of pages for static memory usage; and more.

For a detailed description of Memory Pools and how to use them, see the Memory Pool chapter.

Logging

The SIP Server includes a Log module that produces textual output messages for the purpose of debugging, monitoring and tracking the activity of the SIP Server. The logger can be configured to work on different levels of detail. You

can select a global log level for the entire SIP Server. You can also assign different log levels to the subcomponents of the SIP Server, such as Proxy Core, Register Server, Messages and so on.

For more information about logging see the SIP Server Log chapter.

CREATING AN APPLICATION

This section describes the basic code necessary for writing an application using SIP Server, including:

- Initialization
- Event processing
- Configuration
- Logging
- Getting module handles
- Registering callbacks

INITIALIZATION

Before performing any SIP-related activity, you must initialize the SIP Server. The SIP Server Manager API function, RvSipServerMgrConstruct(), performs SIP Server initialization. Alternatively, the RvSipServerMgrConstructFromFile() function can be called to construct the SIP Server with a configuration specified in a configuration file. The file may be overwritten during server runtime when configuration changes.

RvSipServerMgrConstruct() receives the following parameters:

sizeofCfg

Input parameter. The size of the configuration structure.

RvSipServerCfg

Input/output parameter. Sets the parameters in the RvSipServerCfg structure with your configuration values. You can use RvSipServerInitCfg() to initialize the configuration structure with the SIP Server default values. The output parameter includes the values with which the SIP Server was initialized.

RvSipServerMgrHandle

Output parameter. You should supply this parameter when working with the SIP Server Manager API functions.

RvSipServerMgrConstructFromFile() receives the following parameters:

szCfgFilename

Input parameter. The XML file name contains the SIP Server configuration.

pEvHandler

Input parameter. A pointer to the SIP Server Manager event handler.

RvSipServerMgrHandle

Output parameter. You should supply this parameter when working with the SIP Server Manager API functions.

Construct and Destruct Functions

Both construct functions perform the following:

- Construct the subcomponents of the SIP Server, including construction of the underling SIP Stack.
- Allocate memory.
- Initialize timers, the log file, and the interface to the network.

The RvSipServerMgrDestruct() function destroys the SIP Server and frees all resources that the SIP Server uses.

To initialize the SIP Server from a configuration file

1. Declare and annihilates -RvSipServerMgrEvHandler with the application event handlers.
2. Initialize the SIP Server by calling RvSipServerMgrConstructFromFile() giving the name of the XML configuration file and application event handler structure as parameters.

To initialize the SIP Server giving a configuration structure

3. Declare the RvSipServerCfg structure and the RvSipServerMgrHandle.
4. Call RvSipServerInitCfg() to fill the configuration structure with the SIP Server default values.
5. Set your own configuration values in the configuration structure. The remainder of the SIP Server configuration will be adjusted according to the changes you have made.

6. Initialize the SIP Server by calling the RvSipServerMgrConstruct() function.

The SIP Server Manager (*SIPServerMgr*) replaces the values of the parameters initialized with RV_SIP_SERVER_DEFAULT in the RvSipServerCfg structure with the default values used to perform initialization.

Sample Code

The sample code in Figure 3-1 demonstrates how to initialize the SIP Server.

```
/*=============================================================*/
#define APP_LOCAL_INTERFACE = "172.23.5.80"
RvSipServerCfg       serverCfg;
RvSipServerMgrHandle hServerMgr;

RvStatus AppInitSipServer()
{
RvStatus rv;
RvSipServerMgrInitCfg(sizeof(serverCfg), &serverCfg));
serverCfg.maxProxyCoreObj          = 50;
serverCfg.maxRegisterNum           = 10;
serverCfg.stackTcpEnabled          = RV_TRUE;
serverCfg.sipServerDomainsList     = APP_LOCAL_INTERFACE;
serverCfg.sipServerDomainsListSize = strlen(APP_LOCAL_INTERFACE)+1;
serverCfg.localUdpAddressList      = APP_LOCAL_INTERFACE;
serverCfg.localUdpAddressListSize  = strlen(APP_LOCAL_INTERFACE)+1;
serverCfg.localTcpAddressList      = APP_LOCAL_INTERFACE;
serverCfg.localTcpAddressListSize  = strlen(APP_LOCAL_INTERFACE)+1;
rv = RvSipServerMgrConstruct(sizeof(serverCfg),
                             &serverCfg,
                             &hServerMgr);
if(rv != RV_OK)
{
printf("Failed to initialize the Sip Server");
}

  return rv;
}
/*=============================================================*/
```

Figure 3-1 *Initializing the SIP Server*

In the above sample code, the MaxProxyCoreObj and MaxRegisterNum settings limit the SIP Server to creating a maximum of 50 *ProxyCoreObj* objects and 10 *RegisterServerObj* objects simultaneously. The SIP Server configuration parameters will adjust to the new numbers you set.

CONFIGURATION

The SIP Server enables you to configure memory consumption, debugging filters of the log file, network parameters and timer parameters.

Configuration parameters are categorized as:

- Resource allocation
- Behavior and policy
- Log filters
- Network configuration
- Timer configuration

RESOURCE ALLOCATION

The SIP Server allocates all required memory upon initialization and the application must predetermine the resources that the SIP Server will require during the operation at initialization.

BEHAVIOR AND POLICY

The application can control the policy and behavior of the SIP Server using configuration parameters. The configuration can determine behavior such as whether the SIP Server will authenticate incoming requests, whether it will work with Record Route, and whether it will work in a transaction stateful or stateless mode.

LOG FILTERS

The application can control how the SIP Server produces log messages by setting filters either individually for each layer of the SIP Server or for all layers simultaneously. The application sets the logging filters by configuring the xxxLogFilters fields in RvSipServerCfg, with any combination of the filters listed in Table 3-1 below.

Table 3-1 *Log Filters*

Filter	Description
RVSIPSERVER_LOG_NO_LOG	No log will be outputed.
RVSIPSERVER_LOG_DEFAULT_FILTER	Use the log filers defined by for the SIP Server default (defaultSipServerLogFilters).
RVSIPSERVER_LOG_INFO_FILTER	A description of SIP Server activity.

Table 3-1 *Log Filters*

RVSIPSERVER_LOG_DEBUG_FILTER	Provides detailed log messages of SIP Server activity.
RVSIPSERVER_LOG_WARN_FILTER	A warning about a possible non-fatal error.
RVSIPSERVER_LOG_EXCEP_FILTER	A fatal error has occurred that prevents the SIP Server from continuing to operate.
RVSIPSERVER_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 Server.
RVSIPSERVER_LOG_LOCKDBG_FILTER	The details of all the locking operations of the SIP Server.
RVSIPSERVER_LOG_ENTER_FILTER	Indicates that an API function was called. This filter effects the logging of the Core and ADS modules only.
RVSIPSERVER_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 sample code in Figure 3-2 demonstrates how to instruct the Register Server module to print only INFO and ERROR messages to the Log, while all other modules work with default log filters.

```
/*================================================================*/
RvSipServerCfg serverCfg;
RvSipServergrInitCfg(sizeof(serverCfg), &serverCfg));
serverCfg. regServerLogFilters  =
RVSIPSERVER_LOG_INFO_FILTER
RVSIPSERVER_LOG_ERROR_FILTER;
/*================================================================*/
```

Figure 3-2 *Log Filter Sample Code*

Network Parameters

The SIP Server allows you to configure the following settings:

Local Listening IP Addresses

The SIP Server listening addresses. The application must configure at least one local address for each transport that was configured (UDP, TCP or TLS). The SIP Server enables you to listen to IPv6 local addresses. The application should configure the associated ports for the addresses. If the port is not defined, UDP and TCP will use 5060 and TLS will use 5061.

Record Route Configuration

If the application is a proxy that implements the Record-Route mechanism, the application can configure the Record-Route header that will be set for each interface of the SIP Server. For more information, see the SIP Server Configuration chapter.

Via Address configuration

the application can control the via address that is set in each outgoing request by setting a Via header for each interface of the SIP Server. For more information, see the SIP Server Configuration chapter.

Note Using local annotation addresses, such as “0.0.0.0” or [0:0:0:0:0:0:0:0] is prohibited.

Timer values

The SIP specification makes use of different timers to complete its transactions. You can configure the SIP Server 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 Server configuration, see the SIP Server Configuration chapter.

Event Processing

After initiating the SIP Server, you will need to instruct the SIP Server to process events as they occur. You do this by calling RvSipServerMgrProcessEvents().

Sample Code

The sample code in Figure 3-3 below demonstrates how to instruct the SIP Server to process incoming events.

```
/*-----------------------------------------------------------------*/
int main()
```

```
{
RvStatus rv;
/*Initializes the RADVISION SIP Server.*/
rv = AppInitSipServer();
if(rv != RV_OK)
{
printf("Failed to initialize the SIP Server\n");
return -1;
}
...
/* Commands the SIP Server to process the events queue.*/
RvSipServerMgrProcessEvents();
return 0;
}
/*-----------------------------------------------------------------*/
```

Figure 3-3 *Processing Incoming Events*

GETTING MODULE HANDLES

In some cases, in order to use a specific API function, you need the handle of the manager of that module. For example, in order to use the Register Server Manager API, you need a handle to the Register Server Manager (*RegServerMgr*). The Register Server Manager API provides functions that enable you to get the required module manager handle.

Sample Code

The sample code in Figure 3-4 below demonstrates how to get the *RegServerMgr* handle.

```
*-----------------------------------------------------------------*/
RvSipProxyRegServerMgrHandle AppGetCallLegMgr (
RvSipServerMgrHandle hServerMgr)
{
RvSipProxyRegServerMgrHandle hRegServerMgr;
RvSipServerMgrGetRegServerMgrHandle (hServerMgr,
                                     &hRegServerMgr);
Return hRegServerMgr;
}
/*-----------------------------------------------------------------*/
```

Figure 3-4 *Getting the Register Server Manager Handle*

REGISTERING CALLBACKS

The SIP Server interacts with the application using API functions that the application can use and events, in the form of callbacks, to which the application may listen and react. The events are grouped according to the different objects, and you can register to receive callbacks regarding a specified object using each object manager. For example, if you wish to receive callbacks regarding a *RegServerObj*, you should set your application events implementations using RvProxyRegServerMgrSetEvHandler().

The application can decide which events it would like to implement and sets these callbacks with the application implementation functions. All the other events should be set with NULL. Setting the event implementations of the application should be done upon initialization of the SIP Server and before starting to work.

Sample Code

The sample code in Figure 3-5 below demonstrates registering application events of a *RegServerObj*. In this case, the application implemented only three of the four events. (The application is not interested in being notified when updating the Location Database Server Component fails.)

```
/*=========================================================================*/
RvStatus AppSetRegisterEvents(
          RvSipServerMgrHandle hSipServerMgr)
{

RvProxyRegServerEvHandler AppRegServerEvHandler;
RvProxyRegServerMgrHandle hRegServerMgr;
RvStatus                  rv = RV_OK;


/* Retrieves the RegServerMgrHandle from the SipServerMgr
 object. */
….

AppRegServerEvHandler.pfnEvRegServerObjCreated = AppRegServerObjCreatedEv;
AppRegServerEvHandler.pfnEvRegServerObjRegMsgRcvd = AppRegServerObjReqRcvdEv;
AppRegServerEvHandler.pfnEvRegServerObjStateChanged = AppRegServerObjStateChangedEv;
AppRegServerEvHandler.pfnEvRegServerObjRespMsgToSend = AppRegServerObjRespMsgToSendEv;
AppRegServerEvHandler.pfnEvRegServerObjDBUpdateFailed = NULL;

rv = RvProxyRegServerMgrSetEvHandler(
                      hRegServerMgr,
```

```
                              &AppRegServerEvHandler,
                              sizeof(AppRegServerEvHandler));


      if(rv != RV_OK)
      {
      printf("Failed to set Register Server events\n");

      }

      return (rv);
      }
/*=========================================================================================*/
```

Figure 3-5 *Registering Application Events of a RegServerObj*

SIP Server Working Mode

The RADVISION SIP Server can work as Proxy, Stateless or Transaction Stateful, as B2BAF, and it can also handle Register requests as Registrar and handle SUBSCRIBE requests as Events Server.

Handling incoming requests by the RADVISION SIP Server is done according to the four available SIP Server working modes:

- **Stateless**—the SIP Server will handle any incoming requests as Stateless Proxy, forwarding requests and responses to their destination without any knowledge of the state machine of the transactions.
- **Transaction Stateful**—the SIP Server will handle any incoming requests as Transaction Stateful Proxy, forwarding requests to their destination, handling parallel and sequential forking, associating responses received to the outgoing requests and forwarding the appropriate responses to back stream.
- **Dialog**—the SIP Server will handle incoming requests as B2BAF, forwarding requests and responses with knowledge of the state of the dialogs.
- **Dynamic**—working in Dynamic mode lets the application decide how to handle a new incoming request that is not yet associated with a dialog, as Stateless Proxy, Transaction Stateful Proxy or B2BAF.

When configured in Transaction Stateful, Dialog or Dynamic working mode, the application can decide to handle incoming Register requests with a Request-URI of the local domain of the Server—as Registrar, and handle incoming SUBSCRIPTION and PUBLISH requests with a Request-URI of the local domain of the Server—as Events Server.

THREADING MODEL

The SIP Server can work in one of the following modes:

- **Threadsafe mode**—the SIP Server is not multithreaded but the application is. Therefore the SIP Server protects its objects using locks. In this mode, the StackNumberOfProcessingThreads configuration parameter is set to zero.
- **Multithreaded mode**—the SIP Server works with one main thread and several processing threads. All SIP Server objects are protected by locks. In this mode, the StackNumberOfProcessingThreads configuration parameter is set to number greater than zero.

WORKING IN MULTITHREADED MODE

To work in a multithreaded mode, the application should set the StackNumberOfProcessingThreads configuration parameter to a value bigger than zero.When working in a 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 following events are inserted into 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 asynchronic manner using the processing queue.
- Object states—states that the SIP Server needs to handle in an asynchronic manner that are inserted into the processing queue. For example, the connection object states and the MsgSendFailure state of the transaction object.

Mandatory Steps for Multithreaded Applications

Thread-safe and Multithreaded applications must follow these guidelines:

- The SIP Server is constructed from the main thread. The application must also destruct the SIP Server from the main thread.
- RvSipServerProcessEvents() 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 must be prepared to handle events (callbacks) that are received from different threads, including application threads.

For more information about multithreaded configuration parameters, see Multithreading Configuration Parameters in the SIP Server Configuration chapter.

4

WORKING WITH THE PROXY CORE

INTRODUCTION

The Proxy Core layer is responsible for routing SIP requests to UA servers and SIP responses to user agent clients (AUC). The Proxy Core makes a routing decision and modifies the request before forwarding it to the next hop. When a request is received, the Proxy Core layer decides if it needs to respond to the request on its own. For example, the request may be malformed or may need credentials from the client before forwarding it.

The Proxy Core layer can operate in either transaction stateful or stateless mode for each new received request. In stateless mode, the Proxy Core simply forwards a message to the resolved address based on the message information. Once a message has been forwarded, all the information regarding the message is discarded.

When acting as a transaction stateful, the Proxy Core stores information (such as transaction state and messages) about each incoming and outgoing request. It uses this information in order to process future messages associated with the request. In this mode, the Proxy Core layer can choose to fork the request to multiple destinations, and it processes the received responses for each forwarded request.

The Proxy Core API of the SIP Server Platform contains a set of functions and function callbacks. Using the Proxy Core API, the application can determine its SIP Server policy by setting the Proxy Core Object (*ProxyCoreObj*) attribute. These attributes effect the processing of requests and responses, such as sending a 100 “trying” on received INVITE request, or recursing 3xx responses. The application can also control a *ProxyCoreObj* and its Proxy Core Transaction

(*ProxyCoreTransc*) or Proxy Transmitter (*ProxyTrx*) objects. The application can cancel a request or one of its pending branches, forward the request to one or more destinations, and redirect or reject it.

Proxy Core Entities

The Proxy Core API relates to the following three entities:

- Proxy Core Policy Manager (*ProxyPolicyMgr*)
- Proxy Core Object (*ProxyCoreObj*)
- Proxy Core Transaction (*ProxyCoreTransc)*
- Proxy Transmitter (*ProxyTrx*)

Proxy Policy Manager

The *ProxyPolicyMgr* is responsible for managing the *ProxyCoreObj*, *ProxyCoreTransc* and *ProxyTrx* objects. The Proxy Policy Manager API functions enable the application to set the event handler for the *ProxyCoreObj* and for the *ProxyCoreTransc* that runs in transaction stateful mode for the *ProxyTrx* that runs in stateless mode. The Proxy Policy Manager API functions also enable the application to initiate a message sending by providing API for creating a *ProxyCoreObj*.

Proxy Core Object

The *ProxyCoreObj* is responsible for handling incoming and outgoing requests and responses. This object is created when requests or responses (in stateless mode only) are received. The *ProxyCoreObj* contains zero or one proxy server transaction and zero or more *ProxyCoreTransc* objects, which are created when the request is forwarded. In stateless mode the *ProxyCoreObj* also contains one *ProxyTrx* which is created when the message is forwarded.

The Proxy Core Objects API functions enable the application to handle received requests and responses according to its policy, to forward requests (sequential or parallel), to determine which response will be forwarded, and to access the Proxy Core Object parameters and change them. A *ProxyCoreObj* is initialized with the attributes from configuration and the application can change these attribute per *ProxyCoreObj*, using the API functions.

Proxy Core Transaction

A *ProxyCoreTransc* is an object at the SIP Server level that associates with a single SIP Stack transaction. A *ProxyCoreTransc* object may be of server or client type. A server *ProxyCoreTransc* represents an incoming request. A client *ProxyCoreTransc* represents an outgoing request for a specified destination.

The application can access and control a client *ProxyCoreTransc* using the Proxy Core Transaction API functions when working in stateful mode.

Proxy Transmitter

A *ProxyTrx* is an object at the SIP Server level that associates with a single SIP Stack transmitter. A *ProxyTrx* represents an outgoing request or response when working in stateless mode. The application can create a stateless *ProxyCoreObj* and *ProxyTrx*, and initiate message forwarding using the Proxy Transmitter API functions. When working in stateless mode the application can access and control a *ProxyTrx* using the Proxy Transmitter API functions.

If the application wants to be notified about the new *ProxyTrx* behavior it should register to its callbacks. If the application registers to RvProxyCoreTrxStateChangeEv() the former callback RvProxyCoreObjSLRequestFinalDestResolvedEv() will not be called (even if it is implemented). The application can continue to get its former callback by not implementing RvProxyCoreTrxStateChangeEv().

Request in Transaction Stateful Mode

Figure 4-1 illustrates the objects created by the SIP Server when handling an incoming request while working in transaction stateful mode.

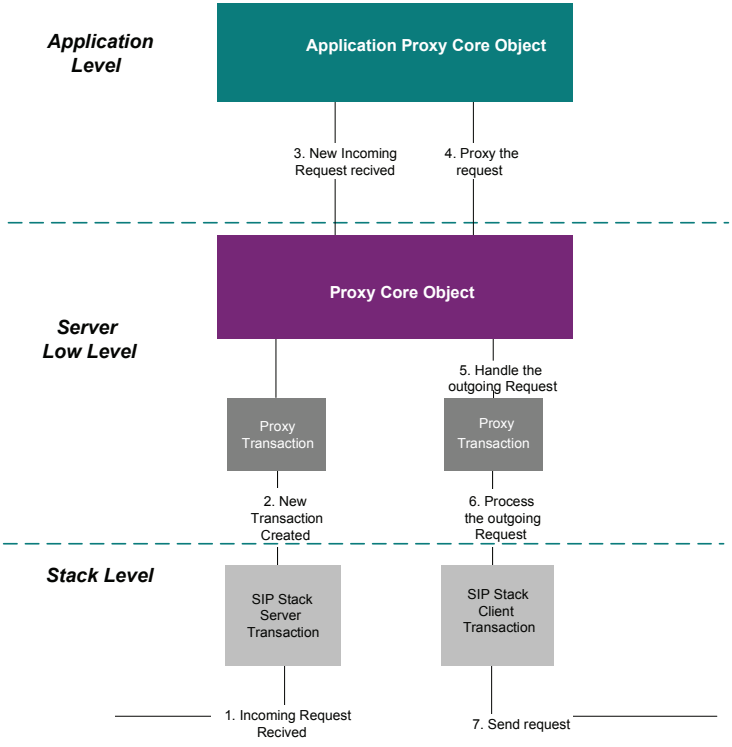


Figure 4-1 *Incoming Request in Transaction Stateful Mode*

The steps for handling the incoming request are as follows:

1. An incoming Request message is received from the network.

   The message is not associated with any existing SIP Stack transactions, therefore a new SIP Transaction object is created to handle the incoming request.
2. A new *ProxyCoreObj* with a Proxy Server Transaction is created to handle the new incoming request.
3. The application can open its own object associated with the *ProxyCoreObj* to handle the new incoming request and receive callbacks associated with this request, with its object handle.
4. After completing the request validation process and resolving the destination addresses for the request, the application chooses to proxy the request.
5. A new Proxy Client Transaction is created to handle the outgoing request.
6. A corresponding SIP Stack client transaction is created to handle the sending of the request.
7. The SIP Stack client transaction sends the request to the network.

Working with Handles

All Proxy Core objects (*ProxyCoreObj*, *ProxyCoreTransc*, *ProxyTrx* and *ProxyPolicyMgr*) are identified using handles. The application must supply these handles when using the Proxy Core API. The application can supply its own handle to be attached to these objects. The associated application object will be supplied on each callback.

The Proxy Core object handles are listed below:

- **RvProxyPolicyMgrHandle**—defines a handle for the *ProxyPolicyMgr*. The application receives this handle using RvSipServerMgrGetPolicyMgrHandle()
- **RvProxyCoreObjHandle**—defines a handle for the *ProxyCoreObj*. The application receives this handle on Rv*ProxyCoreObj*ectCreatedEv() or RvProxyCoreObjectInternalCreatedEv().
- **RvProxyTranscHandle**—defines a handle for the *ProxyCoreTransc*. The application receives this handle on RvProxyCoreTranscCreatedEv().

- **RvProxyTrxHandle**—defines a handle for the *ProxyTrx*. This handle is needed in all Proxy Transmitter API functions. The application receives this handle on RvProxyCoreTrxCreatedEv().

Proxy Core Policy Manager API

The Proxy Core Policy Manager API contains a set of functions that allow the application to create a new *ProxyCoreObj* and initiate message sending by supplying a message handle or message identifiers. The Proxy Core Policy Manager API also contains a set of functions that allow the application to set and examine Proxy Policy Manager parameters.

Proxy Core Policy Manager Parameters

The application can access Proxy Core Policy Manager parameters via Set and Get functions. The following parameters are available:

DNS Servers

In the process of address resolution, host names are resolved to IP addresses by sending DNS queries to the SIP Server DNS servers. If one server fails the next DNS server on the list is queried. When the SIP Server is constructed, a DNS servers list is set to the SIP Server using the computer configuration. The Proxy Core Policy Manager keeps the list. The application can provide a new set of DNS servers to the Proxy Core Policy Manager. It can also get the current DNS servers list.

DNS Domains

The SIP Server 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 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.

When the SIP Server initializes, the DNS domain list is set according to the configuration of the computer and is managed by the Proxy Core Policy Manager. The application can provide a new set of DNS domains to the SIP Server. It can also get the current list of DNS domains.

Proxy Core Policy Manager Control

The following API functions provide Proxy Policy Manager control:

RvProxyCorePolicyMgrCreateSFCoreObj()

Creates a new stateful *ProxyCoreObj* with a client *ProxyCoreTransc* and associates the application object to the new object. The application handles that are given as input parameters will be supplied at all callbacks relevant to this object. This function may be used to initiate a request by the Proxy Core layer.

Note: Initiating request is not a standard SIP proxy behavior.

RvProxyCorePolicyMgrCreateSLCoreObj()

Creates a new stateless *ProxyCoreObj* with a *ProxyTrx* and associates the application object to the new object. The application handles that are given as input parameters will be supplied at all callbacks relevant to this object. This function may be used to initiate a request by the Proxy Core layer.

Note: Initiating request is not a standard SIP proxy behavior.

Proxy Core Object API

The Proxy Core Object API contains a set of functions and function callbacks that allow the application to set or get the parameters of the *ProxyCoreObj*, *ProxyCoreTransc* and *ProxyTrx* objects.

Proxy Core Object Parameters

The following Proxy Core Object parameters are available:

Behavior Parameters

These parameters define the *ProxyCoreObj* behavior while handling requests and responses. These attributes indicate how to treat received requests and responses. When a *ProxyCoreObj* is created it is initialized with the values of parameters from configuration. If the application wishes to overwrite this behavior on a specific *ProxyCoreObj*, it can call the parameter API function in the initial states, RVPROXY_CORE_OBJ_STATE_IDLE or RVPROXY_CORE_OBJ_STATE_REQUEST_RECEIVED.

The behavior parameters are listed below:

Auto Try

Indicates whether the *ProxyCoreObj* should automatically send a 100 response to an incoming INVITE request.

Authentication Require

Indicates whether the *ProxyCoreObj* should authenticate the incoming request (except for CANCEL and ACK).

Loop Detection

Indicates whether the *ProxyCoreObj* should perform loop detection.

Recurse3xx

Indicates whether to recurse a redirection response.

Record Route mode

Indicates whether to insert Record Route headers in requests; whether to insert Record Route headers in requests and rewrite them when handling responses; or not to add Record Route headers at all. This parameter can be modified for a Transaction Stateful *ProxyCoreObj*.

Add Auth Header To Response

Indicates whether the *ProxyCoreObj* will add WWW-Authentication or Proxy-Authentication headers when the application initiated a 401 or 407 response (respectively). This parameter can be accessed using Set functions.

Mode Parameters

Represents the *ProxyCoreObj* working mode. The three mode types relevant to the *ProxyCoreObj* are as follows:

Stateless

The *ProxyCoreObj* runs in stateless mode.

Once the mode is set in the configuration, the application cannot change the mode per incoming request.

Transaction Stateful

The *ProxyCoreObj* runs in transaction stateful mode.

Once the mode is set in the configuration, the application cannot change the mode per incoming request.

Dynamic

This mode is a configuration mode only. When you configure the SIP Server to work in dynamic mode, it informs the application on receipt of a new request using the RvProxyCoreObjNewRequestRcvdEv() callback function. With this

callback, the application needs to decide whether to handle this request as transaction stateful or stateless, and to return its decision to the SIP Server. This parameter is not modifiable.

Other Proxy Core Object Parameters

State

Represents the state of the *ProxyCoreObj* in its state machine. This parameter is not modifiable.

Received Request

Represents the received request message. The application can only access this parameters using a Get function, which will return a handle to this message. This parameter is not modifiable.

Note The returned message handle could be NULL if no request has been received or if the message has already been released, for example, after receiving a final response.

Invalid Reason

When the SIP Server receives a request message, it performs a validation process according to the rules defined in the RFC. If the received request is invalid, the *ProxyCoreObj* assumes the RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST state with a reason that reflects the invalidation reason. If at any time the application would like to know the reason, the application can access this parameter with a Get function. This parameter is not modifiable.

Destination Address List

On receipt of a request message, the *ProxyCoreObj* tries to determine the request target as defined in the RFC. A set of target request addresses is saved in the Destination Address List. The addresses in the Destination Address List are

used to forward the request. The application can access this list with Get and Set functions and update the list using the SIP Server List API functions. Changing the Destination Address List should be done before forwarding the request.

Note The SIP Server List object is not protected for a multithreaded environment at the SIP Server level. It is the responsibility of the application to lock this object to ensure multithreading safety.

Route List

Provides the application with an easy way to add Route headers to any outgoing requests from the specified *ProxyCoreObj* according the server policy as defined in the RFC. All Route headers should be of loose routers.

Cancel Pair

When the SIP Server receives a CANCEL request, a new *ProxyCoreObj* is created. A "Cancel Pair" consists of two elements:

- The *ProxyCoreObj* of the CANCEL request.
- The *ProxyCoreObj* to be cancelled.

This parameter is not modifiable.

Session-Timer Preference

The application can set the Session-Timer preference parameters associated with this *ProxyCoreObj*. The application can set these parameters in transaction stateful mode only in a pre-forwarding state.

Authentication Header

The application can indicate whether or not the *ProxyCoreObj* should add a WWW-Authentication header or Proxy-Authentication header to an initiated 401 or 407 response message (respectively).

Proxy Core Object Control

The following API functions provide *ProxyCoreObj* control:

RvProxyCoreObjResolveAddr()

Starts the address resolution process in order to determine the next request hop. The application must use this function only in the RVPROXY_CORE_OBJ_STATE_REQUEST_VALID state.

RvProxyCoreObjProxyRequest()

Use this function in the RVPROXY_CORE_OBJ_STATE_ADDRESS_RESOLVED state to forward the incoming request in sequential or parallel forking to the destination addresses according to the Destination Address List.

RvProxyCoreObjRedirectRequest()

Use this function in the RVPROXY_CORE_OBJ_STATE_REQUEST_VALID or RVPROXY_CORE_OBJ_STATE_ADDRESS_RESOLVED state to send a 3xx redirection response. The application needs to supply a contact list to add to this redirection response. The application can use this function only with 300 to 399 response codes, in transaction stateful mode only.

RvProxyCoreObjRejectRequest()

Sends a 4xx-6xx-reject response. The application can use this function in the following states:

- RVPROXY_CORE_OBJ_STATE_REQUEST_RECEIVED
- RVPROXY_CORE_OBJ_STATE_REQUEST_VALID
- RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST
- RVPROXY_CORE_OBJ_STATE_AUTHENTICATION_FAILED
- RVPROXY_CORE_OBJ_STATE_ADDRESS_RESOLVED
- RVPROXY_CORE_OBJ_STATE_ADDRESS_NOT_FOUND

The application can also use this function with 0 as a response code which will generate and send an appropriate response to this invalid request. For example, when calling Reject() with 0 on illegal Max-Forwards, the *ProxyCoreObj* generates and send 483 response message.

RvProxyCoreObjTerminate()

Causes an immediate termination of the *ProxyCoreObj*. A stateful *ProxyCoreObj* is a self-terminated object, and its termination depends on its state. By using this function, the application can terminate a *ProxyCoreObj* before its normal termination. After calling this function, the *ProxyCoreObj* will assume the RVPROXY_CORE_OBJ_STATE_TERMINATE state and the application cannot use this object any more. A stateless *ProxyCoreObj* is terminated when its associated *ProxyTrx* is terminated. It is the responsibility of the application to terminate the *ProxyTrx* when it reaches this state.

RvProxyCoreObjCancel()

Sends a CANCEL request to all *ProxyCoreObj*-forwarded branches. The application can use this function only in transaction stateful mode, in the RVPROXY_CORE_OBJ_STATE_SEQ_PROXYING_REQUEST or RVPROXY_CORE_OBJ_STATE_PARALLEL_PROXYING_REQUEST states and on any request type. When calling this function, a *CancelCoreObj* will be created and a CANCEL request will be sent on any of the pending *ProxyCoreTransc* objects. The application is informed of the new *CancelCoreObj* using RvProxyCoreObjectInternalCreatedEv() callback function.

Note The *ProxyCoreObj* state will be changed to REQUEST_CANCELLED only if at least one transaction was actually cancelled.

RvProxyCoreObjetSendProvisionalResponse()

Use this function in the RVPROXY_CORE_OBJ_STATE_REQUEST_RECEIVED state to send a provisional response.

RvProxyCoreObjDisregardInvalidRequest()

Use this function in the RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST or RVPROXY_CORE_OBJ_STATE_AUTHENTICATION_FAILED state to ignore this invalidation and to continue with the request processing. The application can use this function for any Invalid reason except for Loop Detected.

Proxy Core Object Events

The Proxy Core Object API supplies several events, in the form of callback functions, to which the application may listen and react. In order to listen to an event, the application should pass the event handler pointer to the *ProxyPolicyMgr*. When an event occurs, the *ProxyCoreObj* calls the event handler function using the pointer.

The following events are used in all *ProxyCoreObj* modes.

RvProxyCoreObjectCreatedEv()

Notifies the application that a new *ProxyCoreObj* has been created. The newly created *ProxyCoreObj* always assumes the RVPROXY_CORE_OBJ_STATE_IDLE state. The application can create its own object to associate with this *ProxyCoreObj*. The handle of the application object will be supplied in any callbacks function related to this *ProxyCoreObj*.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjStateChangeEv()

This event is probably the most useful of the events that the *ProxyCoreObj* reports. Through this function, the application receives notifications of *ProxyCoreObj* state changes and the associated state change reason, and it can act upon the new state. For example, upon receipt of an RVPROXY_CORE_OBJ_STATE_AUTHENTICATION_FAILED state notification, the application can respond with a 401/407 response code.

RvProxyCoreObjMsgToSendEv()

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

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjMsgReceivedEv()

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

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyPolicyMgrNewRequestRcvdEv()

Called when working in dynamic mode and a new request is received. At this event, the application must decide whether the *ProxyCoreObj* that will be opened as a reaction to this new request should work in stateless mode or in transaction stateful mode.

RvProxyCoreObjInvalidResponseEv()

Called when the *ProxyCoreObj* received an Invalid Response. The message will be discarded and the proxy will continue according to the *ProxyCoreObj* mode. In stateless mode, the *ProxyCoreObj* will be terminated. In transaction stateful mode, the Proxy Core will decide whether to terminate the *ProxyCoreObj* or keep waiting for other responses, according to the *ProxyCoreObj* state and the number of active proxy transactions.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjProcessResponseEv()

Called when a response is received on an outgoing request. This callback notifies of the SIP Server's recommended action for the received response, such as forwarding the response or storing the response in the response context (in stateful mode only). The application can choose another action, and if this action does not contradict the RFC or the internal logic, this action will be performed for this response. Otherwise the SIP Server continues with its recommended action. The decision as to the recommended action for the received response depends on the response type and the *ProxyCoreObj* state.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjInitiateResponseEv()

Called when a response is initiated by the SIP Server. The application can prevent the Proxy Core from sending the response if it chooses to overwrite the recommended action with the RVPROXY_RESPONSE_DISCARD action in the requiredAction argument.

The Proxy Core initiates a final response when no final response was received for the *ProxyCoreObj*. For example, a 408 response will be initiated by the SIP Server if all transactions are terminated and no final response was yet received.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjSupplyRecordRouteForReqEv()

Allows the application to supply Record Route header parameters for the specified *ProxyCoreObj*. The Record Route header supplied will be used for the outgoing requests associated with the *ProxyCoreObj*. Therefore, these parameters must be one of the domains of the proxy so that the next request of this session will be sent to the proxy.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjRewriteRecordRouteInResp()

Allows the application to rewrite the Record Route header, which was inserted previously by this proxy, in a response message.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjRequestFinalDestResolvedEv()

Notifies the application that the SIP Server “knows” the explicit IP address, port and transport that is going to send the specified stateful request message. Before this callback is called, the SIP Server sets the local interface it will use to send the request. Choosing the address will be based on the first match according to the address type, transport and port. The application can call RvProxyCoreTranscGetResolvedFinalDest() on this event to retrieve this information. Using this information, the application can base its decision on which local interface will be used to forward this request.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjSLRequestFinalDestResolvedEv()

Notifies the application that the SIP Server "knows" the explicit IP address port and transport that is going to send the specified stateless request message. Before this callback is called, the SIP Server sets the local interface it will use to send the request. Choosing the address will be based on the first match according to address type, transport and port. The application retrieves the final destination of the request (address, transport and port) on this event. Using this information, the application can base its decision on which local interface will be used to forward this request and to set the address, transport and port values in the OUT parameters.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjOtherURLRcvdEv()

Consults the application about an incoming address whose URL type is not supported by the SIP Server. This callback is called whenever a *ProxyCoreObj* receives a request with an unsupported URL in the Req-Uri. The application should use this callback as follows:

1. Determine whether or not the Req-Uri indicates one of the proxy domains. If TRUE, the proxy will search the Location Database to determine the request target.
2. Rewrite the incoming address in a canonical form that will be used in hash keys and further comparisons.

Remark: Rewriting the incoming address is allowed only when the address was identified as the local domain of the proxy.

Note It is not permitted to call the Terminate() API function from this callback.

Additional Events When Working in Transaction Stateful Mode

The following events are used by the Proxy Core Object API when working in transaction stateful mode.

RvProxyCoreTranscCreatedEv()

Notifies the application that a new client *ProxyCoreTransc* object has been created. The application can create its own object to associate with this *ProxyCoreTransc* object. The handle of the application object will be supplied in any callback functions related to this object.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreTranscStateChangeEv()

Notifies the application that a client *ProxyCoreTransc* object state was changed. The application can act upon the new state. For example, upon receipt of the RVPROXY_CORE_TRNSC_TERMINATED state notification, the application can terminate its associated proxy transaction.

Currently, the only state that the *ProxyCoreTransc* object assumes after the IDLE state is the TERMINATED state.

RvProxyCoreObjectInternalCreatedEv()

Notifies the application that a new *ProxyCoreObj* was created. The newly created *ProxyCoreObj* always assumes the RVPROXY_CORE_OBJ_STATE_IDLE state. This callback is called only for *ProxyCoreObj* objects that are created automatically by the Proxy Core. For example, a *ProxyCoreObj* is created internally when the application has called RvProxyCoreObjCancel() or RvProxyCoreTranscCancel().

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreObjBestRespToForwardEv()

After a request was forward with parallel forking, if no response was forwarded to the user agent client (UAC), and all client transactions have been terminated, the SIP Server should choose one of the received responses, called “best response,” and forward it to the UA server as a final response.

This callback notifies the application of the response that was chosen by the SIP Server as best response. The application can replace the message and set another message handle to send as best response. If the application set another message

as best response, the SIP Server copies this message and uses it as the best response to forward. If the message was replaced, no additional processing will be done for this response.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreTranscOtherURLToSendEv()

Notifies the application that a message needs to be sent (in stateful mode), and the destination address is a URL type that the proxy does not currently support. In order for the message to be sent, the application must convert this URL into a SIP URL.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreTranscNewConnInUseEv()

Indicates that the Transaction is now using a new connection. The connection can be completely new or a suitable connection that was found in the hash.

Note It is not permitted to call the Terminate() API function from this callback.

Proxy Core Transaction API

The Proxy Core Transaction API contains a set of functions and function callbacks that allow the application to set or get the parameters of the *ProxyCoreTransc* objects.

Proxy Core Transaction Parameters

The following Proxy Core Transaction parameters are available:

ProxyCoreObj

Represents the associated *ProxyCoreObj*. This parameter is not modifiable.

Cancel Pair

When the SIP Server sends a CANCEL request on one of the *ProxyCoreTransc* objects, a new *ProxyCoreObj* object is created. This newly created *ProxyCoreObj* is the Cancel Pair of the *ProxyCoreTransc* that initiated the CANCEL request. The application can only access the state parameter with a Get function. This parameter is not modifiable.

DNS List

When the SIP Server works with DNS, sending a request may result in a list of address to which the Request URI is resolved. The DNS list is valid after the sending of the request to the first address in the list has failed and the RVPROXY_CORE_TRANSC_STATE_MSG_SENT_FAILURE state was reported.

The Application can get the list, examine the elements in the list, and add, remove or change the order of the elements in the list using Transport DNS List API functions, such as RvSipTransportDNSListPopSrvElement() and RvSipTransportDNSListGetSrvElement().

Received Response

Represents the last received response message. The application can only access this parameter using a Get function, which will return a handle to this message. This parameter is not modifiable.

Note The returned message handle can be NULL if no response has been received yet.

Invalid Reason

When the SIP Server receives a response message, it performs a validation process according to the rules defined in the RFC. If the received response is Invalid, the application will receive notification using the RvProxyCoreObjInvalidResponseEv() callback. This Response will be discarded and the reason for that invalidation will be stored in Invalid Reason parameter. If on any time the application would like to know that reason, it can access this parameter with a Get function. This parameter is not modifiable.

Persistency

The application can instruct the *ProxyTransc* on whether or not to try and use persistent connection. The application can change the *ProxyTransc* persistency only at the RvProxyCoreTranscCreatedEv() callback (before the request is sent).

Connection

The application can set a connection to be used by the *ProxyTransc*. The *ProxyTransc* will try to use the given connection for outgoing requests only. Responses are sent on the same connection as the request. If the connection will

not fit the *ProxyTransc* local and remote addresses it will be replaced. The application can set the transaction connection only at the RvProxyCoreTranscCreatedEv() callback (before the request is sent).

Local Address

The application can set the local address from which the *ProxyCoreTransc* will send outgoing requests. The SIP Server can be configured to listen to many local addresses. Each local address has a transport type (UDP, TCP or TLS) and an address type (IPv4 or IPv6). When the SIP Server sends an outgoing request, the local address (from where the request is sent) is chosen according to the characteristics of the remote address. Both the local and remote addresses must have the same characteristics, such as the same transport type and address type. If several local addresses are configured with same characteristics, the SIP Server will use the first configured match.

The application can force the *ProxyCoreTransc* to choose a specific local address for a specific transport and address type. For example, it can force the *ProxyCoreTransc* to use the second configured UDP/IPv4 local address. If the *ProxyCoreTransc* sends a request to a UDP/IPv4 remote address, it will use the local address that you set instead of the default first local address.

Current Local Address

The local address the *ProxyCoreTransc* will use to send outgoing requests. The *ProxyCoreTransc* holds six types of local addresses:

- UDP address on IP4
- UDP address on IP6
- TCP address on IP4
- TCP address on IP6
- TLS address on IP4
- TLS address on IP6.

The value of this parameter is the actual address from the six addresses that were used or are going to be used.

From Header

The From header of the *ProxyCoreTransc*.

To Header

The To header of the *ProxyCoreTransc*.

Call Id

The Call-Id value of the *ProxyCoreTransc*.

CSeq Step Number

The CSeq-Step number of the *ProxyCoreTransc*.

Via Branch

The branch that will be added to the top Via header of outgoing requests.

Outbound Message

The application can get the message that the *ProxyCoreTransc* is going to send. It is only valid in states where the *ProxyCoreTransc* is about to send a request message, for example, RVPROXY_CORE_TRANSC_STATE_IDLE. The application can use the message to add headers to the message before it is sent.

Proxy Core Transaction Control

The following API functions provide control over *ProxyCoreTransc* objects.

RvProxyCoreTranscCancel()

Sends a CANCEL request to a forwarded branch. The application can use this function only in transaction stateful mode on Sequential Proxying or Parallel Proxying and on any request type. When calling this function, a Cancel *ProxyCoreObj* is created and a CANCEL request is sent. The application is informed of the new Cancel *ProxyCoreObj* by the RvProxyCoreObjectInternalCreatedEv() callback function.

RvProxyCoreTranscDnsContinue()

Continues to send the request to the next address in the DNS address list. This function may be called after the application received notification of a sending failure in the RVPROXY_CORE_TRANSC_STATE_MSG_SENT_FAILURE state, and wishes to continue to send the request to the next address in the DNS list.

Before calling this function, the application may manipulate the DNS addresses list. The request will be sent to the first address in the list.

RvProxyCoreTranscDnsTerminate()

The application should use this function if it does not want to continue to the next address in the DNS address list. This function terminates the current *ProxyCoreTransc*. If the *ProxyCoreObj* state is RVPROXY_CORE_OBJ_STATE_SEQ_PROXYING_REQUEST, the SIP Server will continue to send the request to the next address in the destAddress list.

RvProxyCoreTranscTerminate()

Causes an immediate shutdown of the *ProxyCoreTransc* object. A Proxy Core Transaction is a self-terminated object. The termination of *ProxyCoreTransc* depends on SIP Stack transaction timers, and the termination of its associated *ProxyCoreObj*. By using this function, the application can terminate a *ProxyCoreTransc* object before its normal termination.

RvProxyCoreTranscMake()

Initializes a *ProxyCoreTransc* and sends a request. Before calling this function the application should call RvProxyCorePolicyMgrCreateSFCoreObj() to create a new *ProxyCoreObj* and *ProxyCoreTransc*.

RvProxyCoreTranscRequest()

Creates and sends a request message according to the given method, with the given address as a Request-URI.

RvProxyCoreTranscRequestMsg()

Sends a request message to the remote party using a given prepared message object. This function will normally be used when the message to be sent is already prepared.

RvProxyCoreTranscSendToFirstRoute()

Indicates whether or not to send the request according to first Route header (as a loose router) or according to the Request-URI.

Proxy Core Transmitter API

The Proxy Core Transmitter API contains a set of functions and function callbacks that allow the application to set or get the parameters of the *ProxyTrx* objects.

Proxy Core Transmitter Parameters

The following Proxy Core Transmitter parameters are available:

DNS List

When the SIP Server works with DNS, sending a request may result in a list of addresses to which the Request-URI is resolved. The DNS list is valid after sending the request to the first address in the list failed and the RVPROXY_CORE_TRX_STATE_MSG_SEND_FAILURE state was reported. The application can get the list, examine the elements in the list, and add, remove or change the order of the elements in the list using Transport DNS List API functions, such as RvSipTransportDNSListPopSrvElement() and RvSipTransportDNSListGetSrvElement().

ProxyCoreObj

Represents the associated *ProxyCoreObj*. This parameter is not modifiable.

Current State

Represents the state of the *ProxyTrx* in its state machine. This parameter is not modifiable.

Local Address

The local address from which the *ProxyTrx* will send outgoing requests. The SIP Server can be configured to listen to many local addresses. Each local address has a transport type (UDP, TCP or TLS) and an address type (IPv4 or IPv6). When the SIP Server sends an outgoing request, the local address (from where the request is sent) is chosen according to the characteristics of the remote address. Both the local and remote addresses must have the same characteristics, such as the same transport type and address type. If several local addresses are configured, the SIP Server will use the first configured match.

Current Local Address

The local address that the *ProxyTrx* will use to send the next message. This is the actual address from the six addresses that were used or are going to be used.

Outbound Details

A request sent by the *ProxyTrx* will use the outbound detail specifications as a remote address. The Request-URI will be ignored. However, the outbound proxy will be ignored if the message contains a Route header or if the RvProxyCoreTrxSetIgnoreOutboundProxyFlag() function was called.

Destination Address List

The destination address is the final address (IP, port and transport) that the *ProxyTrx* will use for sending the message. The destination address is calculated in the address resolution process, which takes into account the existence of the outbound address, Route headers and other Proxy Transmitter parameters.

The application can get the destination address only in the FINAL_DEST_RESOLVED state of the *ProxyTrx* which 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 *ProxyTrx* will not perform address resolution.

Via Branch

The application can set the top Via branch of the *ProxyTrx*. The *ProxyTrx* will add the branch to the top Via header of outgoing requests. Setting the Via branch should be done before forwarding the request.

Persistency

When working with TCP or TLS, the application can instruct the *ProxyTrx* to try to locate a suitable open connection in the connection hash before constructing a new connection.

Connection

The application can supply the *ProxyTrx* with a connection object that the *ProxyTrx* may use. The *ProxyTrx* will hold this connection in its internal database. When sending a message, the *ProxyTrx* will use the connection only if it fits the local and remote addresses. Otherwise, the *ProxyTrx* will either locate a suitable connection in the connection hash or create a new connection.

Keep Message Flag

Indicates that the *ProxyTrx* should not destruct the message immediately after encoding is completed. Before the *ProxyTrx* sends a message, it first encodes the message object to a buffer and then sends the buffer to the remote party. After encoding is completed, the *ProxyTrx* destructs the message object. In case of send failure, the *ProxyTrx* moves to the MSG_SEND_FAILURE state. In this state, the application can re-send the message object to the next address in the DNS list using RvProxyCoreTrxSendMsg(). The application can instruct the *ProxyTrx* not to destruct the message after encoding by setting the KeepMsg

Flag to RV_TRUE and, in this case, it can supply NULL as a message handle to the RvProxyCoreTrxSendMsg() function. The message will be destructed only upon termination of the *ProxyTrx*.

Ignore Outbound Proxy Details Flag

Instructs the *ProxyTrx* to ignore its outbound proxy when sending a request. When the message includes one or more Route headers, the *ProxyTrx* 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 *ProxyTrx* 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. In this case, the message object does not include Route headers but the *ProxyTrx* still needs to ignore its outbound proxy.

Outbound Details

A request sent by the *ProxyTrx* will use the outbound detail specifications as a remote address. The Request-URI will be ignored. However, the outbound proxy will be ignored if the message contains a Route header or if the RvProxyCoreTrxSetIgnoreOutboundProxyFlag() function was called. But if RvProxyCoreTrxSetForceOutboundAddrFlag() was called, the request will be sent to the outbound proxy regardless of the message content.

Proxy Core Transmitter Control

The following API functions provide control of *ProxyTrx* objects.

RvProxyCoreTrxTerminate()

Causes an immediate shutdown of the *ProxyTrx*. The *ProxyTrx* is not a self-terminated object and it is the responsibility of the application to terminate it. This function can be called in any state of the *ProxyTrx*. Naturally it is recommended to terminate the object in the RVPROXY_CORE_TRX_STATE_MSG_SENT or RVPROXY_CORE_TRX_STATE_MSG_SEND_FAILURE state.

After calling this function, the *ProxyTrx* object will assume the RVPROXY_CORE_TRX_STATE_TERMINATED state and the application can no longer use this object. Note that terminating a *ProxyTrx* object terminates its associated *ProxyCoreObj* as well.

RvProxyCoreTrxSendMsg()

Sends a message to a remote party using a given, prepared message object. The application will normally use this function when the message to be sent is already prepared. Call this function after creating a stateless *ProxyCoreObj*. When working with enhanced DNS feature, the application can call this function in the RVPROXY_CORE_TRX_STATE_MSG_SEND_FAILURE state for re-sending a request (except for ACK and CANCEL). Note that re-sending a request results in switching to stateful mode (RFC 3263).

Proxy Core Transmitter Events

The Proxy Core Object API uses the following events when working in stateless mode.

RvProxyCoreTrxCreatedEv()

Notifies the application that a new *ProxyTrx* object has been created. The newly created *ProxyTrx* always assumes the RVPROXY_CORE_TRX_STATE_IDLE state. The application can create its own object to associate with this *ProxyTrx*. The handle of the application will be supplied in any callback functions related to this object.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyCoreTrxStateChangeEv()

Notifies the application that a *ProxyTrx* object state was changed. The application can act upon the new state. For example, upon receipt of the RVPROXY_CORE_TRX_STATE_TERMINATED state notification, the application can terminate its associated *ProxyTrx*.

RvProxyCoreTrxOtherURLToSendEv()

Notifies the application that a message needs to be sent (in stateless mode), and the destination address is a URL type that the proxy does not currently support. In order for the message to be sent, the application must convert this URL into a SIP URL.

Note It is not permitted to call the Terminate() API function from this callback.

Proxy Core Object State Machine

The Proxy Core Object state machine represents the progress of the state of a *ProxyCoreObj* from the initial state when a request is received until the forwarding of the request and handling of the response, when working in transaction stateful mode. The RvProxyCoreObjStateChangedEv() callback reports *ProxyCoreObj* state changes and state change reasons. The state change reason indicates how the *ProxyCoreObj* reached the new state.

The state machine is divided into the following parts:

- Handling Incoming request—*ProxyCoreObj* progress from object creation through request validation, authentication and address resolution, regardless of whether the object is working in stateless or transaction stateful mode.
- Transaction Stateful Proxying—*ProxyCoreObj* progress from address resolution to forwarding request and processing responses.
- Stateless Proxying—*ProxyCoreObj* progress from address resolution to forwarding request and responses in stateless mode.

The *ProxyCoreObj* is associated with the following states:

RVPROXY_CORE_OBJ_STATE_IDLE

The initial state of the Proxy Core Object state machine. Upon creation of the *ProxyCoreObj*, the *ProxyCoreObj* assumes the RVPROXY_CORE_OBJ_STATE_IDLE state. It remains in this state until a Request or Response has been processed.

RVPROXY_CORE_OBJ_STATE_REQUEST_RECEIVED

Upon a receipt of request (not retransmission on transaction stateful mode) the *ProxyCoreObj* assumes the RVPROXY_CORE_OBJ_STATE_REQUEST_RECEIVED state. In this state the request validity will be checked. The application may send a provisional response or reject the request by using the API functions.

RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST

If the received request is Invalid the *ProxyCoreObj* will assume the RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST state. In this state, the application may disregard the invalidation reason and continue with the

request processing using RvProxyCoreObjDisrigardInvalidRequest(). Or the application may reject the request with the response code related to the state change reason.

RVPROXY_CORE_OBJ_STATE_AUTHENTICATING

After the request validation has been completed and found valid, the *ProxyCoreObj* will assume the RVPROXY_CORE_OBJ_STATE_AUTHENTICATING state while authenticating the request (only if Authentication Require was set to RV_TRUE).

RVPROXY_CORE_OBJ_STATE_REQUEST_VALID

After the request validation has been completed and found valid, the *ProxyCoreObj* assumes the RVPROXY_CORE_OBJ_STATE_REQUEST_VALID state. In this state, the application needs to call one of the following functions to proceed:

- RvProxyCoreObjResolveAddr()—proceeds to RVPROXY_CORE_OBJ_STATE_RESOLVING_DESTINATION state
- RvProxyCoreObjAcceptRequest()/ RvProxyCoreObjRejectRequest()/ RvProxyCoreObjRedirectRequest()—proceeds to one of the Final Response Sent states, according to the response class and request type.

RVPROXY_CORE_OBJ_STATE_AUTHENTICATION_FAILED

If the Request authentication failed, the *ProxyCoreObj* assumes the RVPROXY_CORE_OBJ_STATE_AUTHENTIATION_FAILED state. In this state, the application can choose whether to reject the request by calling Reject() or disregard the authentication failure and proceed to the RVPROXY_CORE_OBJ_STATE_RSOLVING_DESTINATION state by calling DisrigardInvalidRequest().

RVPROXY_CORE_OBJ_STATE_RESOLVING_DESTINATION

In this state, the *ProxyCoreObj* tries to determine the request destination (which can be one or more addresses). If the resolved list is empty (meaning that no destination address was found) the *ProxyCoreObj* assumes the

RVPROXY_CORE_OBJ_STATE_DESTINATION_ADDRESS_NOT_FOUND state. Otherwise the state its state be changed to RVPROXY_CORE_OBJ_STATE_ADDRESS_RESOLVED.

RVPROXY_CORE_OBJ_STATE_ROUTE_PROCESSED

The *ProxyCoreObj* assumes this state after the execution of "Route Information Preprocessing". Route Information Preprocessing is a part of address resolution (RFC 3261, section 16.4) which was designed to convert old routing methods (strict routing and maddr) to the new method of "loss routing". The *ProxyCoreObj* then continues with the address resolusion process by executing the "Determine Request Target" (RFC 3261, section 16.5) which resolves the request target and causes the *ProxyCoreObj* to assume the XXX_ADDRESS_RESOLVED state (if the request detination was found) or XXX_ADDRESS_NOT_FOUND (if no destination was found). If the application knows the request target by any other means, it can skip the execution of the "Determine Request Target" by calling RvProxyCoreObjSetDestAddress() syncronically within the RvProxyCoreObjStateChangeEv() when the ROUTE_PROCESSED state is reported.

RVPROXY_CORE_OBJ_STATE_DESTINATION_ADDRESS_NOT_FOUND

Indicates that the *ProxyCoreObj* was unable to determine to which destination address the request should be forwarded. In this state, the application can supply one or more destination addresses to which the request should be forwarded using SetDestAddress(). Or, the application may reject the request using a match response code.

RVPROXY_CORE_OBJ_STATE_ADDRESS_RESOLVED

Indicates that one or more destination address for this request was found. In this state, the application can proxy the request (in sequential or parallel mode) or reject it.

RVPROXY_CORE_OBJ_STATE_SEQ_PROXYING_REQUEST

Indicates that the application chose to proxy a request in sequential mode. Requests are sent, one at a time, to the destination address according to the list of resolved addresses in the *ProxyCoreObj*. If a 3xx response is received, and RecurseOn3xx is on, the contacts from the 3xx response will be added to the end of the address list.

RVPROXY_CORE_OBJ_STATE_PARALLEL_PROXYING_REQUEST

Indicates that the application chose to proxy a request in parallel mode. Requests are sent simultaneously to all destination addresses according to the list of resolved addresses in the *ProxyCoreObj*. If a 3xx response is received, and RecurseOn3xx is on, new requests will be sent to all contacts from the 3xx simultaneously.

RVPROXY_CORE_OBJ_STATE_REQUEST_CANCELLED

When a valid CANCEL was received for a server transaction or the application initiated Cancel(), the *ProxyCoreObj* cancels all pending client transactions and waits for a final response.

RVPROXY_CORE_OBJ_STATE_REQUEST_6XX_CANCELLING

When a 6xx is received for an Invite request that was sent (on one of the *ProxyCoreObj* branches), the *ProxyCoreObj* cancels all pending client transactions and waits for a final response. After all clients received a 487 or timed out, the 6xx response will be forwarded as a final response.

RVPROXY_CORE_OBJ_STATE_GENERAL_FINAL_RESPONSE_SENT

A Final response was forwarded or sent by the SIP Server on a non-INVITE transaction.

RVPROXY_CORE_OBJ_STATE_AWAITING_ACK

A non-2xx final response was forwarded or sent by the SIP Server on an INVITE transaction. The *ProxyCoreObj* is waiting for an ACK to this response.

RVPROXY_CORE_OBJ_STATE_ACK_RCVD

An ACK was received for the server transaction, after a non-2xx final response was sent.

RVPROXY_CORE_OBJ_STATE_INVITE_2XX_RESPONSE_SENT

A 2xx final response was forwarded on an INVITE transaction.

RVPROXY_CORE_OBJ_STATE_REQUEST_SENT

The *ProxyCoreObj* will assume this state after a request was sent and in case this is not a request that was forwarded, but a request that was initiated by the SIP Server. For example, when a CANCEL request is received, it is handled by

the SIP Server hop to hop, meaning that the SIP Server initiates sending a CANCEL request. In this the *ProxyCoreObj* will assume this state after the request was sent.

RVPROXY_CORE_OBJ_STATE_FINAL_RESPONSE_RCVD

A final response was received on a request that the SIP Server initiated. Since the SIP Server initiated sending the request, the response is for the SIP Server and the SIP Server does not need to forward this response.

RVPROXY_CORE_OBJ_STATE_BEFORE_TERMINATED

The last state before the final termination of the *ProxyCoreObj*. This is the last chance for the application to retrieve information from the *ProxyCoreObj*.

RVPROXY_CORE_OBJ_STATE_TERMINATE

The final state of the *ProxyCoreObj*. Upon reaching the RVPROXY_CORE_OBJ_STATE_TERMINATE state, the application can no longer refer to the *ProxyCoreObj*.

STATE MACHINE DIAGRAMS

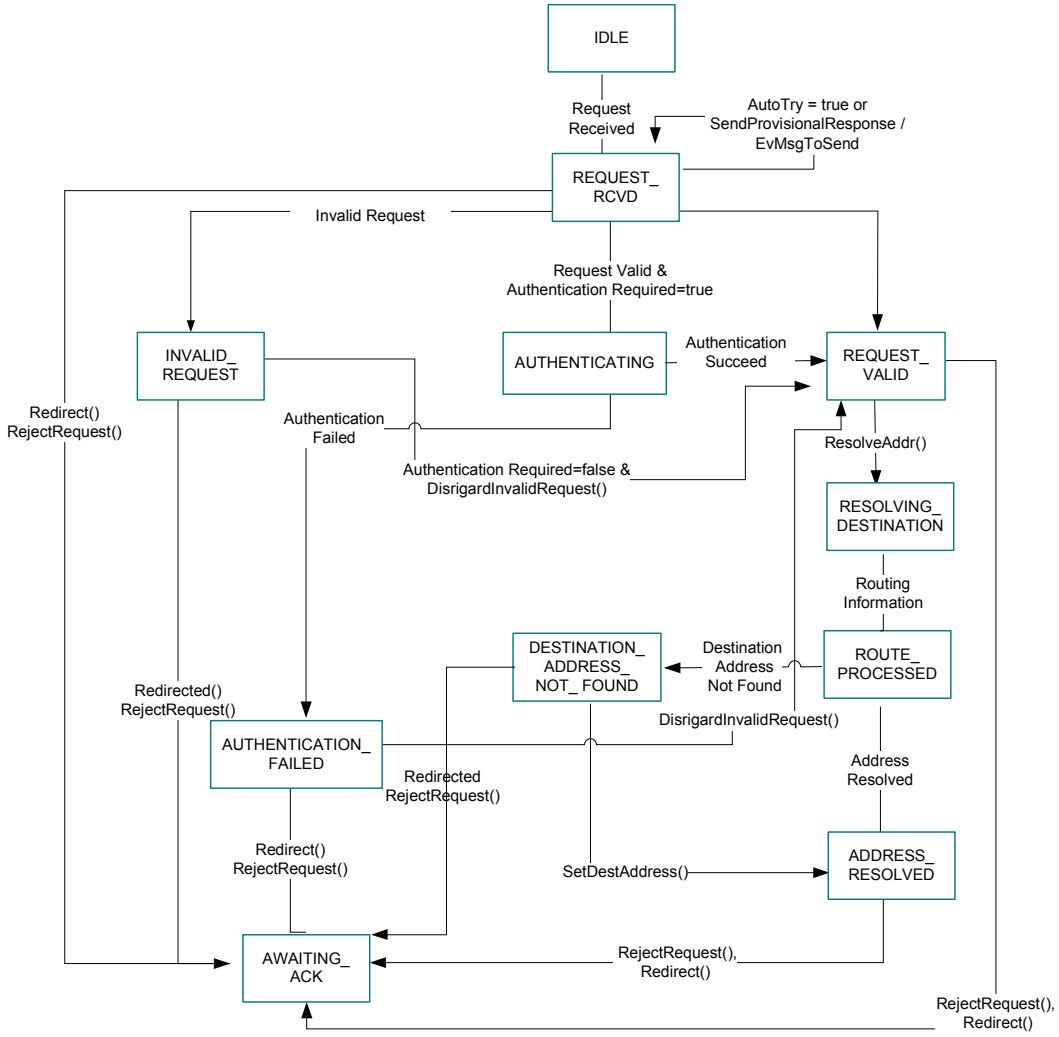


Figure 4-2 *Handling Incoming Request Proxy Core Object State Machine*

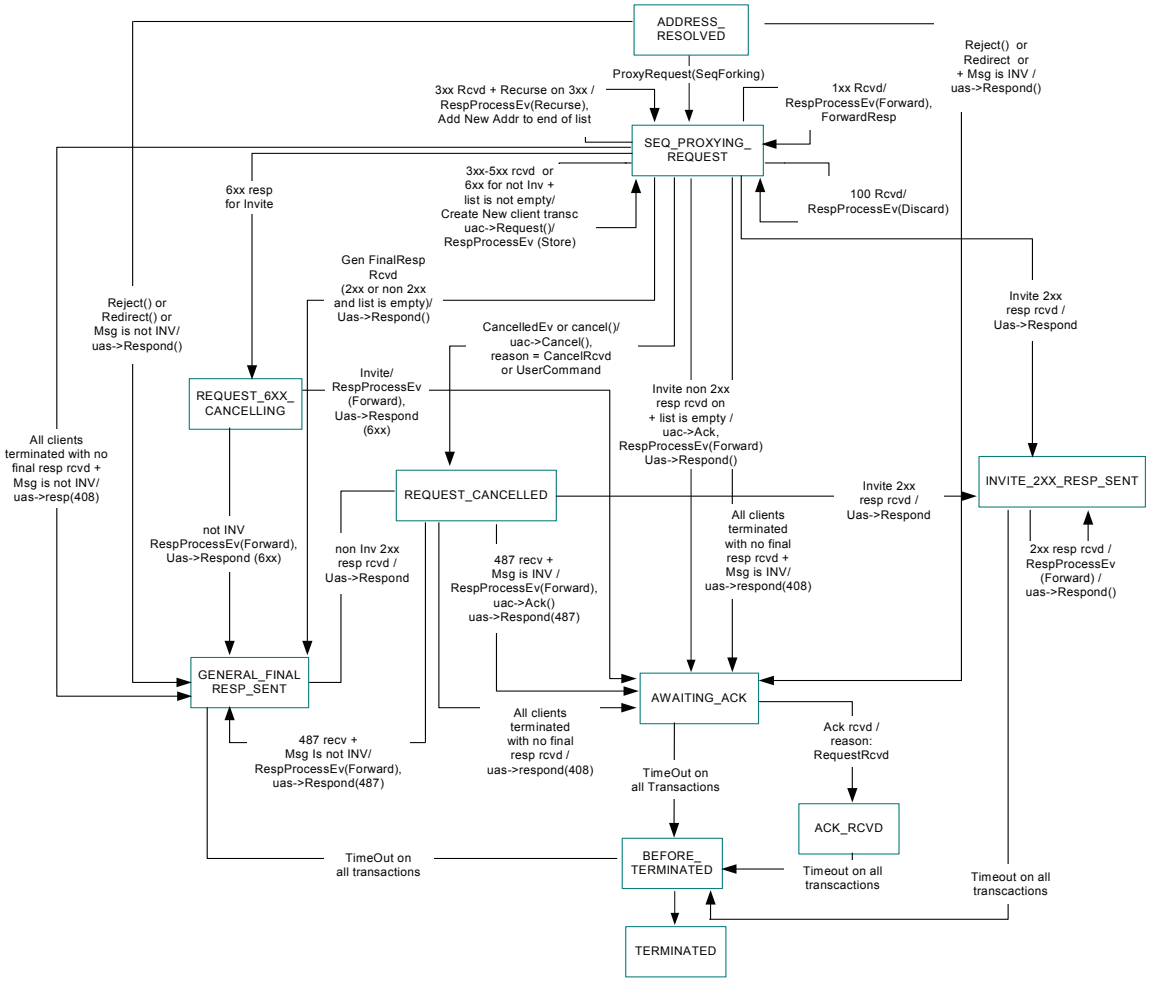


Figure 4-3 *Transaction Stateful ProxyCoreObj—Sequential Proxying State Machine*

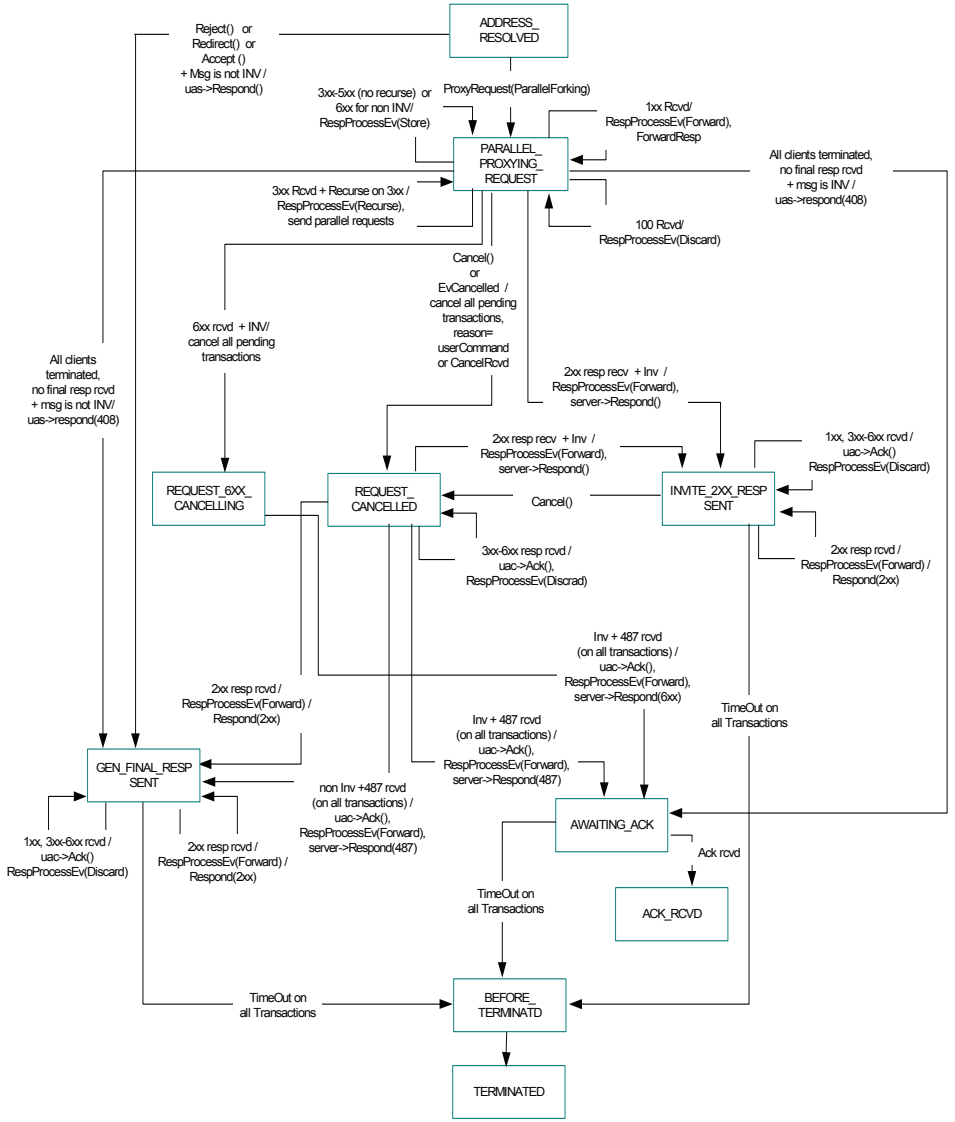


Figure 4-4 *Transaction Stateful ProxyCoreObj—Parallel Proxying State machine*

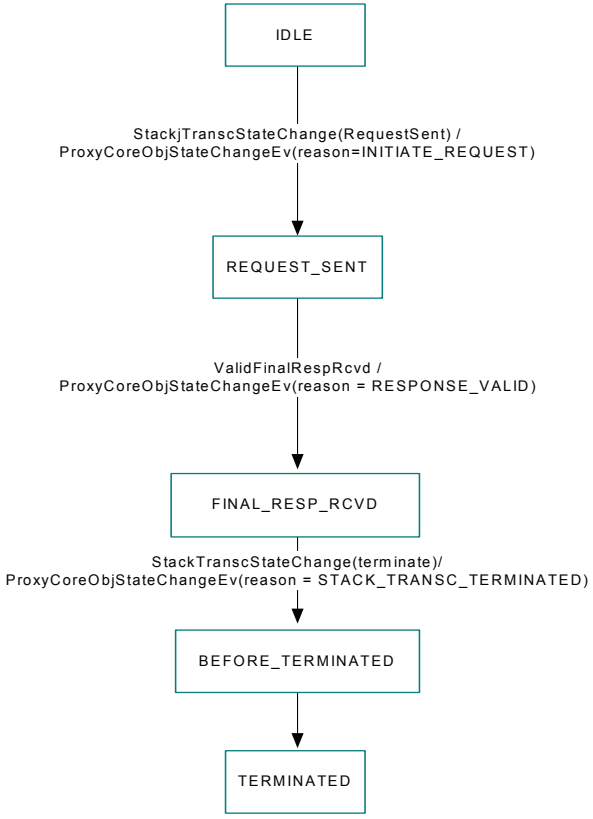


Figure 4-5 *Transaction Stateful ProxyCoreObj—Request Initiated by the SIP Server State Machine*

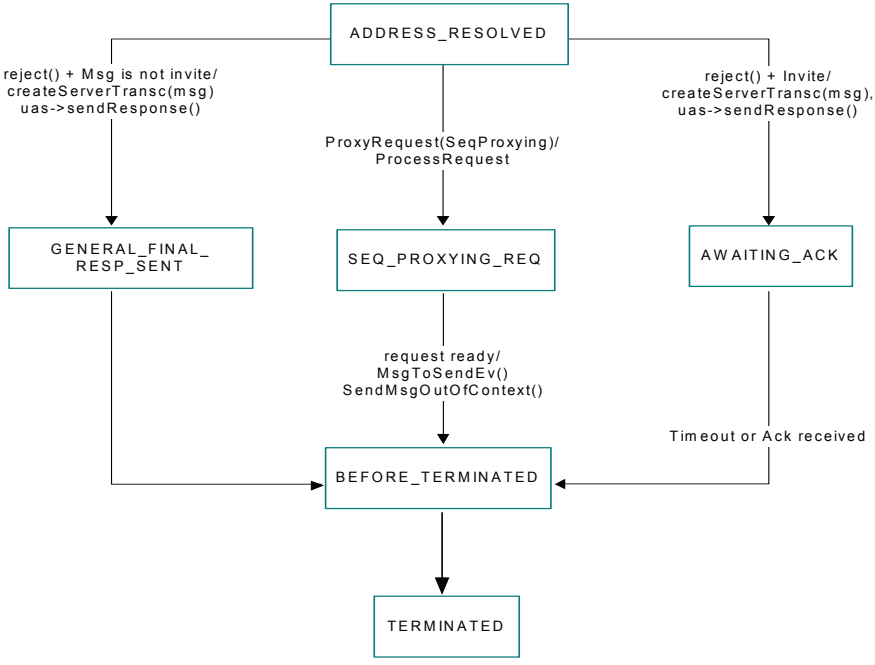


Figure 4-6 *Stateless ProxyCoreObj Forwarding Request State Machine*

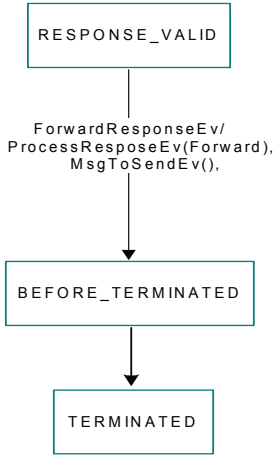


Figure 4-7 *Stateless ProxyCoreObj Forwarding Valid Response State Machine*

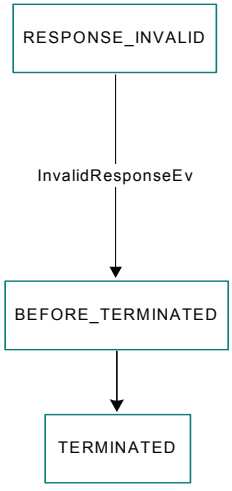


Figure 4-8 *Stateless ProxyCoreObj Forwarding Invalid Response State Machine*

Proxy Transaction State Machine

The Proxy Transaction state machine represents the progress of the state of a *ProxyTransc* from the initial state when a request is sent until a final response is received, when working in Transaction Stateful mode. The RvProxyCoreTranscStateChangeEv() callback reports *ProxyTransc* state changes and state change reasons. The state change reason indicates how the *ProxyTransc* reached the new state.

RVPROXY_CORE_TRANSC_STATE_IDLE

The initial state of the Proxy Transaction state machine. Upon creation of the *ProxyTransc*, the *ProxyTransc* assumes the RVPROXY_CORE_TRANSC_STATE_IDLE state. It remains in this state until a Request is sent.

RVPROXY_CORE_TRANSC_STATE_REQUEST_SENT

The *ProxyTransc* assumes this state after sending an outgoing request.

RVPROXY_CORE_TRANSC_STATE_MSG_SEND_FAILURE

Indicates that sending a message (request) failed. When the *ProxyTransc* assumes this state, the application can choose whether to continue sending the request using RvProxyCoreTranscDnsContinue(), or terminate the current *ProxyTransc* using RvProxyCoreTranscDnsTerminate().

The *ProxyTransc* stays in this state until the application chooses whether to terminate or continue. If the application chooses to continue, the *ProxyTransc* will assume the RVPROXY_CORE_TRANSC_STATE_REQUEST_SENT state.

RVPROXY_CORE_TRANSC_STATE_BEFORE_TERMINATED

The last state before final termination of the *ProxyTransc*. This is the last chance for the application to retrieve information from the *ProxyTransc*.

RVPROXY_CORE_TRANSC_STATE_TERMINATED

The final state of the *ProxyTransc*. Upon reaching the Terminated state, the application can no longer refer to the *ProxyTransc*.

State Machine Diagram

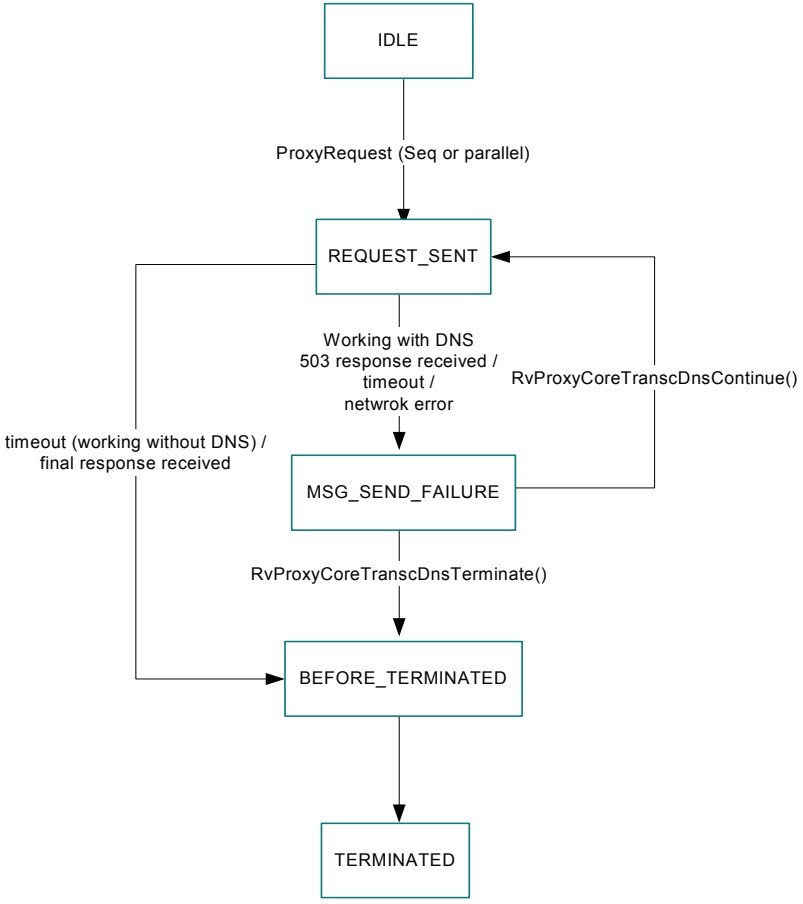


Figure 4-9 *Proxy Transaction State Machine*

Proxy Transmitter State Machine

The Proxy Transmitter state machine represents the progress of the state of a *ProxyTrx* from the initial state when a message is received until it is sent, when working in Stateless mode. The RvProxyCoreTrxStateChangeEv() callback reports *ProxyTrx* state changes and state reasons. The state change reason indicates how the *ProxyTrx* reached the new state.

RVPROXY_CORE_TRX_STATE_IDLE

The initial state of the *ProxyTrx*. Upon *ProxyTrx* creation, the *ProxyTrx* assumes the IDLE state. It remains in this state until RvProxyCoreTrxSendMessage() is called. The *ProxyTrx* then moves to the RESOLVING_ADDR state.

RVPROXY_CORE_TRX_STATE_RESOLVING_ADDR

Indicates that the *ProxyTrx* is about to start the address resolution process. After RvProxyCoreTrxSendMessage() is called, the *ProxyTrx* moves to the RESOLVING_ADDR state. The *ProxyTrx* then uses RFC 3261 rules to get the remote URI from the message object, 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 *ProxyTrx* moves to the FINAL_DEST_RESOLVED state.

RVPROXY_CORE_TRX_STATE_FINAL_DEST_RESOLVED

Indicates that the *ProxyTrx* has completed the DNS address resolution process and that it has a destination address to use (IP, port and transport). The application can use the RvProxyCoreTrxGetDestAddress() function to get the destination address. The application can change the destination address using the RvProxyCoreTrxSetDestAddress() function. The FINAL_DEST_RESOLVED state is the last chance for the application to fix the Via header. The message will be sent when the state changed callback returns.

RVPROXY_CORE_TRX_STATE_READY_FOR_SENDING

Indicates that the *ProxyTrx* 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.

RVPROXY_CORE_TRX_STATE_MSG_SEND_FAILURE

Indicates that an error occurred and the *ProxyTrx* will not be able to send the message. The change state reason indicates the type of failure.

In the MSG_SEND_FAILURE state, the application must do one of two things:

- Terminate the *ProxyTrx* using the RvProxyCoreTrxTerminate() function.
- Send the message again to the next address in the DNS list by calling the RvProxyCoreTrxSendMessage() function. This option is available only if the enhanced DNS feature is enabled.

If the application set the KeepMsgFlag, the original message was not destructed and will be sent again without further processing.

Remarks:

- When choosing to send the request again, this time to the next address in the list, the proxy will send the request statefully. The client *ProxyTransc* will be created and the *ProxyTrx* will be terminated (see RFC 3263). The *ProxyTrx* will not be terminated until the application takes one of the above actions.
- The MSG_SEND_FAILURE state is the way the *ProxyTrx* 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 *ProxyTrx* will not move to the MSG_SEND_FAILURE state. However, network errors will always cause the *ProxyTrx* to assume the MSG_SEND_FAILURE state.

RVPROXY_CORE_TRX_STATE_MSG_SENT

Indicates that the message was sent successfully to the remote party. This state is an indication that the application should now terminate the *ProxyTrx* object that has completed all its tasks and cannot be used any longer.

RVPROXY_CORE_TRX_STATE_TERMINATED

This is the final state of the *ProxyTrx*. Upon reaching the TERMINATED state, you can no longer reference the *ProxyTrx* object.

STATE MACHINE DIAGRAM

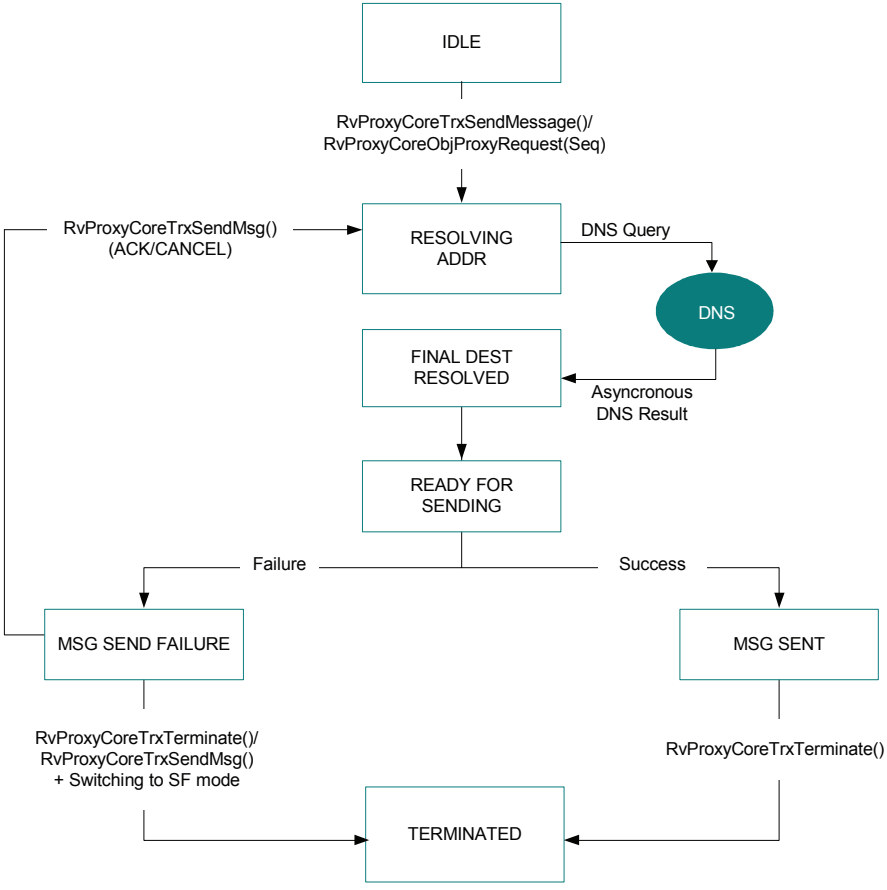


Figure 4-10 *Proxy Transmitter State Machine*

WORKFLOW DIAGRAMS

DIAGRAM ONE

Figure 4-11 describes proxying a request.

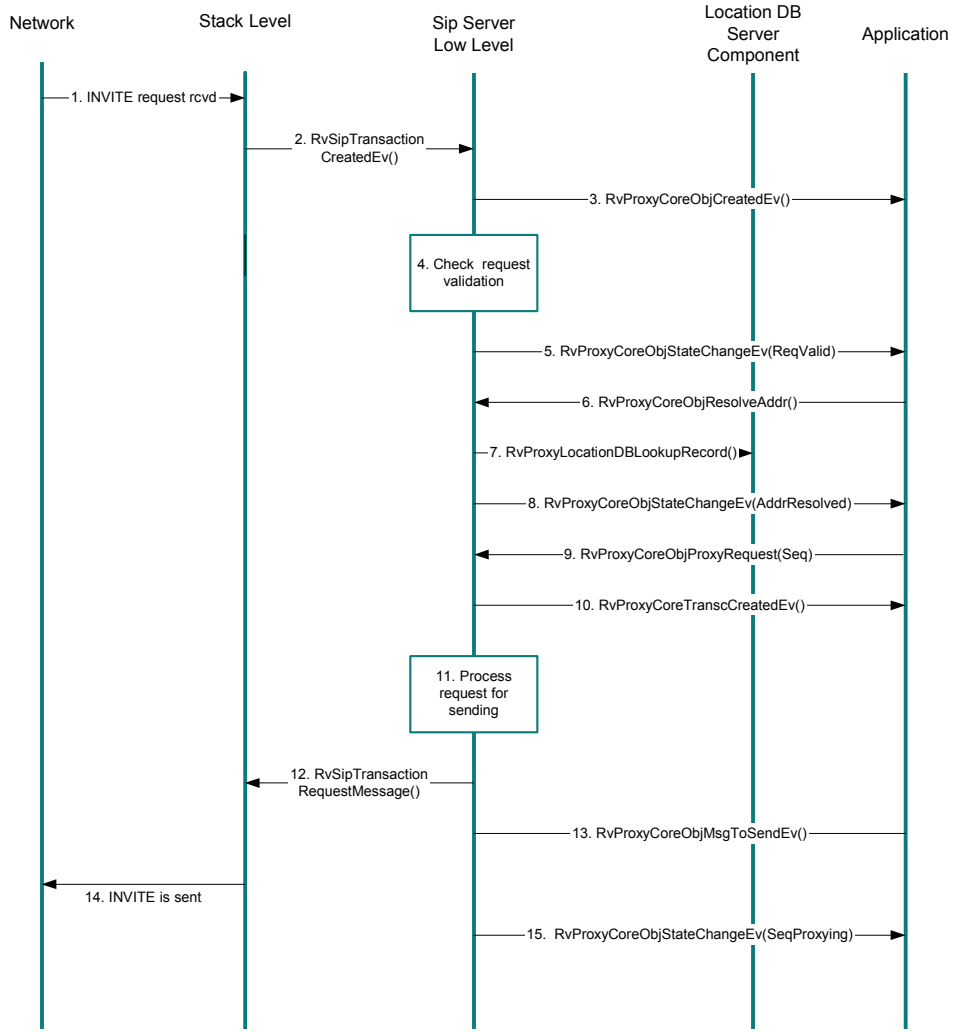


Figure 4-11 *Proxying a Request*

Proxying a request includes the following stages:

1. A new incoming INVITE request was received.
2. The SIP Stack creates a new Stack Transaction to handle the incoming request and notifies the SIP Server Low Level.
3. The SIP Server Low Level creates a new stateful *ProxyCoreObj* (and *ProxyTransc* of server type) to handle the new request, and notifies the application. The application can create its own object that will be associated with the *ProxyCoreObj*.
4. The *ProxyCoreObj* checks request validity according to RFC requirements and local configuration (such as Max-forward and Authentication if required). In this case, the request is valid.
5. Notifies the application that the *ProxyCoreObj* state is REQUEST_VALID.
6. The application requests to resolve the destination addresses for the request.
7. While resolving the destination addresses for the request, the SIP Server Low Level consults the Location Database Server Component on whether there is a record for the address of the incoming request, and what are the bindings for this address.
8. After constructing the destination addresses list, the application receives notification that the *ProxyCoreObj* state is ADDRESS_RESOLVED and the application can choose whether to reject, redirect or proxy the request according to its local policy.
9. The application chooses to proxy the request in sequential mode.
10. While handling the process of the request, a new *ProxyTransc* (of client type) is created for this request. The application can create its own object that will be associated with the new *ProxyTransc*.
11. The SIP Server Low Level processes the request for sending (such as adding a Record Route header if required and updating Max-Forward).
12. The SIP Server Low Level asks the Stack Transaction to send the request.

13. The SIP Server notifies the application about the message that is going to be sent. The application can modify the message at this callback.
14. The request is sent into the network.
15. The application receives notification that the *ProxyCoreObj* state was changed to SEQ_PROXYING.

DIAGRAM TWO

Figure 4-12 describes creating and sending an out-of-dialog request (stateful mode).

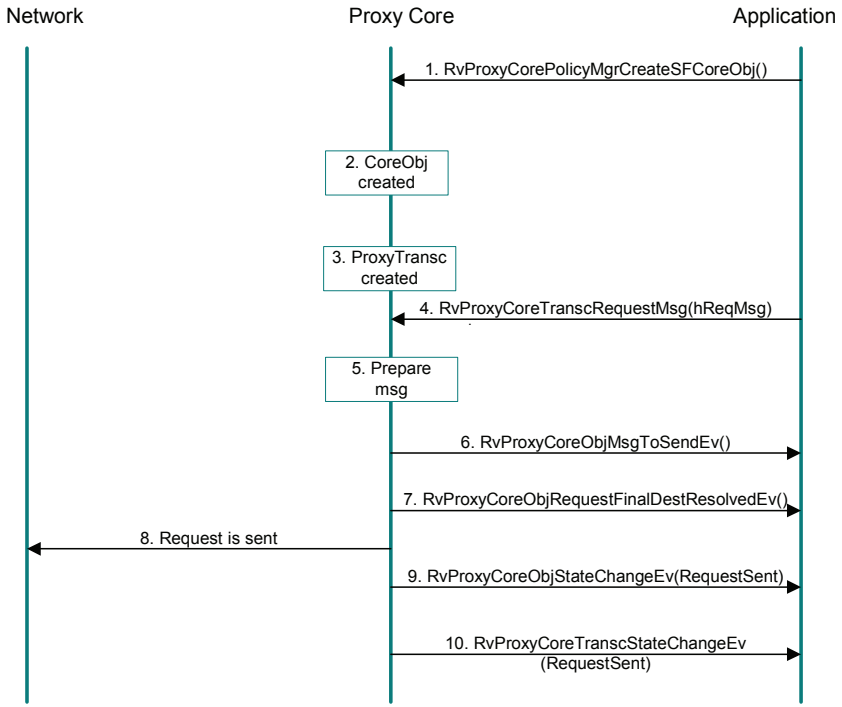


Figure 4-12 *Creating and Sending a Request*

To initiate a request

1. The application requests to create a stateful *ProxyCoreObj*.
2. The SIP Server creates a stateful *ProxyCoreObj*.
3. The SIP Server creates a client *ProxyCoreTransc*.

4. The application requests to send the supplied message according to the Request-URI of the request.
5. The SIP Server processes the request for sending (such as updating the Via header if required).
6. The SIP Server notifies the application about the message that is going to be sent. The application can modify the message at this callback.
7. The SIP Server notifies the application that it has constructed the destination addresses list.
8. The request is sent into the network.
9. Notifies the application that the *ProxyCoreObj* state is REQUEST_SENT.
10. Notifies the application that the *ProxyCoreTransc* state is REQUEST_SENT.

DIAGRAM THREE

Figure 4-13 describes stateless forwarding of an incoming request

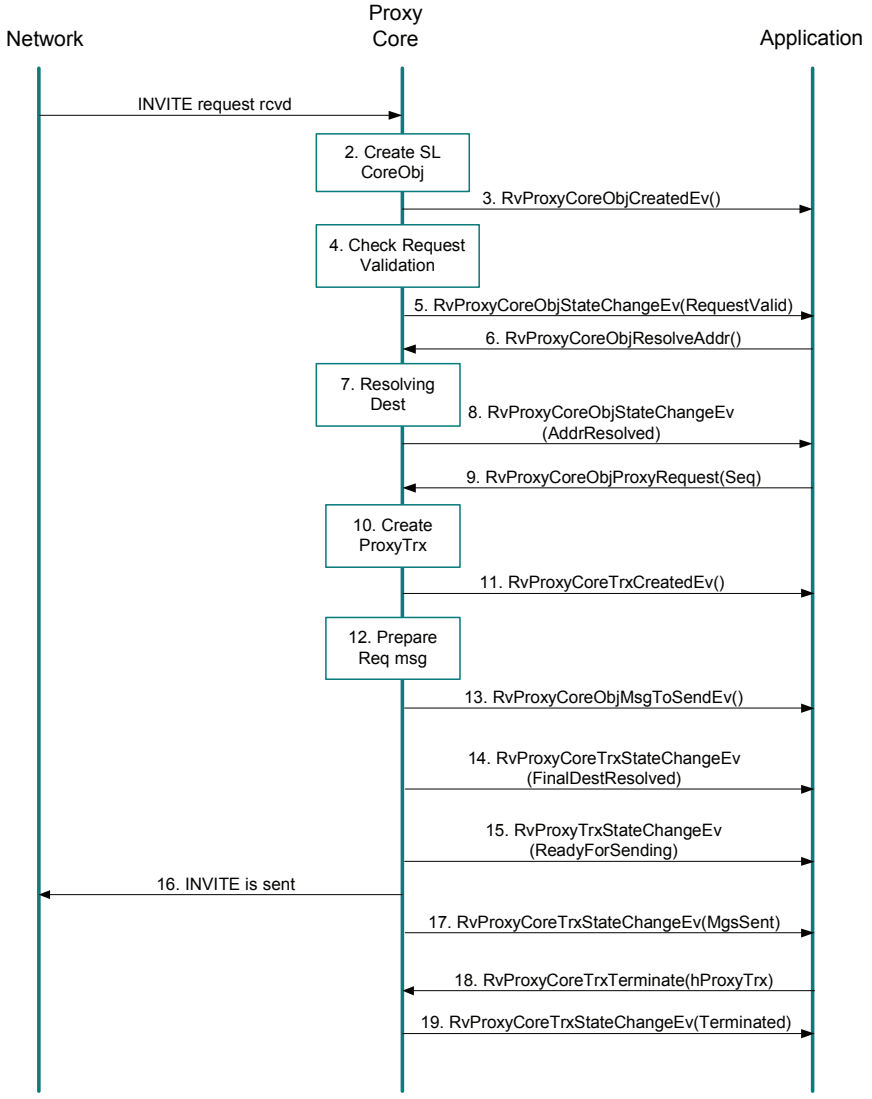


Figure 4-13 *Stateless Forwarding*

To proxy a request

1. A new incoming request was received. The SIP Stack notifies the SIP Server.
2. The application chooses to handle the request in stateless mode. The SIP Server creates a new stateless *ProxyCoreObj* to handle the new request.
3. The SIP Server notifies the application regarding the *ProxyCoreObj* creation. The application can create its own object that will be associated with the *ProxyCoreObj*.
4. The *ProxyCoreObj* checks the request validity according to RFC requirements and local configuration (such as Max-forward and authentication if required). In this case the request is valid.
5. Notifies the application that the *ProxyCoreObj* state is REQUEST_VALID.
6. The application requests to resolve the destination address for the request.
7. Resolving the destination addresses for the request according to RFC 3261.
8. After constructing the destination addresses list, the application receives notification that the *ProxyCoreObj* state is ADDRESS_RESOLVED and the application can choose whether to reject or proxy the request according to its local policy.
9. The application chooses to proxy the request in sequential mode (the only mode allowed when working as a stateless proxy).
10. The SIP Server creates a new *ProxyTrx* to proxy the request.
11. The SIP Server notifies the application regarding the *ProxyTrx* creation. The application can create its own object that will be associated with the *ProxyTrx*.
12. The SIP Server processes the request for sending (such as adding Record Route if required and updating Max-Forward).
13. The SIP Server notifies the application about the message that is going to be sent. The application can modify the message at this callback.

14. Notifies the application that the *ProxyTrx* state is FINAL_DEST_RESOLVED.
15. Notifies the application that the *ProxyTrx* state is READY_FOR_SENDING.
16. The request is sent into the network.
17. Notifies the application that the *ProxyTrx* state is MSG_SENT.
18. The application requests to terminate the *ProxyTrx*.
19. Notifies the application that the *ProxyTrx* state is TERMINATED. The application can no longer use this object.

SAMPLE CODE

REGISTERING APPLICATION CALLBACKS

The following code demonstrates how to register application callbacks of the RvProxyCoreObjEvHandler structure.

To register application callbacks

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

Sample Code

```
/*=========================================================================================*/
void AppPolicyMgrSetCoreObjEvHandler(RvProxyCoreObjEvHandler hPolicyMgr)
{
  /*step 1*/
  RvProxyCoreObjEvHandler appEvHandlers;
  /*step 2*/
  memset(&appEvHandlers,0,sizeof(RvProxyCoreObjEvHandler));
  /*step 3*/
  appEvHandlers.pfnEvCoreObjStateChange = AppCoreObjStateChangeEv;
  /*step 4*/
 RvProxyCorePolicyMgrSetCoreObjEvHandler(hPolicyMgr, &appEvHandlers,
                                                    sizeof(RvProxyCoreObjEvHandler));
}
/*=========================================================================================*/
```

HANDLING CORE OBJECT CREATION

To handle a Core Object creation

1. Allocate a new application handle for the new Core object.
2. Set Core object attributes.
3. Exchange the handle with the Proxy Core.

Sample Code

```
/*-------------------------------------------------------------------*/
void RVCALLCONV AppCoreObjectCreatedEv(
   IN RvProxyCoreObjHandle             hCoreObj,
   OUT RvProxyCoreObjAppHandle*        pAppCoreObj)
{
      AppCoreObj  *pNewAppObj = NULL;

      pNewAppObj = (AppCoreObj*)malloc(sizeof(AppCoreObj));
      if(NULL == pNewAppObj)
      {
            printf("Failed to create an application object");
            return;
      }
      else
      {
        /* A good time to set the Core object attributes. */

        /* Want to authenticate the request. */
        rv = RvProxyCoreObjSetAuthRequired(hCoreObj,
        RV_TRUE);
        if(RV_Success != rv)
        {
            printf("Failed to Set Authentication Required");
            return;
        }

        /* Want to use Record-Route mechanism on request only. */
        rv = RvProxyCoreObjSetRecordRouteMode
        (hCoreObj,RVSIPSERVER_RECORD_ROUTE_MODE_REQ_ONLY);
        if(RV_Success != rv)
```

```
        {
            printf("Failed to Set Record-Route mode");
            return;
      }

        /* Exchanges handles with the Proxy Core. */
        *pAppCoreObj = (RvProxyCoreObjAppHandle)pNewAppObj;
        pNewAppObj->hCoreObj = hCoreObj;
      }


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

Implementation of the State Changed Callback Function

The following code demonstrates an implementation of the State Changed callback function.

Sample Code

```
/*=========================================================================*/
void RVCALLCONV AppCoreObjStateChangeEv(
           IN RvProxyCoreObjHandle         hCoreObj,
           IN RvProxyCoreObjAppHandle      pAppCoreObj,
           IN RvProxyCoreObjState          eState,
           IN RvProxyCoreObjReason         eReason)
{
     RvStatus rv = RV_Success;

    switch (eState)
    {
    case RVPROXY_CORE_OBJ_STATE_REQUEST_VALID:
        rv = RvProxyCoreObjResolveAddr(hCoreObj);
        if(RV_Success != rv)
          {
            printf("Failed to resolve address list \n");
            return;
           }
    break;
    case RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST:
       {
          switch(eReason)
          {
```

```
            case RVPROXY_CORE_OBJ_REASON_REQUEST_LOOP_DETECTED:
                rv = RvProxyCoreObjRejectRequest(hCoreObj, 482,
                NULL);
                if(RV_Success != rv)
                {
                 printf("Failed to Reject with 482 \n");
                 return;
                }
                break;
    case RVPROXY_CORE_OBJ_REASON_REQUEST_ILLEGAL_MAX_FORWARD:
          rv = RvProxyCoreObjRejectRequest(hCoreObj, 483,
          NULL);
          if(RV_Success != rv)
          {
              printf("Failed to Reject with 483 \n");
              return;
           }
           break;
       default:
           break;
       }
   }
   break;
   default:
      break;
 }
}
/*=================================================================*/
```

5

WORKING WITH THE REGISTER SERVER

INTRODUCTION

The Register Server layer of the SIP Server supplies a high-level interface for Registrar applications. The Register Server handles registration requests and responses.

The Register Server API allows the application to decide which registration requests should be handled by the server, and how to handle a specific registration request. A registration request can be processed locally by the Register Server if its destination is one of the domains for which the SIP Server is responsible.

An application can choose to accept or reject a registration request handled locally and can modify the registration information. The Register Server uses the Location Service Database (local or remote LDAP) for storing the registration data. If the application chooses not to handle the Register request, the request will be handled as a general request, using the Proxy Core.

REGISTER SERVER ENTITIES

The Register Server API relates to the following entities:

REGISTER SERVER MANAGER

The Register Server Manager (*RegServerMgr)* manages the Register Server Objects (*RegServerObj)*. The Register Server Manager API functions allow the application to set event handlers for the *RegServerObj*.

REGISTER SERVER OBJECT

A Register Server Object (*RegServerObj*) represents a registration request that should be handled locally by the Register Server. The Register Server Object API functions allow the application to accept or reject a registration request according to the application policy.

Each *RegServerObj* creates a record that holds the registration information from the Register request message. The *RegServerObj* is a stateful object that can assume any state from a set defined in the Register Server Type Definitions API. The state of the *RegServerObj* represents different stages of the registration process.

REGISTER SERVER DATA OBJECTS

The Register Server Data Objects are objects that are used for holding the registration information. The Register Server Data Objects API allows creating and modifying these data structures.

The Data Objects are listed below:

- ***RegServerRecord***—holds the registration information. A record can be created from a received registration request message, or by the application. Holds a list of bindings and a key that identifies the record.
- ***RegServerBinding***—holds registration data of a single registration entry. Includes a contact header, which is the address to which a user registers.
- ***RegServerKey***—identifies the record with a To header that represents the registering user.

Note The Register Server Data Objects are not protected for a multithreaded environment at the SIP Server level. It is the responsibility of the application to lock the objects to ensure multithreading safety when creating such objects.

REGISTER SERVER HANDLES

All *RegServerObj* objects are identified using handles. These handles must be supplied when calling API functions. The handles are listed below:

- **RvProxyRegServerMgrHandle**—defines the handle for the *RegServerMgr*. This handle can be received by calling RvSipServerMgrGetRegServerMgrHandle().

- **RvProxyRegServerObjHandle**—defines the handle for the *RegServerObj*. This handle is given to the application in the RvProxyRegServerObjCreatedEv().
- **RvProxyRegServerRecordHandle**—defines the handle for a *RegServerRecord* object. This handle can be received by calling RvProxyRegServerObjGetRecord(), or when creating a stand-alone record using RvProxyRegServerRecordConstruct().
- **RvProxyRegServerBindingHandle**—defines the handle for a *RegServerBinding* Object. This handle can be received by calling one of the following functions:
  - RvProxyRegServerRecordGetHeadBinding()
  - RvProxyRegServerRecordGetTailBinding()
  - RvProxyRegServerRecordGetNextBinding()
  - RvProxyRegServerRecordGetPrevBinding()
- **RvProxyRegServerKeyHandle**—defines the handle for a *RegServerKey* object. This handle can be received by calling RvProxyRegServerRecordGetKey(), or when creating a stand-alone key using RvProxyRegServerKeyConstruct().

Register Server Manager

The *RegServerMgr* controls the collection of *RegServerObj* objects that are created when a new registration request should be handled locally. The application can use the Register Server Manager API to register application callbacks with the SIP Server.

The function, RvProxyRegServerMgrSetEvHandler(), allows the application to set its event handlers (callback functions) pointers in the SIP Server. The Register Server API declares prototypes for all Register Server callback functions. For example:

```
/*-----------------------------------------------------------------*/
typedef void (RVCALLCONV * RvProxyRegServerObjStateChangedEv)
     (IN RvProxyRegServerObjHandle         hRegServerObj,
      IN RvProxyRegServerObjAppHandle      hAppRegServer,
      IN RvProxyRegServerObjState          eState,
      IN RvProxyRegServerObjReason         eReason);
/*-----------------------------------------------------------------*/
```

According to the prototypes, the application can implement any callbacks that are necessary.

Sample Code: Callback Implementation

```
/*============================================================*/
void RVCALLCONV AppRegServerObjStateChangedEv(
      IN RvProxyRegServerObjHandle       hRegServerObj,
      IN RvProxyRegServerObjAppHandle    hAppRegServer,
      IN RvProxyRegServerObjState        eState,
      IN RvProxyRegServerObjReason       eReason)
{
    if (RVPROXY_REGSERVER_OBJ_STATE_VALID_REG == eState)
    {
        RvProxyRegServerObjAcceptRegistration(hRegServerObj,
                        (0,
                         NULL,
                         RV_TRUE);
    }
}
/*============================================================*/
```

Figure 5-1 *Callback Implementation*

Callback Function Structure

All callback functions are gathered together in a structure called RvProxyRegServerEvHandler. This structure is where the application should set its callback function pointers and give it as a parameter to the RvProxyRegServerMgrSetEvHandler() function. The Register Server will notify the application of an event using the callback functions implemented.

To assign a callback

1. Declare an RvProxyRegServerEvHandler structure.
2. Initialize the structure members to NULL.
3. Set the application defined callbacks in RvProxyRegServerEvHandler.
4. Call RvProxyRegServerMgrSetEvHandler().

Register Server API

The Register Server API includes Register Server parameters and Register Server control functions, as explained in the following sections.

Register Server Parameters

The Register Server Parameters consist of:

- Register Server Object parameters

- Register ServerRecord parameters
- Register Server Binding parameters
- Register Server Key parameters

RegServerObj Parameters

The application can set or examine Register Server Object parameters via Set and Get API functions. The following parameters are available:

Record

The record is created from the Register message when the *RegServerObj* is created. The record of the *RegServerObj* can be accessed with a Get function. The Get function will return the handle of the record, which can be used to modify it.

State

The state parameter of the *RegServerObj* indicates the state of the registration process. The state parameter can only be accessed with a Get function. The state is not modifiable.

Received Message

The *RegServerObj* holds a copy of the Register request that was received. The message handle can be accessed with a Get function. The message can be retrieved only until after the *RegServerObj* reaches the VALID or INVALID state.

Authentication Required

Each *RegServerObj* holds a parameter that indicates whether the Register request that is handled by this object should be authenticated. The bAuthenticate flag can be accessed and set with Set and Get functions.

Add Auth Header To Response

Indicates whether the *RegServerObj* will add WWW-Authentication or Proxy-Authentication headers when the application initiated a 401 or 407 response (respectively). This parameter can be accessed using Set functions.

Handle Parameters

RegServerMgr

The *RegServerObj* holds a handle to the *RegServerMgr*. The *RegServerMgr* handle can be accessed with a Get function.

ApplicationHandle

The *RegServerObj* holds a handle to the application owner of this object. The application handle can be accessed and set with Get and Set functions. This handle will be supplied when calling callback functions for events for this *RegServerObj*.

RegServerRecord Parameters

A *RegServerRecord* is created from the Register message when the *RegServerObj* is created. The application can also construct (and destruct) a stand-alone *RegServerRecord*.

The following parameters are available:

Binding List

A binding in the binding list may be Set or Get according to its place in the list (first, last, next or previous), or may be removed from the bindings list. The application can get a handle to the *RegServerBinding* from the *RegServerRecord* and modify it.

Key

Each *RegServerRecord* has a *RegServerKey* that identifies it. The *RegServerKey* can be set or examined via Set and Get API functions.

RegServerBinding Parameters

A *RegServerBinding* is created when a *RegServerRecord* is constructed from a registration message. Each *RegServerBinding* represents a Contact header in the registration message. When a new *RegServerBinding* is constructed by the application in a *RegServerRecord*, the Contact header, CSeq header, and call-ID parameters must be set. A *RegServerBinding* created by the application is not valid until all its parameters are set.

Contact header, CSeq header, Call-Id

The application can examine and set each of these Register Server Binding parameters via the Set and Get API functions.

RegServerKey Parameters

A *RegServerKey* is created when a *RegServerRecord* is constructed. The application can also construct and destruct a stand-alone *RegServerKey*.

TO Header

The application can examine and set the TO header parameter of the *RegServerKey* through Get and Set API functions. After constructing a stand-alone *RegServerKey*, its TO header must be set.

Register Server Control

The following API functions provide control for the Register Server.

RvProxyRegServerObjAcceptRegistration()

After a *RegServerObj* was created and reached the VALID_REG state, this function may be used to accept the registration. The application can choose the response code and reason for accepting the registration, and choose whether the Location Service Database should be updated with this registration. Accepting the registration will result in sending a response to the registering client. The response will include all updated registration information in contact headers.

RvProxyRegServerObjRejectRegistration()

After a *RegServerObj* was created and reached the VALID_REG or INVALID_REG state, this function may be used to reject this registration. The application can choose the response code and reason for rejecting the registration. A response will be sent to the registering client.

RvProxyRegServerObjDisregardFailure

If the *RegServerObj* reaches the INVALID_REG state (the request was invalid or authentication failed), the application can choose to disregard the failure and change the *RegServerObj* state to VALID_REG.

RvProxyRegServerObjTerminate()

The application can choose to terminate the *RegServerObj* in any state. All resources of the *RegServerObj* will be released and this handle should not be used by the application anymore.

Register Server Events

The Register Server API supplies several events, in the form of callback functions, to which the application may listen and react.

In order to listen to an event, the application should first define a special structure called the event handler and then pass the event handler pointer to the *RegServerMgr*. When an event occurs, the Register Server calls the event handler function using the pointer. See the sample code in Figure 5-4.

The following events are supplied with the Register Server API:

RvProxyRegServerObjCreatedEv()

The Register Server calls this event after a *RegServerObj* was created. A *RegServerObj* is created when a Register request is handled locally by the Register Server. The *RegServerObj* is responsible for handling this Register request. An application object associated with the newly created *RegServerObj* can be supplied.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyRegServerObjMsgRcvdEv

This event is called when a Register request is handled locally by the Register Server. This event is called after RvProxyRegServerObjCreatedEv() is called. The application can access the message.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyRegServerObjStateChangedEv()

The StateChangedEv notifies the application of state changes and the associated state change reason. For example, upon receiving VALID_REG state notification, the application can choose whether to accept or reject this registration.

RvProxyRegServerRespMsgToSendEv()

The RespMsgToSend event is called before sending a final response to the registering client. Upon receiving this event, the response message may be modified.

Note It is not permitted to call the Terminate() API function from this callback.

RvProxyRegServerDBUpdateFailed()

The Register Server calls this event when an error occurred while updating the Location Database Server Component with the registration record, after the application chose to accept a registration and to update the Location Database

Server Component. The application can choose whether to reject the registration and send a 500 response to the registering client, or to ignore the failure and continue accepting the registration.

Register Server Object State Machine

The Register Server Object state machine represents the state of handling a registration request. RvProxyRegServerObjStateChangedEv() callback reports *RegServerObj* state changes and state change reasons. The state change reason indicates how or why the *RegServerObj* reached the new state.

The *RegServerObj* associates with the following states:

RVPROXY_REGSERVER_OBJ_STATE_IDLE

The IDLE state is the initial state of the Register Server Object state machine. When a *RegServerObj* is created it is initialized with the IDLE state. The *RegServerObj* changes its state to REQ_RCVD.

RVPROXY_REGSERVER_OBJ_REQ_RCVD

The *RegServerObj* changes its state to REQ_RCVD from IDLE. This state indicates that a Register request was received and is handled locally by this *RegServerObj*.

The *RegServerObj* validates the request in this state. If the request is invalid, the *RegServerObj* will change its state to INVALID_REG. If the request is valid and authentication is required, the *RegServerObj* will change its state to AUTHENTICATING. If authentication is not required, the *RegServerObj* will change its state to VALID_REG.

RVPROXY_REGSERVER_OBJ_STATE_AUTHENTICATING

The *RegServerObj* reaches this state if the Register request is valid and authentication is required. At this stage, the *RegServerObj* performs server-side authentication. If authentication fails, the *RegServerObj* will change its state to INVALID_REG. Otherwise, it will change its state to VALID_REG. This state is informative and the application is not expected to react.

RVPROXY_REGSERVER_OBJ_STATE_VALID_REG

The *RegServerObj* reaches the VALID_REG state after the validation and authentication (if required) of the request was successful. Validating the request includes checking the TO header, legal expirations, and legal star contacts according to SIP RFC definitions. From this state the application can choose whether to accept or reject the registration request.

RVPROXY_REGSERVER_OBJ_STATE_INVALID_REG

The *RegServerObj* reaches the INVALID_REG state after the validation or authentication of the request has failed. From this state the application can choose to reject the registration request or terminate the *RegServerObj*, or to disregard the failure and change the state to the VALID_REG state.

RVPROXY_REGSERVER_OBJ_STATE_FINAL_RESPONSE_SENT

The *RegServerObj* reaches the FINAL_RESPONSE_SENT state after sending a final response to the registering client.

RVPROXY_REGSERVER_OBJ_STATE_BEFORE_TERMINATE

The last state before final termination of the *RegServerObj*. This is the last chance for the application to access the *RegServerObj*.

RVPROXY_REGSERVER_OBJ_STATE_TERMINATE

The final state of the *RegServerObj*. Upon reaching the TERMINATE state, the application can no longer reference the *RegServerObj*.

RVPROXY_REGSERVER_OBJ_STATE_EXECUTING_QUERY

The *RegServerObj* reaches this state if the application accepted the valid Register request and the Location Service Database was configured to use Remote LDAP for Location service operations. At this stage, the *RegServerObj* queries/updates the Remote LDAP database with the REGISTER request data. If the query execution fails, the *RegServerObj* will reject the request. Otherwise, a 2xx final response will be sent.

STATE MACHINE

Figure 5-2 shows the Register Server Object state machine.

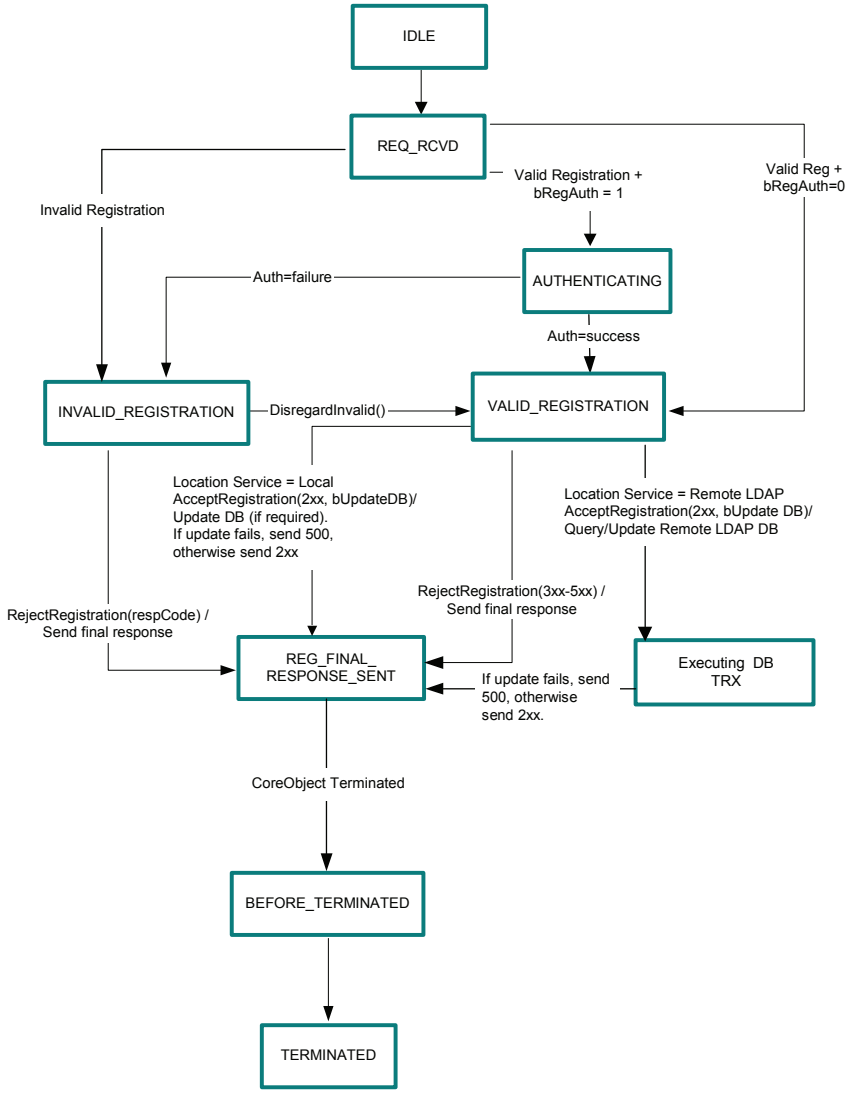


Figure 5-2 *Register Server State Machine*

WORKFLOW DIAGRAMS

The following diagrams describe accepting a valid registration and rejecting an invalid registration when the *RegServerObj* is configured to use the Local Database (Location Database Server Component) for the Location Service.

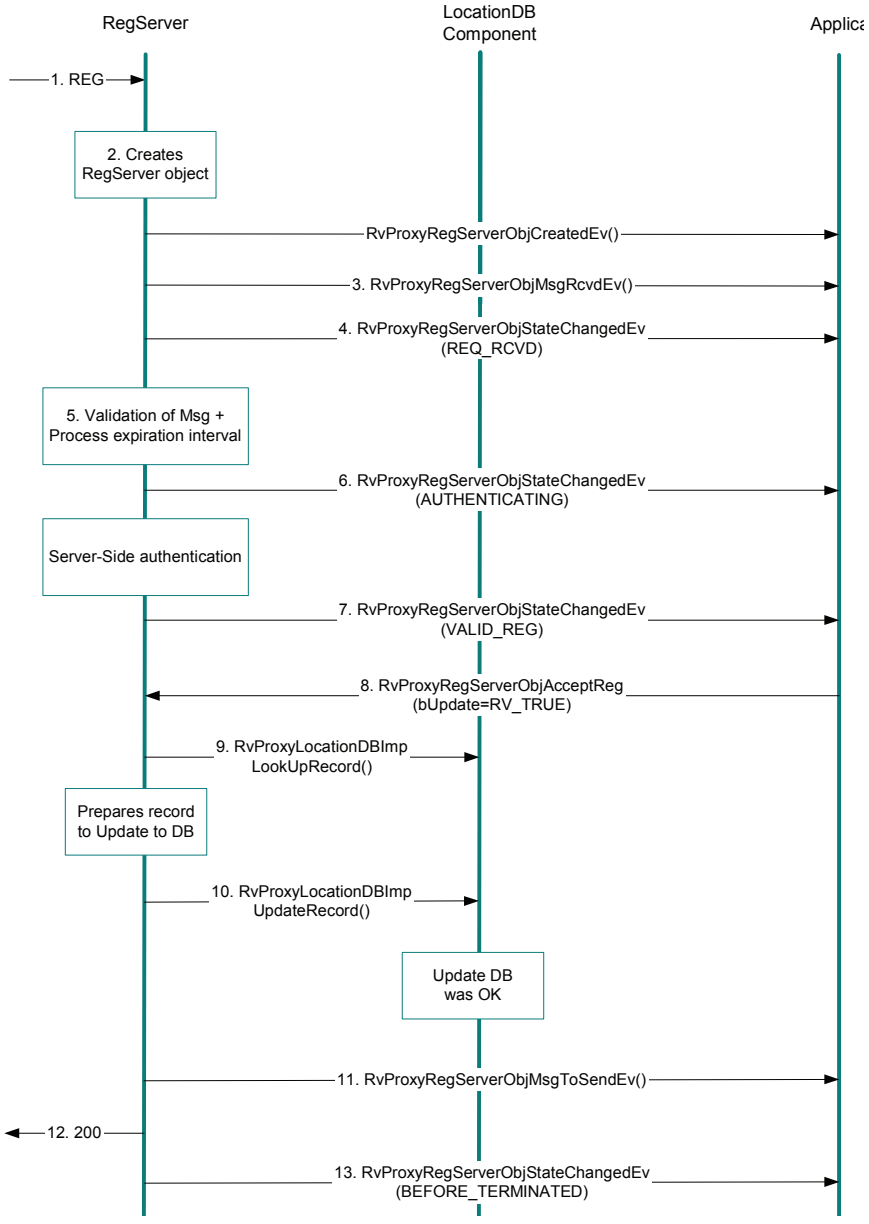
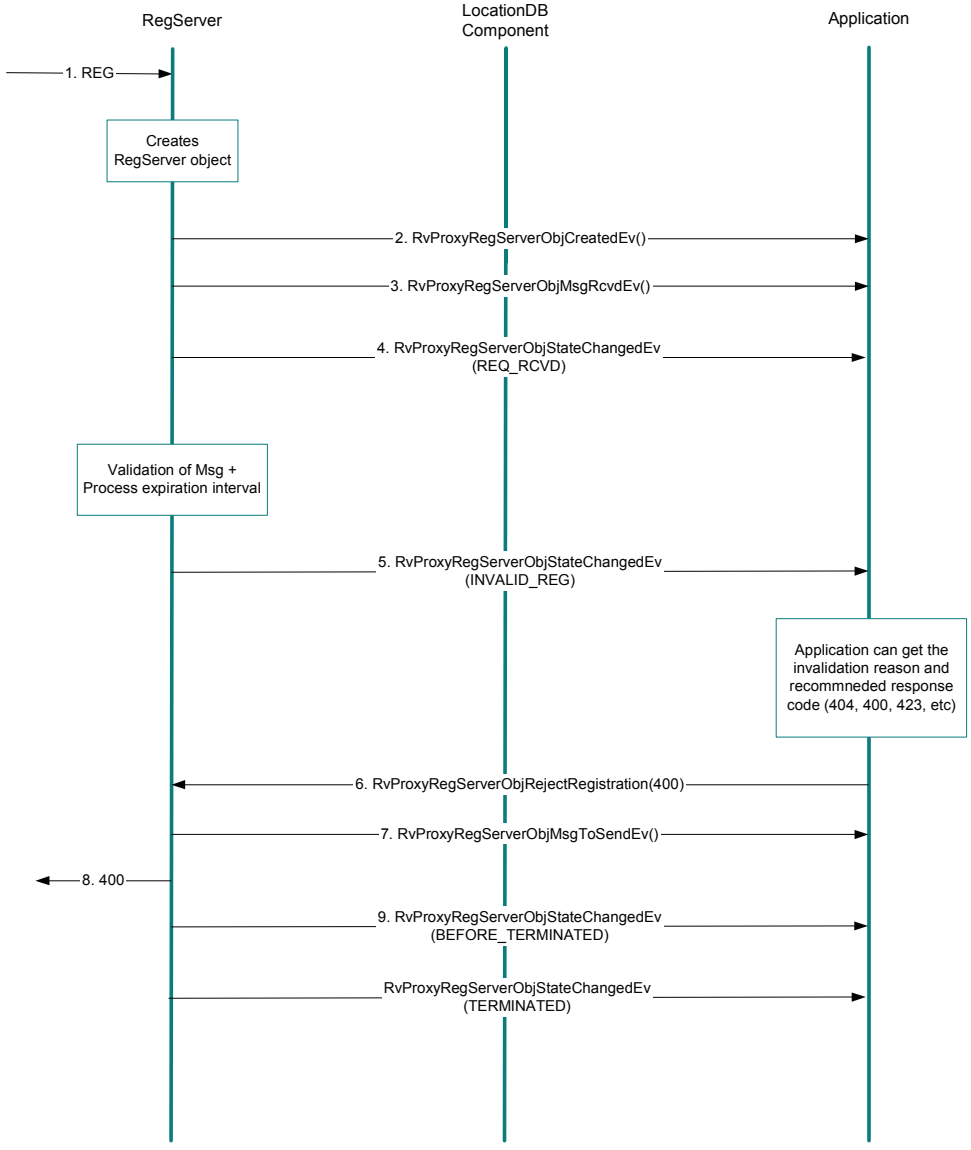


Figure 5-3 *Accepting a Valid Registration*

Accepting a valid registration includes the following stages:

1. A Register request is received by the SIP Stack, and should be handled locally by the Register Server.
2. When the *RegServerMgr* is notified about the local Register request, it creates a new *RegServerObj* which will handle the Register request from this stage. The Register Server module notifies the application about this new object that was created. The application will use this object from this point on to handle the registration.
3. The *RegServerObj* notifies the application about the received message.
4. The *RegServerObj* changes its state to REQ_RCVD.
5. The *RegServerObj* checks if this message is a valid registration request—the message is valid.
6. Since authentication is required, the *RegServerObj* changes its state to AUTHENTICATING.
7. Authentication succeeds and the *RegServerObj* changes its state to VALID-REG.
8. The application decides to accept the Register request.
9. The *RegServerObj* checks the Location Database Server Component if there is an existing record for the registering user. The *RegServerObj* uses the record found in the Location Database and the new record to prepare a record that should be updated in the Location Database.
10. The *RegServerObj* updates the Location Database Server Component with the updated registration information of the user.
11. The Register Server module initiates a 200 response with the updated user registration record. The Register Server module adds contact headers which are the updated user registration entries. The application is notified about the response about to be sent.
12. The 200 response is sent.
13. The *RegServerObj* changes its state to BEFORE_TERMINATED and then terminates.

REJECTING AN INVALID REGISTRATION



Rejecting an invalid registration includes the following stages:

1. A Register request is received by the SIP Stack. The Register request should be handled locally by the Register Server.
2. When the *RegServerMgr* is notified about the local Register request, it creates a new *RegServerObj*, which will handle the Register request from this stage. The Register Server module notifies the application about this new *RegServerObj* that was created. The application will use this object from this point on to handle the registration.
3. The *RegServerObj* notifies the application about the message that was received.
4. The *RegServerObj* changes its state to REQ_RCVD.
5. The *RegServerObj* checks if this message is a valid registration request. The message is invalid, and the *RegServerObj* changes its state to INVALID_REGISTRATION.
6. The application decides to reject the Register request with response code 400.
7. The Register Server module notifies the application that a response is about to be sent.
8. A 400 Response is sent.
9. The *RegServerObj* changes its state to BEFORE_TERMINATED and then terminates.

SAMPLE CODE

The sample code in Figure 5-4 shows how to create and update the Location Database Server Component (Local Database) using a *RegServerRecord*.

INSERTING A RECORD

The application can insert a record to the Location Database Server Component using the Register Server API.

To insert a record in to the Location Database Server Component

1. Create a *RegServerRecord*.
2. Insert the data into the new *RegServerRecord*.
3. Use the Location Database Server Component functions for inserting the *RegServerRecord*.

CREATING A REGISTRATION

The application can create its own registration through the Register Server.

To create a record in the Location Database

1. Create a *RegServerRecord*.
2. Insert the data to the new *RegServerRecord*.
3. Use Location Database Component functions for inserting the *RegServerRecord*.

Sample Code

```
/*=================================================================*/
void RVCALLCONV AppRegServerCreateNewRecord(
                  RvProxyRegServerMgrHandle     hRegServerMgr,
                  RvProxyLocationDBMgrHandle    hLocationDBMgr,
                  RvSipPartyHeaderHandle        hToHeader,
                  RvSipContactHeaderHandle      hContactHeader,
                  RvSipCSeqHeaderHandle         hCSeqHeader)
{
    RvProxyRegServerRecordHandle  hNewRecord  = NULL;
    RvProxyRegServerKeyHandle     hNewKey     = NULL;
    RvProxyRegServerBindingHandle hNewBinding = NULL;
    RvStatus                      rv          = RV_Success;
    RvProxyQueryStatus            eStatus;

    /*step 1*/
    hNewRecord = RvProxyRegServerRecordConstruct(hRegServerMgr,NULL);
    if(hNewRecord == NULL)
    {
          printf("Failed to create new registration record\n");
          return;
    }
    /*step 2*/
    hNewKey = RvProxyRegServerRecordGetKey(hNewRecord);
    if(hNewKey == NULL)
    {
          printf("Failed to get key handle from record\n");
          return;
    }
    rv = RvProxyRegServerKeySetToHeader(hNewKey, hToHeader);
    if(rv != RV_Success)
    {
          printf("Failed to set To header in key\n");
```

```
      return;
}
rv = RvProxyRegServerBindingConstructInRecord(hNewRecord,
                                 RVPROXY_FIRST,
                                 NULL,
                                 &hNewBinding);
if(rv != RV_Success)
 {
      printf("Failed to construct new binding in record\n");
      return;
}
rv = RvProxyRegServerBindingSetContactHeader(hNewBinding,
                                              hNewRecord,
                                              hContactHeader);
if(rv != RV_Success)
{
      printf("Failed to set contact header in new binding\n");
      return;
}


rv = RvProxyRegServerBindingSetCallIdStr(hNewBinding,
                                       hNewRecord,
                                       "callId-str");
if(rv != RV_Success)
{
      printf("Failed to set call-id in new binding\n");
      return;
}


 RvProxyRegServerBindingSetCSeqHeader(hNewBinding, hNewRecord, hCSeqHeader);
if(rv != RV_Success)
{
      printf("Failed to set CSeq header in new binding\n");
      return;
}

/*step 3*/
rv = RvProxyLocationDBImpUpdateRecord(hLocationDBMgr,
                                      hNewRecord,
```

```
                                        hNewKey,
                                        &eStatus);
    if(rv!=rv_Success)
    {
          printf("Failed updating record in Location DB");
    return;
    }
    rv=RvProxyRegServerRecordDestruct(hNewRecord, NULL);


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

Figure 5-4 *Creating and Updating a Registration*

6

WORKING WITH SIP MESSAGES

INTRODUCTION

The RADVISION 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 6-1 illustrates a SIP *message*.

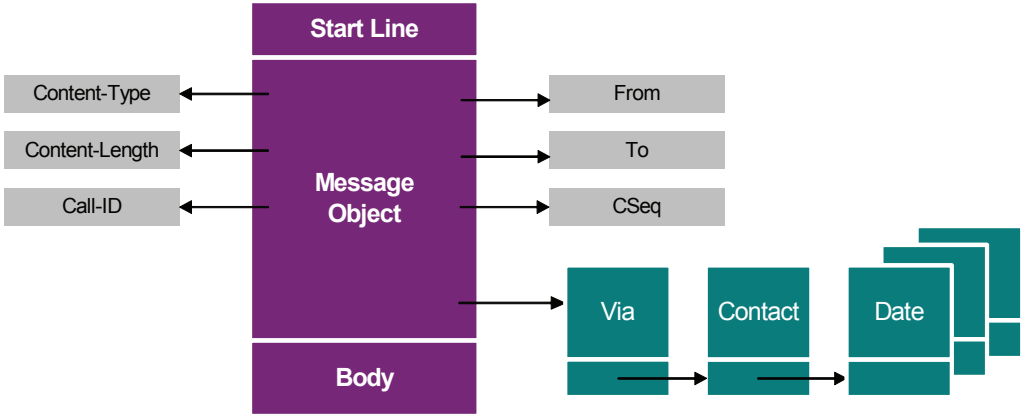


Figure 6-1 *Message Object*

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 6-2 illustrates a To header object. (To and From SIP headers are both kept in a Party header object.)

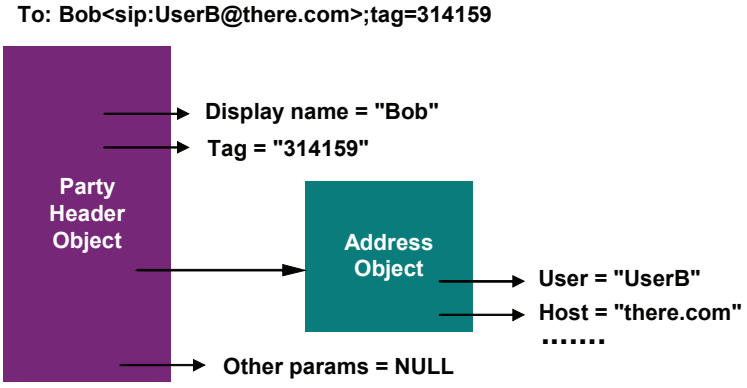


Figure 6-2 *Header Object*

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 6-3 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.

.

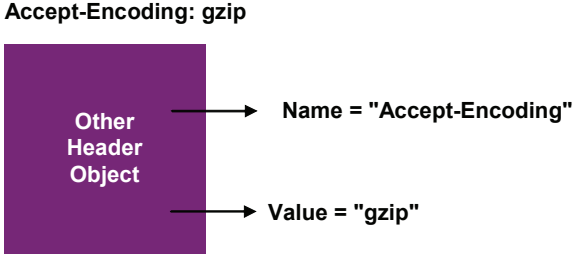


Figure 6-3 *Accept-Encoding SIP Header*

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 6-1 represents the headers that can accept compact form and the API functions that should be used to set compact form to each header.

Table 6-1 *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 6-1, 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.

Memory Management

Message objects and message parts, such as header and address objects, are always placed on RPOOL memory pages. When working with a message object, the message and all its related headers are placed on a single memory page. When working with stand-alone headers, the construction of the header is separated from the page allocation. This enables you to place each header on a different memory page, or several headers on the same memory page. When you free the memory page, all the headers that are placed on this page are also freed and are no longer accessible.

A memory page is a block of memory that is allocated and freed as a single unit. Therefore, when you use a “set” or “parse” operation on a header that is already initialized with data, the SIP Stack does not free the existing data. It allocates a new, empty place on the existing page and sets the new information there. As a result, it is not recommended to perform multiple set or parse operations on a single header, since each such operation allocates more memory. Alternatively, you should use a new page and a new header before each parse operation.

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 6-4 illustrates how the *message* holds a To header object with a syntax error.

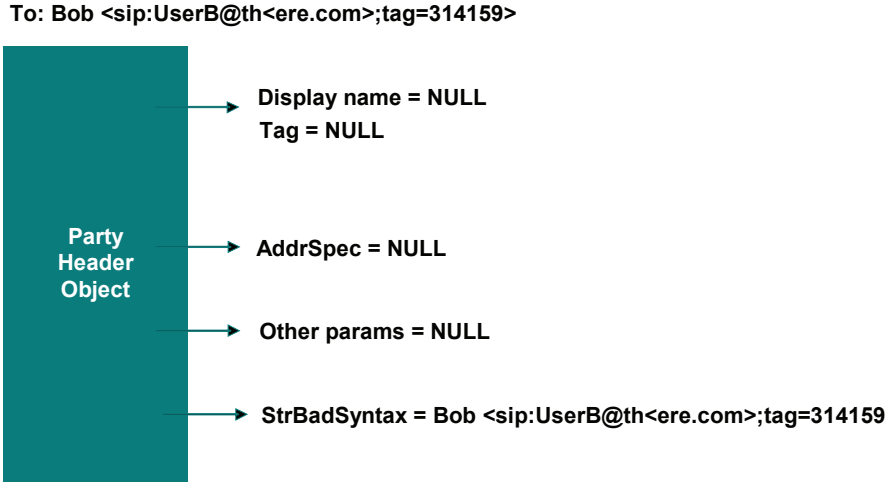


Figure 6-4 *Example of Syntax Error*

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 *RADVISION SIP Toolkit 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 6-5 illustrates how the Stack handles a bad-syntax *message* across layers when the application chooses the "continue processing" option.

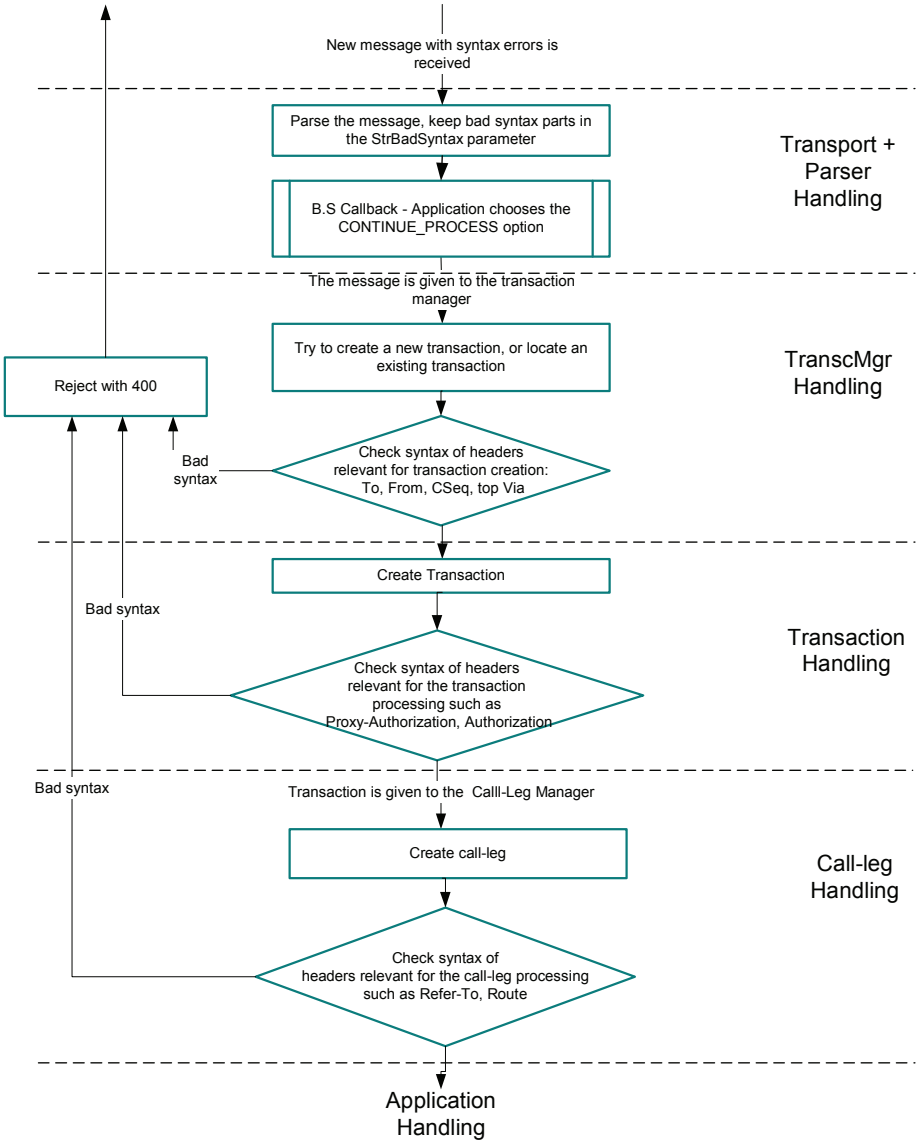


Figure 6-5 *Handling a Bad-syntax Message*

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 or SDP.
- 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 6-6 shows the structure and relationship between the Body and the Body Part objects.

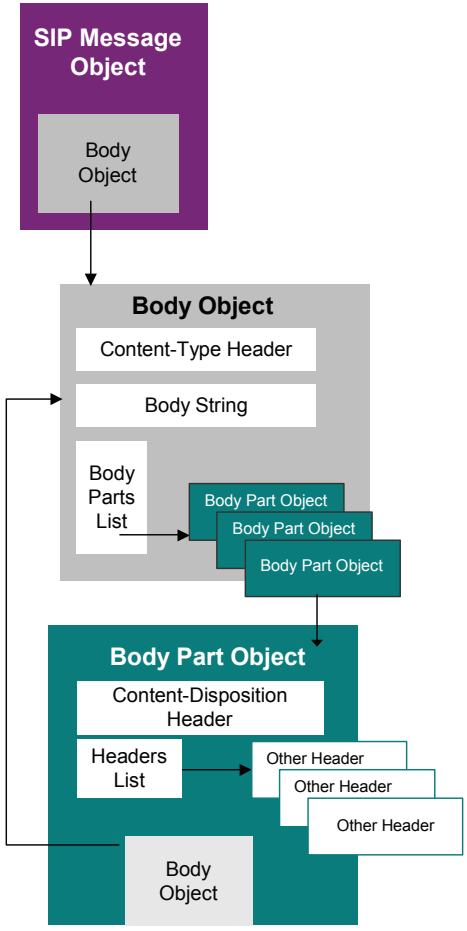


Figure 6-6 *Body and Body Part Objects*

Figure 6-7 is an example of a SIP *message* with a multipart MIME body. Each of the message parts is related to its corresponding object.

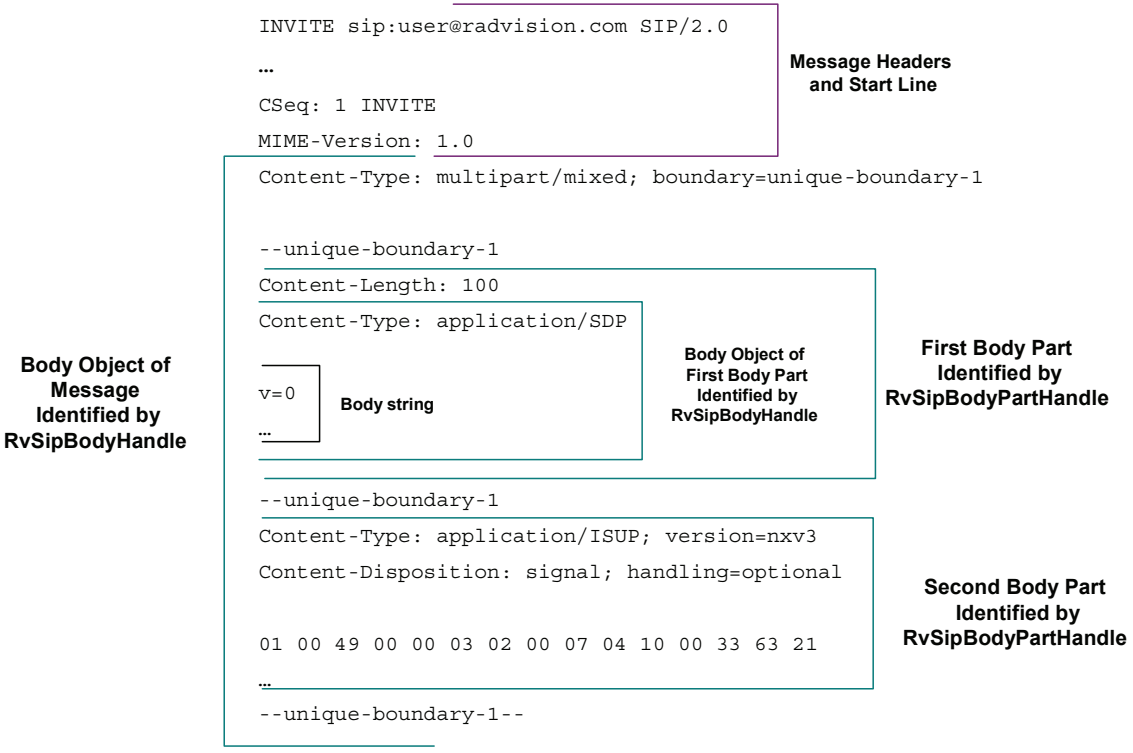


Figure 6-7 *Multipart SIP Message*

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 RADVISION 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 RADVISION 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 RADVISION 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 RADVISION 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 RADVISION 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().

7

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 RADVISION 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.

Note The *transmitter* that the *call-leg* created for sending ACK on 2xx is an exception. Such a *transmitter* is declared persistent only if the *call-leg* itself is persistent.

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 *RADVISION SIP Server Reference Guide*.

RvSipRegClientSetPersistency()

Sets the persistency level of *register-client* objects.

RvSipPubClientSetPersistency()

Sets the persistency level of *publish-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 *RADVISION SIP Server Reference Guide*.

RvSipRegClientGetPersistency()

Gets the persistency level of *register-client* objects.

RvSipPubClientGetPersistency()

Gets the persistency level of *publish-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 *RADVISION SIP Server 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 state.

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 7-1 illustrates the Client Connection state machine.

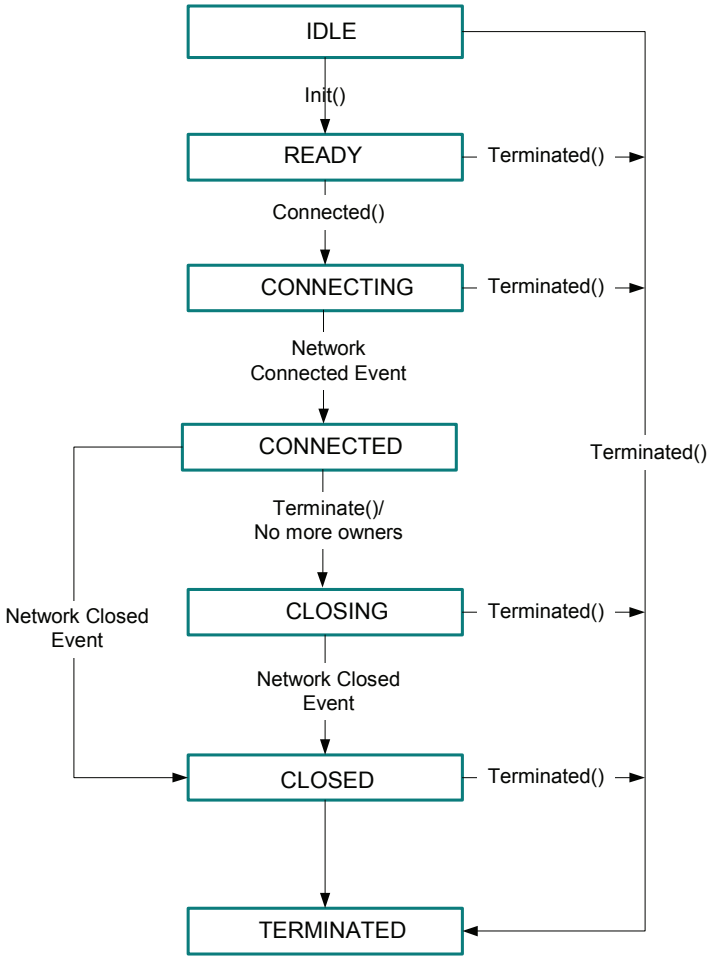


Figure 7-1 *Connection State Machine*

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.

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 7-2 illustrates the Server Connection state machine.

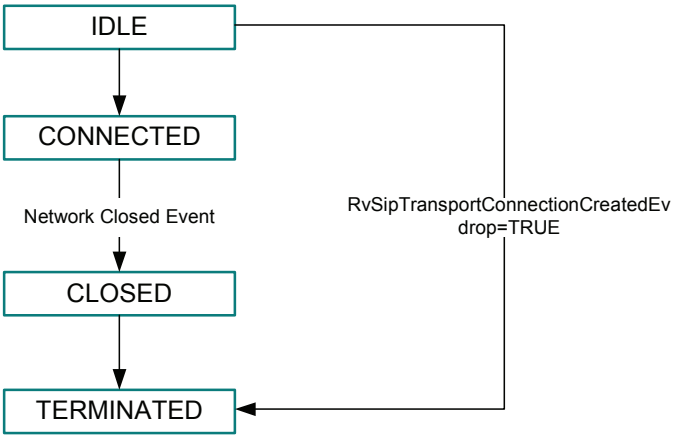


Figure 7-2 *Server Connection State Machine*

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 7-3.

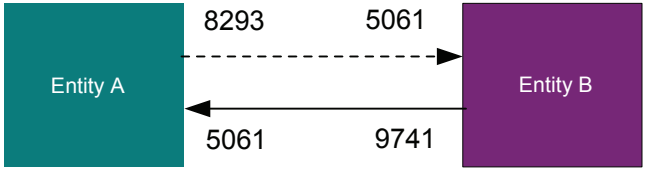


Figure 7-3 *Double Connection between SIP Entities*

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*.

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.)

RADVISION SIP Stack and TLS

The RADVISION 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.

The SIP Stack currently supports TLS transport on Windows, UNIX and Linux platforms. To compile the SIP Stack with TLS, use the RV_TLS_ON compilation flag. (When compiling on UNIX and Linux 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 *RADVISION SIP Toolkit 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 RvSipTransportConnectionTlsHandshake().

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

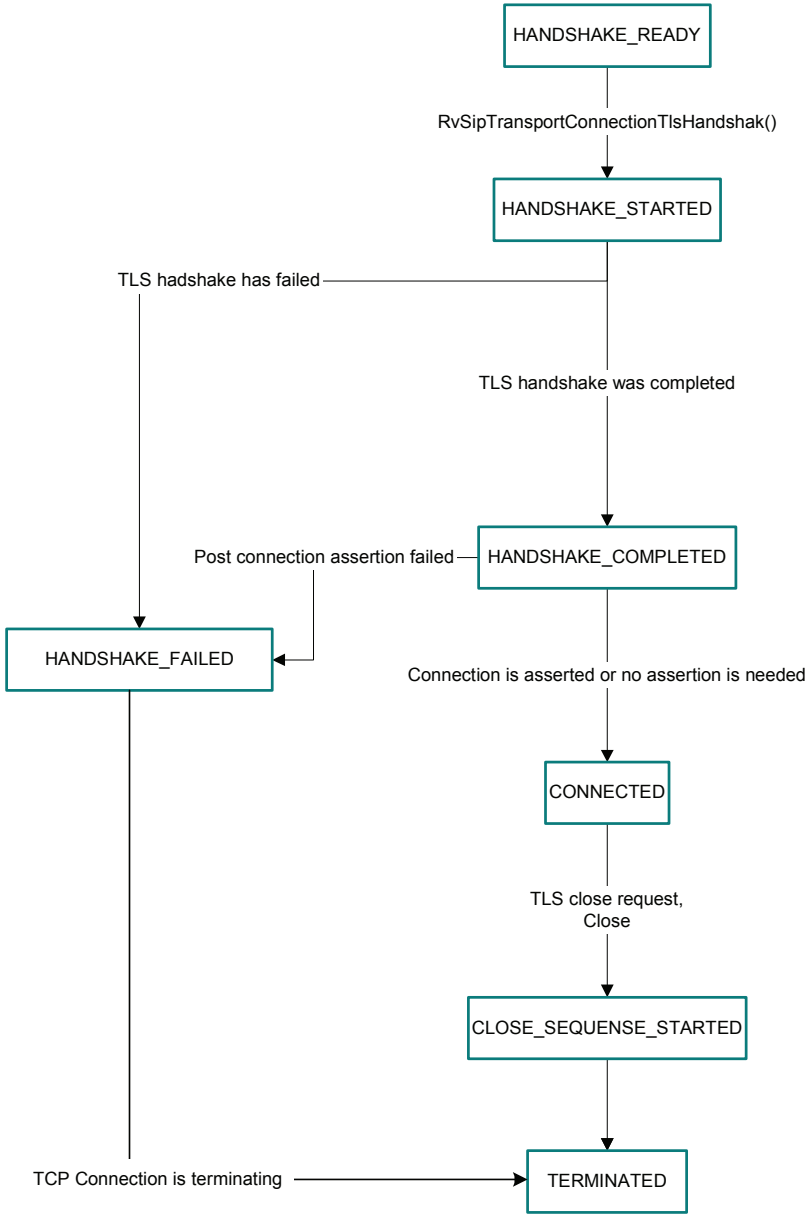


Figure 7-4 *TLS Connection State Machine*

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*.

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 RADVISION 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 RADVISION 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 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.

Dynamic Local Address

The network configuration parameters of the SIP Server Platform include the Domain List, as well as the Local Address, Via and Record-Route parameters. Since server uptime is highly important and the change of listen addresses should be done quicky, Local Address-related information is gathered in a Listen Interface Record (LIR). The LIR is a data structure that contains the local listening address (IP, port and transport) which are mandatory members for all local address manipulations, along with the Via and Record-Route addresses that corresponds to the local listening address.

The SIP Server Platform provides the Dynamic Local Addresses (DLA) feature to support the dynamic change of local addresses and/or domains. Adding and removing listen addresses (Via and Record-Route) and/or domains allows servers (such as SBCs) to run on networks that are dynamically added or

removed after blades are added and removed from the system, or following a server configuration change. The maxNumOfLIR configuration parameter controls the amount of supported LIRs.

DLA API

The API functions for Local Address manipulation are:

- RvSipServerTransportMgrLIRAdd()
- RvSipServerTransportMgrLIRFindByAddress()
- RvSipServerTransportMgrLIRRemove()
- RvSipServerTransportMgrLIRGetFirst()
- RvSipServerTransportMgrLIRGetNext()
- RvSipServerTransportMgrGetNumOfLocalAddress()

The API functions for Domain List manipulations are:

- RvSipServerTransportMgrGetNumOfDomains()
- RvSipServerTransportMgrDomainAdd()
- RvSipServerTransportMgrDomainRemove()
- RvSipServerTransportMgrDomainGetFirst()
- RvSipServerTransportMgrDomainGetNext()

Sample Code

The following sample code illustrates a removal and addition of an LIR.

```
/*============================================================*/
RvSipTransportAddr                              localAddress;
RvSipServerTransportLocalInterfaceRecord        localLIR;
RvSipServerTransportLocalInterfaceRecord*       pFarLIR = NULL;

/* Removing UDP address. */
strcpy(localAddress.strIP, "172.16.70.102");
localAddress.port = 5060;
localAddress.eTransportType = RVSIP_TRANSPORT_UDP;
localAddress.eAddrType = RVSIP_TRANSPORT_ADDRESS_TYPE_IP;
RvSipServerTransportMgrLIRFindByAddress(hTransportMgr,&localAddres
s ,&pFarLIR);
RvSipServerTransportMgrLIRRemove(hTransportMgr,  pFarLIR);

/* Adding listen TCP address.  */
strcpy(localLIR.localAddress.strIP, "172.16.70.102");
localLIR.localAddress.port = 5060;
localLIR.localAddress.eTransportType = RVSIP_TRANSPORT_TCP;
localLIR.localAddress.eAddrType = RVSIP_TRANSPORT_ADDRESS_TYPE_IP;
```

```
/* Set Via. */
RvSipViaHeaderSetHost(localLIR.hVia,"172.16.70.102");
RvSipViaHeaderSetPortNum(localLIR.hVia, 5060);
/*-----------------------------------------------------------*/
```

8

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 RADVISION 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, “radvision.com” is the suffix for the “host1.radvision.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.

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.

Note When working in Windows, the SIP Stack tries to fetch the DNS servers and DNS suffixes from the Windows configuration. To verify that your system is configured correctly, look at the output of the “`ipconfig/all`” command, and make sure that the “DNS Suffix” item is configured both in the “Windows IP Configuration” section and in the “Adapter Connection Specific” section. The “DNS Servers” should also be configured.

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 8-1. 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).

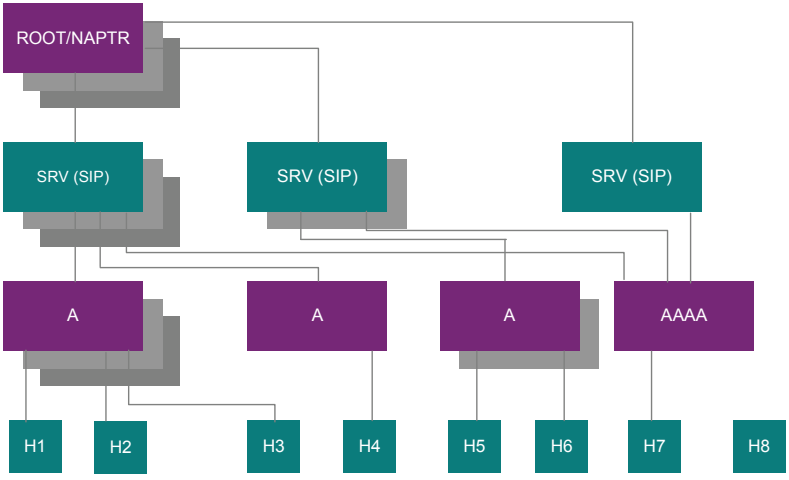


Figure 8-1 *A DNS/SRV Spanned Tree*

DNS/ENUM Record

RFC 3824 specifies the usage of ENUM NAPTR records for relating between E.164 numbers and URIs.

RADVISION 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 8-2 illustrates a DNS List object.

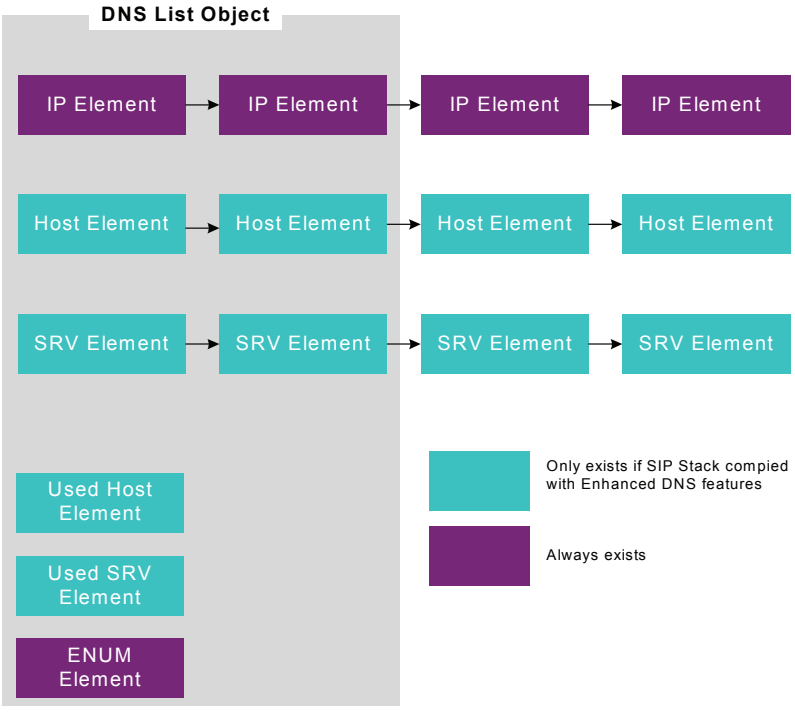


Figure 8-2 *DNS List Object*

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, Register-Clients, and Publish-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 *RADVISION SIP Toolkit 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 *RADVISION SIP Toolkit 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.

Publish-Client Layer

The RVSIP_PUB_CLIENT_STATE_MSG_SEND_FAILURE state indicates a message send failure. The application can obtain the DNS List object handle by calling the RvSipPubClientDNSGetList() function. Once the application has a handle to the list, it can manipulate the list using the Transport API.

Publish-Client DNS API

Calling the RvSipPubClientDNSContinue() function causes the SIP Stack to prepare an internal cloned transaction and terminate the original transaction. If the application wishes, it can call RvSipPubClientGetOutboundMsg() and manipulate the message before sending it. To actually try sending the message to the next address in the DNS list, the application should call the RvSipPubClientDNSReSendRequest() function. Calling RvSipPubClientDNSGiveUp() indicates that the application does not want to keep trying to send the message, and will move the *publish-client* object to the FAILED state.

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 asynchronously:

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.

DNS Caching

The RADVISION SIP Toolkit 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.

9

LOCATION DATABASE SERVER COMPONENT

INTRODUCTION

The Location Database Server Component is an application module that supplies storage services for registration information. The Location Database can store registration information for a specific user, and update this data upon request. It can also search for registration information for a specific user and return all stored information, or remove a user from the database.

The Location Database module is a general implementation for a location database, suitable for different applications and environments. The SIP Server defines an interface for the Location Database, which includes a set of functions which should be implemented by the Location Database. The SIP Server will use this set of functions for storing registration information when handling Register requests, and for address resolution.

The Location Database module can be replaced with any location database application that will implement the interface defined for the Location Database, while keeping the functionality required for each function.

LOCATION DATABASE INTERFACE

The following functions are the interface between the SIP Server and the Location Database Server Component. These functions are used for handling registration requests and for address resolution.

All database actions are atomic. If an action fails, the record in the database will remain unchanged.

RvProxyLocationDBImpUpdateRecord()

Receives an updated record as it should be saved in the Location Database. The updated record should be saved as is in the Location Database, and should replace an old record for this key if such exists.

RvProxyLocationDBImpInsertRecord()

Adds new data to the Location Database without any data processing. If a record for this key already exists, the new bindings should be added as is to its existing binding list. If no such record exists, a new record should be added.

RvProxyLocationDBImpLookupRecord()

Performs a lookup on the Location Database and searches for an existing record identified by the key given as an argument to the function. This function returns the record found with valid bindings only, after removing all expired bindings. If no valid bindings are found for this record, the RV_NotFound value should be returned and the binding list of the record should be empty.

RvProxyLocationDBImpRemoveRecord()

Removes a record from the Location Database Registration Table and de-allocates it. The key and binding list of the record should be removed as well.

Replacing the Location Database Implementation

The Location Database Server Component is designed so it can be replaced with any implementation of a Location Database that will work with the SIP Server.

Creating a Location Database implementation includes the following steps:

1. Implementing the Location Database interface functions while using the prototypes as defined in the *RvProxyRegServerTypes.h* file.

   The implementation of the interface functions should return the expected results.
2. Setting the RvProxyLocationDBInterface pointers to the interface functions implemented in step 1, as described in the sample code in Figure 9-1.
3. Calling RvSipServerMgrLocationDBSetInterface() with the structure of pointers that are defined. This will cause the SIP Server to call the interface functions defined for using the Location Database Server Component.

4. Working with Register Server handles:

   All Location Database interface functions expect arguments as Register Server handles. These handles contain all the registration information given by the Register Server. The Location Database implementation should be able to supply its data as Register Server handles. This way the Register Server can use the interface functions for saving registrations.

   The Location Database implementation may use any scheme for managing its data, but must supply the Register Server handles with all data needed by the Register Server module.

5. The Location Database Server Component is responsible for removing information that has expired. The implementation should keep track of the expiration times for the bindings and remove any expired data.

Sample Code

The sample code in Figure 9-1 demonstrates how to set the Location Database Implementation interface functions as the Location Database interface functions:

To set the Location Database Implementation interface functions

1. Declare an RvProxyLocationDBInterface structure.
2. Initialize the structure members to NULL.
3. Set your interface functions in the RvProxyLocationDBInterface structure.
4. Call RvSipServerMgrLocationDBSetInterface().

```
/*=============================================================*/
void SetAPIFunctions()
{
    /*step 1*/
    RvProxyLocationDBInterface hLDBFuncHandler;
    /*step 2*/
    memset(&hFuncHandler, 0, sizeof(hLDBFuncHandler));
    /*step 3*/
    hLDBFuncHandler.pfnLocationDBUpdate =
                    AppLocationDBUpdate;
    /*step 4*/
    RvProxyRegServerMgrLocationDBSetInterface(
```

```
                                        hSipServerMgr,
                                        &hLDBFuncHandler,
                                        sizeof(hLDBFuncHandler));
}
/*============================================================*/
```

Figure 9-1 *Setting implementation API Functions*

WORK FLOW DIAGRAM

Figure 9-2 illustrates how to update a registration request.

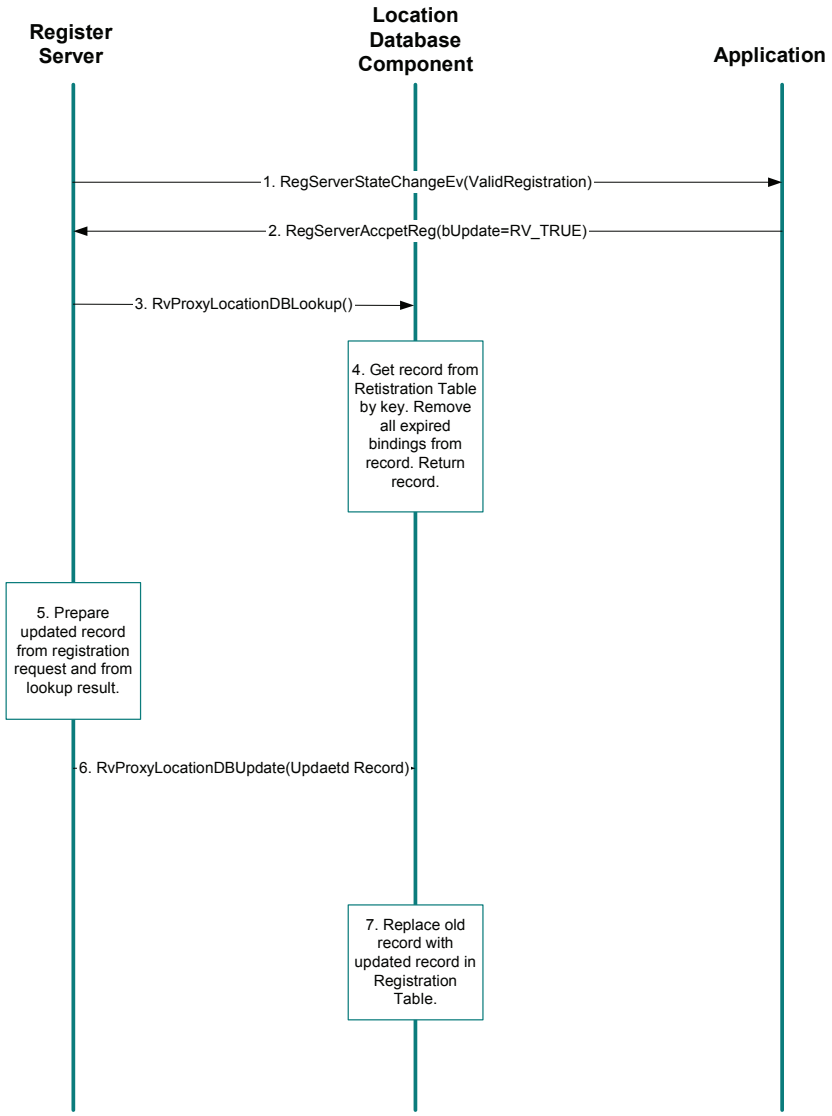


Figure 9-2 *Updating a Registration Request*

Updating a registration request includes the following stages:

1. The Register Server notifies the application about a new valid registration message.
2. The application chooses to accept the registration request and update its data in the Location Database.
3. The Register Server calls the Location Database lookup interface function to check if a record for the registering user already exists.
4. The Location Database checks its database for an existing record for the key given as an argument. If a record is found, it is checked for expired bindings. If such are found, they are removed from the binding list of the record. The record is returned to the Register Server.
5. The Register Server prepares an updated record from the new registration message and the record found by the Location Database, according to the SIP RFC.
6. The Register Server calls the Location Database update API function with the updated record.
7. The Location Database checks if a record for this user already exists in the database. If one is found, it is removed.

   The new record is added to the Location Database Server Component database.

Location Database Server Component API

The SIP Server provides a general implementation of the Location Database Server Component. This implementation is based on a hash table. Each record in the hash table is identified by the key. The key is made from the username and host of the To header of the Register. The following Location Database API functions are provided by the SIP Server and enable using the SIP Server implementation of a Location Database.

All interface functions are implemented by the SIP Server as well.

RvProxyLocationDBImpConstruct()

Constructs the Location Database Server Component. Creates and initializes the Location Database Manager object (*LocationDBMgr)*, and allocates memory. It is the responsibility of the application to construct the Location Database Server Component and set the *LocationDBMgr* handle to the SIP Server Manager

(*SIPServerMgr*) with the RvSipServerMgrSetLocationDBMgr() API function. The application should construct the Location Database Server Component upon initialization.

RvProxyLocationDBImpDestruct()

Destructs the Location Database Server Component. It is the responsibility of the application to destruct the Location Database Server Component when exiting.

RvProxyLocationDBImpGetResourceStatus()

Returns the current status of resources used by the Location Database Server Component.

Sample Code

The sample code in Figure 9-3 demonstrates initializing the Location Database Server Component.

To initialize the Location Database Server Component

1. Declare an RvProxyLocationDBCfg structure.
2. Set the configuration values in this structure.
3. Call RvProxyLocationDBImpConstruct().
4. Call RvProxyRegServerMgrSetLocationDBMgrHandle() with the handle returned from step 3.

```
/*============================================================*/
void InitializeLocationDB(hSipServerMgr)
{
    /*step 1*/
    RvProxyLocationDBCfg       hLDBCfg;
    RvProxyLocationDBMgrHandle hLDBMgr = NULL;
    /*step 2*/
    hLDBCfg.maxRegisteredNum  = 20;
    hLDBCfg.maxContactHeaders = 40;
    hLDBCfg.numPages          = 50;
    hLDBCfg.pageSize          = 512;
    /*step 3*/
    RvProxyLocationDBImpConstruct(hSipServerMgr,
                                  &hLDBCfg,
                                  sizeof(hLDBCfg),
```

```
                                        &hLDBMgr);
    if(hLDBMgr == NULL)
    {
            printf("Error in constructing LocationDB Component\n";
            return;
    }
    /*step 4*/
    RvProxyRegServerMgrSetLocationDBMgrHandle(hSipServerMgr,
                                    HLDBMgr);

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

Figure 9-3 *Initializing the Location Database Server Component*

CONFIGURATION PARAMETERS

The configuration parameters of the Location Database Server Component are as follows:

maxRegisteredNum

The maximum number of registered users that the Location Database Server Component can handle simultaneously.

Default value: RVPROXY_LOCATIONDB_DEFAULT_MAX_REGISTER

maxContactHeaders

The maximum number of bindings for all users in the Location Database Server Component.

Default value: RVPROXY_LOCATIONDB_DEFAULT_MAX_CONTACTS

numPages

The maximum number of pages used by the Location Database Server Component. These pages are used to store the binding data.

Default value: RVPROXY_LOCATIONDB_DEFAULT_NUM_PAGES

pageSize

Defines the size of each allocated page.

Default value: RVPROXY_LOCATIONDB_DEFAULT_PAGE_SIZE

numLookupPages

The maximum number of pages used by the Location Database Server Component for lookup purposes.

Default value:
RVPROXY_LOCATIONDB_DEFAULT_NUM_LOOKUP_PAGES

lookupPageSize

Defines the size of each lookup allocated page.

Default value:
RVPROXY_LOCATIONDB_DEFAULT_LOOKUP_PAGE_SIZE

logFilters

Defines the log filter that will be used by the Location Database Server Component.

Default value: RV_SIP_SERVER_LOG_DEFAULT

10

SECURITY SERVER COMPONENT

INTRODUCTION

The SIP Server Security Component is an application module layer that supplies Server Authentication services for the SIP Server. This component implements an authentication mechanism and hashing algorithms. The SIP Server Security Component can support Digest/MD5 and/or any other authentication mechanism and algorithm. If the Security Component chooses to implement Digest/MD5, the SIP Server will make the authentication while consulting the Security Component about required information for the authentication process. Otherwise, the Security Component needs to authenticate the Authorization header stored in the *ServerAuth* and report the authentication results to the Server Authentication module.

The Server Authentication module interacts with the Security Component in the following ways:

- **Server Authentication Object (*ServerAuth*)**

  The Server Authentication module creates this object, which is responsible for informing the Security Component about the SIP Server requirements, in order to continue with the request authentication process. The *ServerAuth* is used in any authentication scheme and algorithm. The Security Component must implement the *ServerAuth* events in order to be able to authenticate a request.

- **SIP Server Security Component Interface**

  The SIP Server Security Interface includes the events of the *ServerAuth* and additional functions related to Digest/MD5 authentication. The Digest Authentication method implemented by the Server Authentication module 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. If the application wishes to use this SIP Server ability, it needs to implement these interface functions. When the Server Authentication module encounters an Authorization header with a Digest scheme and MD5 algorithm, it will use these interface functions to ask for authentication services, such as checking the validity of Realm and Nonce values, returning the MD5 credential, and making MD5 hashing.

The SIP Server supplies a general implementation of the SIP Server Security Component that supports a Digest scheme and MD5 algorithm. The application can use this implementation or replace it with any other implementation while keeping the required functionality as described below.

Server Authentication Object

The *ServerAuth* interacts between the Security Component and the Server Authentication module while processing the authentication of an incoming request.

Server Authentication Object API

The Server Authentication API contains a set of functions and function callbacks that allow the application to set or examine Server Authentication parameters and control them.

Server Authentication Object Parameters

The application can set or examine Server Authentication parameters via Server Authentication Set and Get API functions. The following parameters are available:

State

Represents the state of the *ServerAuth* in its state machine. This parameter is not modifiable.

Request Message

Represents a handle to a request message that the *ServerAuth* is authenticating. This parameter is not modifiable.

Security Manager

Represents the Security Component Manager (*SecurityMgr*) handle. This handle is set to the Server Authentication module using RvSipServerAuthMgrSetSecurityMgrHandle() at the initialization of the Security Component.

Authentication Manager

Represents the Server Authentication Manager (*ServerAuthMgr*) handle.

Authorization Header

A single SIP request may contain several Authorization headers for different SIP Servers and UAs as defined in the RFC. This parameter is the Authorization header that is currently being authenticated by the SIP Server.

Support Stale

Indicates whether the *ServerAuth* supports the stale parameter on WWW-Authentication and Proxy-Authentication headers. If the *ServerAuth* supports the stale parameters, the *ServerAuth* will look for the specified user of the currently processed Authorization header, even if the nonce does not match the *SecurityMgr* nonce.

Server Authentication Object Control

The following API functions provide *ServerAuth* control:

RvSipServerAuthObjSetPwd()

Supplies a password for the specified user from Authorization header.

The application can use this function only in the RVSIPSERVER_AUTH_STATE_AWAITING_PWD state.

RvSipServerAuthObjSetAuthResult()

Reports the authentication result of the Authorization header for the Server Authentication module. For example:

- The authentication process has succeeded.
- The authentication process has failed.
- The current authorization header is not intended for this server.
- The application wishes to stop the authentication process (authentication will fail).

The application can use this function only in the RVSIPSERVER_AUTH_STATE_AUTH_REQUEST State.

RvSipServerAuthObjSetAuthenticationHeader()

Supplies the Server Authentication module with the Authentication or Proxy-Authentication header to send with the 401/407-response message as a result of authentication failure.

The application can use this function only in the RVSIPSERVER_AUTH_STATE_AWAITING_AUTH_HEADER state.

SERVER AUTHENTICATION STATE MACHINE

The Server Authentication state machine represents the state of a *ServerAuth* while performing authentication. The Security Component receives notification of a *ServerAuth* state change through the RvSipServerAuthObjStateChangeEv() event which will be discussed later in Security Component Interface on page 255. The states are divided into three groups:

- **Common**—common states for Digest/MD5 and other authentication mechanisms.
- **Supporting Digest/MD5**—relevant when authenticating the Digest/MD5 header and the Security Component supports the Digest/MD5 algorithm. If the Security Component does not support the Digest/MD5 algorithm, it does not need to implement and react to these states.
- **Other authentication schemes and algorithms**—relevant when the authenticating other schemes and algorithms (not Digest/MD5) or if the Security Component does not support the Digest/MD5 algorithm.

COMMON STATES

The common states are:

RVSIPSERVER_AUTH_STATE _IDLE

The initial state of the Server Authentication state machine. Upon creation of a *ServerAuth*, the *ServerAuth* assumes this state. It remains in this state until the authorization header is found on the request message.

RVSIPSERVER_AUTH_STATE_NEW_AUTH_HEADER

When *ServerAuth* finds a new Authorization header on the request message it assumes this state. This is an informative state and the Security Component needs to do nothing in this state.

RVSIPSERVER_AUTH_STATE_AUTH_TERMINATE

The final state of the *ServerAuth*. Upon reaching this state, the application can no longer refer to the *ServerAuth*.

DIGEST/MD5 STATES

The Digest/MD5 states are:

RVSIPSERVER_AUTH_STATE_CHECKING_REALM_AND_NONCE

When *ServerAuth* finds an Authorization header which is using the Digest scheme with the MD5 algorithm, it assumes this state. In this state, the *ServerAuth* asks the Security Component whether the Realm and Nonce are valid using RvSipServerSecurityAuthCheckRealmAndNonce(). This is an informative state and the Security Component needs to do nothing in this state.

RVSIPSERVER_AUTH_STATE_AWAITING_PWD

If the Realm and Nonce are valid, the *ServerAuth* assumes this state. In this state, the Security Component needs to supply the user password from the Authorization header using RvSipServerAuthObjSetPwd().

OTHER STATES

Other authentication mechanism states are:

RVSIPSERVER_AUTH_STATE_AUTH_REQUEST

If the currently authenticated Authorization header is not using the Digest scheme with the MD5 algorithm, the *ServerAuth* assumes this state. In this state, the Security Component needs to authenticate the request and report the authentication results to the *ServerAuth* using RvSipServerAuthObjSetAuthResult().

RVPSIPSERVER_AUTH_STATE_AWAITING_AUTH_HEADER

If the request authentication failed, the application must decide whether to reject the request with 401/407 response code or to ignore this failure. If the application decides to reject it using the request with a 401/407 response code, and the default Authentication algorithm is not RVSIP_AUTH_ALGORITHM_MD5, or the authentication has failed on a header that does not contain the MD5 algorithm, the *ServerAuth* assumes this state. In this state, the Security Component needs to supply an Authentication header or a Proxy-Authentication header to be sent with the 401/407 response message using RvSipServerAuthObjSetAuthenticationHeader().

STATE MACHINE

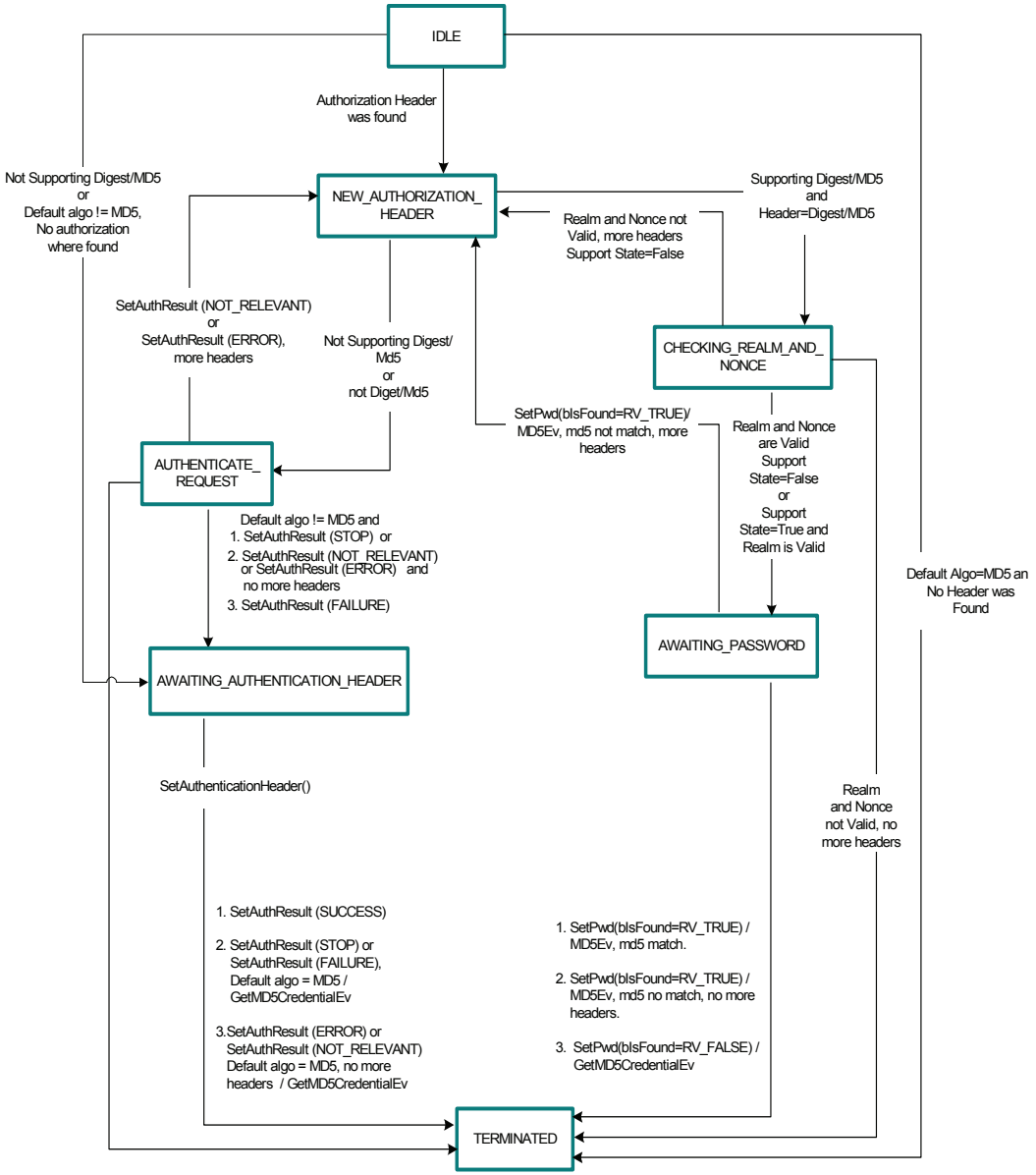


Figure 10-1 *Server Authentication State Machine*

Security Component Interface

The following API functions are the interface between the Server Authentication module and the Security Component. In order to receive to the events, the Security Component needs to pass an event handler pointer to the *ServerAuthMgr*. When an event occurs, the *ServerAuth* calls the event handler function using this pointer.

RvSipServerAuthObjCreated()

Notifies the Security Component that a new *ServerAuth* has been created. The newly created *ServerAuth* always assumes the RVSIPSERVER_AUTH_STATE_IDLE state. The application can create an object to associate with this *ServerAuth*. The handle for the application object will be supplied in any callback function related to this *ServerAuth*.

RvSipServerAuthObjStateChange()

This event is the most useful of the events that the *ServerAuth* reports. Through this function, the Security Component receives notification of a *ServerAuth* state change. The Security Component needs to act upon the new state. For example:

- Upon receipt of a RVSIPSERVER_AUTH_STATE_AWAITING_PWD state notification, the Security Component needs to supply a password related to the specify user from in the Authorization header using RvSipServerAuthObjSetPwd().
- Upon receipt of a SIPSERVER_AUTH_STATE_AUTH_REQUEST state notification, the Security Component needs to authenticate the request using the Authorization header and to report its results to the *ServerAuth* using RvSipServerAuthObjSetResault().

RvSipSecurityAuthCheckRealmAndNonce()

Deprecated.

RvSipServerAuthObjCheckRealmAndNonce

Notifies the component that the *ServerAuth* needs to know whether the realm and nonce are known and valid. If the realm is valid and the nonce is not valid, and the component supports the authentication “stale” parameter, it is recommended that the component call RvSipServerAuthObjSupportStale() to notify the *ServerAuth* that it supports this parameter.

If the Security Component does not wish to support Digest/MD5, it does not need to implement this event.

RvSipSecurityAuthGetMD5Credential()

Notifies the Security Component that it needs to return its MD5 credential, such as Nonce, Qop, Opaqe and URI. These credentials will be used to construct the Authentication or Proxy-Authentication header that will be sent along with the 401/407 response message. If the Security Component does not wish to support Digest/MD5, it does not need to implement this event.

RvSipSecurityAuthMD5()

Deprecated.

RvSipSecurityAuthMD5Ex()

Notifies the Security Component that it needs to perform MD5 hashing on the IN parameter and return the result in the OUT parameter. If the Security Component does not wish to support Digest/MD5, it does not need to implement this event.

RvSipServerSecurityAuthMD5EntityBody()

Notifies the Security Component 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”.

RvSipServerSecurityAuthClientSharedSecret()

Notifies the Security Component that it needs to supply the username and password of the server for client-side authentication.

Work Flow Diagram

Figure 10-2 diagram illustrates the authenticating Digest/MD5 Authorization header, when the Security Component supports Digest/MD5.

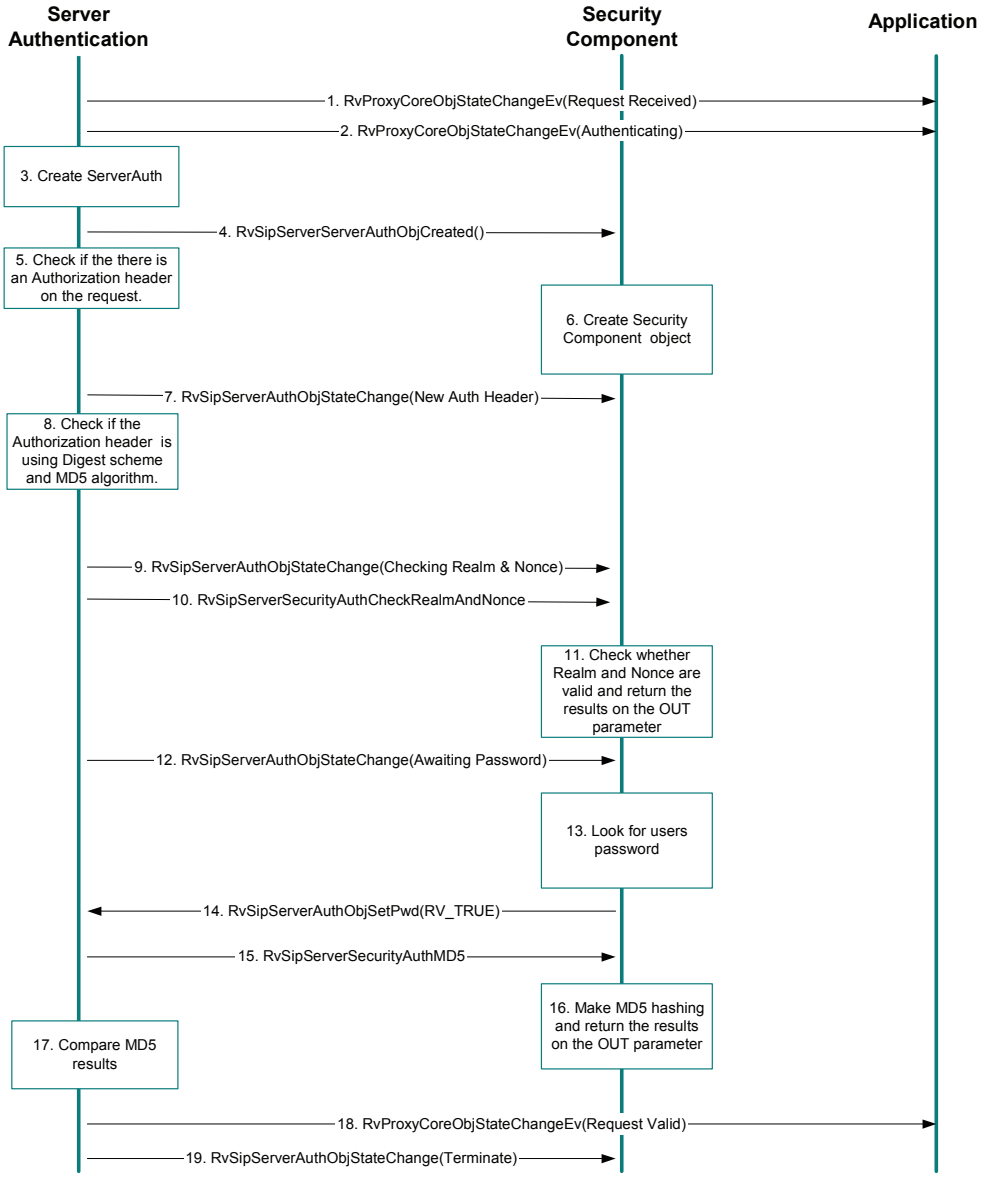


Figure 10-2 *Authenticating Digest/MD5 Authorization header with Security Component supporting Digest/MD5*

Authenticating the Digest/MD5 Authorization header, when Security Component supports Digest/MD5 includes the following stages:

1. A request is received, the SIP Server module informs the application that a request is received.
2. The Server Authentication module starts the request authentication process and informs the application about it.
3. The Server Authentication module creates a *ServerAuth* for the authentication process.
4. The Security Component is informed about the *ServerAuth* creation.
5. The Security Component creates a related object to the *ServerAuth* to associate it with the newly created *ServerAuth*.
6. The Server Authentication module checks if the request contains an Authorization header.
7. After an Authorization header has been found, the Server Authentication module informs the Security Component about this header.
8. The Server Authentication module checks whether the new authorization header is using the Digest scheme and MD5 algorithm for authentication.
9. After the Authorization header has been established as using the Digest/MD5 authentication mechanism, the Server Authentication module informs the Security Component that it is about to verify the Realm and Nonce values.
10. The Server Authentication module asks the Security Component to verify the Realm and Nonce values.
11. The Security Component checks whether the Realm and Nonce values are valid and returns the results to the Server Authentication module in the OUT parameter.
12. After the Security Component has approved that the Realm and Nonce values are valid, the Server Authentication module asks the Security Component to search for the password of the specified user from the Authorization header.
13. The Security Component performs a search for the specified user’s password.
14. When the Security Component finds the password, it reports to the Server Authentication module that the specified user's

password has been found and sets the password to the *ServerAuth*.

15. The Server Authentication module asks for the MD5 hashing service from the Security Component.
16. The Security Component performs MD5 hashing and returns the results in the OUT parameter.
17. The Server Authentication module compares the hashing results with the received request.
18. The Server Authentication module informs the application that the request is valid.
19. The Server Authentication module informs the Security Component that it about to terminate the *ServerAuth*.

Creating an Application Security Component

The Security Component is designed so it can be replaced with other implementations and will continue to work with the SIP Server.

To create an application Security Component

1. Implement the Security Component Interface including *ServerAuth* events. Use the prototype as defined in the *RvSipServerAuthTypes.h* file.
2. Create an *RvSipServerSecurityInterface* object and set the interface implementation for the Security Component to this object. Call RvSipServerAuthMgrSecuritySetInterface() with a pointer to the created object. This causes the SIP Server to call the application implementation of the Security Component.
3. Create an application *SecurityMgr*, and call RvSipServerAuthMgrSetSecurityMgrHandle() with the created object to attach the *SecurityMgr* to the Server Authentication module.

Sample Code

The sample code in Figure 10-3 demonstrates the basic phases that the application needs to go through if it wishes to replace the SIP Server Security Component implementation.

To replace the implementation

1. Declare an RvSipServerSecurityInterface structure.
2. Initialize the structure members to NULL.
3. Assign the implementation of the Security Component interface to RvSipServerSecurityInterface structure.
4. Call RvSipServerAuthMgrSecuritySetInterface() to assign the implementation to the Server Authentication module.

```
/*=============================================================*/
void SetAPIFunctions()
{
 /*step 1*/
 RvSipServerSecurityInterface securityInterface;
 /*step 2*/
 memset(&securityInterface, 0,
         sizeof(securityInterface));
 /*step 3*/
 securityInterface.pfnSecurityServerAuthObjCreated =
          AppSrvAuthObjCreated;
 /* Assigns the rest of the security implementation here. */
 .
 .
 .
 /*step 4*/
 RvSipServerAuthMgrSecuritySetInterface(hServerAuthMgr,
                                        &securityInterface,
                                        sizeof(securityInterface));
}
/*=============================================================*/
```

Figure 10-3 *Setting Interface Functions*

Sample Code

The sample code in Figure 10-4 illustrates the implementation of the RvSipServerAuthObjCreated() event.

To implement the event

1. Create an application object to associate with the newly *ServerAuth*.
2. Initialize the application object.
3. Exchange handles with the SIP Server.

```
/*-----------------------------------------------------------------*/
void RVCALLCONV AppSecurityComponentCreated(
              RvSipServerAuthObjHandle      hSrvAuthObj,
              RvSipServerAuthObjAppHandle*  phAppObj)
{
    AppSecurityObj *pAppSecurityObj = NULL;

   /*1. Creates application Security object. */
    AppSecurityObj =
           (AppSecurityObj*)malloc(sizeof(AppSecurityObj));
    If(NULL == pAppSecurityObj)
    {
          printf("Failed to allocate application security
             component\n");
             return;
    }
   /*2. Initializes the application object. */
    AppInitSecurityObj(pAppSecurityObj, hSrvAuthObj);

   /*3. Exchanges handle with the SIP Server. */
    *phAppObj = pAppSecurityObj;
     pAppSecurityObj->srvAuthObj = hSrvAuthObj;

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

Figure 10-4 *Implementing the ServerAuth Creation Event.*

Sample Code

The sample code in Figure 10-5 illustrates how to implement the RvSipServerAuthObjStateChange() event.

```
/*-----------------------------------------------------------------*/
void RVCALLCONV AppSecurityComponentStateChange(
                RvSipServerAuthObjHandle     hSrvAuthObj,
```

```
                RvSipServerAuthObjAppHandle  hAppObj,
                RvSipServerAuthObjState      eState)
{
    RvBool bIsPasswordFound = RV_FALSE;
    RvChar strPassword[APP_MAX_PWD_LEN];

    switch(eState)
           {
            case RVSIPSERVER_AUTH_STATE_NEW_AUTH_HEADER:
              printf("starting to authorize new Authorization
                     Header\n");
              break;

            case RVSIPSERVER_AUTH_STATE_AWAITING_PWD:
                   bIsPasswordFound = AppLookUpPwd(hAppObj,
                         strPassword);
                   if(RV_TRUE == bIsPasswordFound)
                   {
                      RvSipServerAuthObjSetPwd(hSrvAuthObj,
                                               strPassword,
                                               RV_TRUE);
                    }
                    else
                    {
                      RvSipServerAuthObjSetPwd(hSrvAuthObj,
                                               NULL,
                                               RV_FALSE);
                     }
               break;
            case RVSIPSERVER_AUTH_STATE_AUTH_REQUEST:
             /* Does not support any other then Digest/MD5;
             proceeds to the next header; It may be Digest/
             MD5. */
            RvSipServerAuthObjSetAuthResult(hSrvAuthObj,
                RVSIPSERVER_SECURITY_AUTH_RES_NOT_RELEVANT);
            break;
      default:
                 printf("This state is not implemented\n");
                break;
}
```

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

Figure 10-5 *Implementing a State Change*

SIP Server Security Component Implementation

The SIP Server supplies a general implementation for the Security component, suitable for different applications and environments. The SIP Server implementation supports the Digest/MD5 authentication algorithm and two types of databases. The first is a local database that stores user/password database residing in the memory of the local machine. The second database is the remote LDAP database, where the security component queries an LDAP database for user/password data. The supported database is configured on initialization and cannot be changed. The SIP Server implements all interface functions. The following Security Component API functions are provided by the SIP Serverand enable you to add and remove a user and the user password. The Add and Remove operations are supported only for the local database.

API Functions

The API functions of the SIP Server Security Component implementation are as follows:

RvSipServerSecurityMgrImpConstruct()

Constructs the Security Component and creates and initializes the Security Manager object (*SecurityMgr*). If the Security Component was configure to use local database for the user/password database, the local database is constructed as well. It is the responsibility of the application to construct the Security component and set the *SecurityMgr* handle to the Server Authentication Manager using the RvSipServerAuthMgrSetSecurityMgrHandle() function. This function should be called while initializing the SIP Server.

RvSipServerSecurityMgrImpDestruct()

Destructs the Security Component. If the Security Component is configured to use the local database, the user/password database is destructed as well. It is the responsibility of the application to destruct the Security Component. This function should be called upon shutdown of the SIP Server.

RvSipServerSecurityMgrImpGetResourceStatus()

Returns the current status of resources used by the Security Component.

RvSipServerSecurityMgrImpAddUser()

Adds a user and user password to the Security Component database. This function is disabled when using the Remote LDAP database.

RvSipServerSecurityMgrImpRemoveUser()

Removes a user and user password from the Security Component database. This function is disabled when using the Remote LDAP database.

RvSipServerSecurityImpGetPwd()

Returns the password related to the specified user. This function is disabled when using the Remote LDAP database.

RvSipServerSecurityMgrImpSetClientAuthUsername()

Sets the username of the SIP Server for client-side authentication. The username will be used for authenticating outgoing requests that require authentication.

RvSipServerSecurityMgrImpGetClientAuthUsername()

Returns the username of the SIP Server for client-side authentication.

RvSipServerSecurityMgrImpSetClientAuthPassword()

Sets the password of the SIP Server for client-side authentication. The password will be used for authenticating outgoing requests that require authentication.

RvSipServerSecurityMgrImpGet ClientAuthPassword()

Returns the password of the SIP Server for client-side authentication.

RvSipServerSecurityMgrImpSetNonce()

Sets a new nonce value to the security component (include <"> <">).

Initializing the SIP Server Security Component

The application is responsible for constructing the Security Component and supplying the *SecurityMgr* handle to the *SIPServerMgr* at the initialization of the SIP Server, and for destructing the Security Component when exiting.

Sample Code

The sample code in Figure 10-6 (next page) illustrates initializing the Security Component.

To initialize the component

1. Declare an RvSipServerSecurityCfg structure.
2. Set configuration values in this structure.
3. Construct the SIP Server Security Component implementation.
4. Set the created *SecurityMgr* to the *SIPServerMgr*.

```
/*============================================================*/
void InitializeSecurity(hSipServerMgr)
{
    /*step 1*/
    RvSipServerSecurityCfg        hSecurityCfg;
    RvSipServerSecurityMgrHandle  hSecurityMgr = NULL;
/*step 2*/
    hSecurityCfg.szDomain           = NULL /*optional*/;
    hSecurityCfg.szOpaque           = NULL /*optional*/;
    hSecurityCfg.maxUsers           = 50;
    hSecurityCfg.szRealm            = "MyDomain.Com";
    hSecurityCfg.szPrivateKey       = "qwe123";
    hSecurityCfg.clientAuthUsername = "server1";
    hSecurityCfg.clientAuthPassword = "1234";
    hSecurityCfg.szNonce            = "MyNonce";
/*step 3*/
    RvSipServerSecurityMgrImpConstruct(hSipServerMgr,
                                  &hSecurityCfg,
                                  sizeof(hSecurityCfg),
                                  &hSecurityMgr);
    if(hSecurityMgr == NULL)
    {
         printf("error in constructing Security Component\n";
         return;
    }
    /*step 4*/
    RvSipServerAuthMgrSetSecurityMgrHandle(hSipServerMgr,
                                 hSecurityMgr);

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

Figure 10-6 *Initializing the Security Component*

CONFIGURATION PARAMETERS

The configuration parameters of the Security Server Component are as follows:

maxUsers

The maximum number of users that the Security Server Component database can hold simultaneously.

Default value: RVSIPSERVER_SECURITY_IMP_DEFULT_MAX_USERS

szPrivateKey

Defines the Security Manager private key which is used for nonce generation. If the value of this parameter is NULL, the Security Manager handle is used instead as a default value.

Default value: Security Manager handle

szRealm

Defines the Security Manager realm value (include <"> <">). If the value of this parameter is NULL, the RvSipServerSecurityMgrImpConstruct() function will fail. There is no default parameter available for the realm value.

Default value: NULL.

szOpaque

Defines the Security Manager opaque value. This parameter uses the following format: <"> string <">. If the value of this parameter is NULL, the opaque value will not be added to a 401/407 response.

Default value: NULL.

szQop

Defines the Security Manager quality level of protection value. This parameter uses the following format: <"> string <">. Note the following:

- "auth"—uses the values of the header from the message to build the digital signature, but does not use the body data to build the digital signature.
- "auth-int"—uses the values of the header from the message and the body to build the digital signature. Taking the body into account protects the message integrity.

If the value of this parameter is NULL, the Qop value will not be added to 401/407 response.

Default value: NULL.

szDomain

Defines the Security Manager domain value. This parameter uses the following format: <"> string <">. If the value of this parameter is NULL, the domain value will be added to the 401/407 response.

Default value: NULL.

logFilters

Defines the log filter that will be used by the Security Server Component.

Default value: RV_SIP_SERVER_LOG_DEFAULT

clientAuthUsername

The user name used for client-side authentication.

Default value: NULL.

clientAuthPassword

The user password that is used for client-side authentication.

Default value: NULL.

szNonce

Defines the Security Manager nonce value. This parameter uses the following format <"> string <">. If the value of this parameter is NULL, it will be automatically generated by the Security Manager.

Default value: NULL.

eDbServiceType

The database type that will be used while looking for user password.

Default value: RVSIPSERVER_DB_SERVICE_LOCAL

maxQueries

The maximum number of authentication requests that the Security Server Component can handle simultaneously.

Default value:

RVSIPSERVER_SECURITY_IMP_MAX_SIMULTANEOUS_AUTH.

hLdapMgr

The handle to the LDAP Manager (LDAPMgr). This parameter is relevant only when eDbServiceType = RVSIPSERVER_DB_SERVICE_REMOTE_LDAP.

Default value: NULL.

szPwdAttribName

The name of password attribute of the remote LDAP database.

Default value: NULL.

11

SIP SERVER STD LAYER

INTRODUCTION

The SIP Server STD layer contains a set of non-SIP-related general purpose utilities such as the XML library, generic list, and allocator interface. The SIP Server uses this set of utilities, which can be used by the application as well.

XML LIBRARY

The XML library can be used to handle XML documents. This library allows the application to create, modify, parse and encode an XML document. It also supports XPath for addressing elements in an XML document. The XML Library uses Expat for parsing a buffer into an *XMLDoc* object. The version of Expat with which the SIP Server was tested is 1.95.8. The Expat website is: http://expat.sourceforge.net/. Version 1.95.8 can be found at:

http://prdownloads.sourceforge.net/expat/expat-1.95.8.tar.gz?download

The following is an example of an *XMLDoc* in the Winfo-format (according to RFC 3858).

header → <?xml version="1.0"?>
root tag → <watcherinfo xmlns="urn:ietf:params:xml:ns:watcherinfo" version="0" state="full">
child tag → <watcher-list resource="sip:userA@radvision.com" package="presence">
<watcher id="1879073395" event="subscribe"
status="active">sip:userB@radvision.com</watcher>
</watcher-list>
</watcherinfo>

The XML Library API relates to the following entities:

- XML Manager object (*XMLMgr*)

- XML Document object (*XMLDoc*)
- XML Tag object (*XMLTag*)
- XML Attribute object (*XMLAttr*)
- XML XPath object (*XPath*)

XML Manager Object

The *XMLMgr* manages the collection of the XML Library objects.

XML Document Object

The *XMLDoc* represents an XML document as a tree of *XMLTag* objects. The *XMLDoc* holds the document header and the root tag of the document. The XML Document API functions allow the application to create new XML documents, set the document header and root tag, find XML tags in the document, and encode the XML document to a string. Figure 11-1 illustrates an *XMLDoc*.

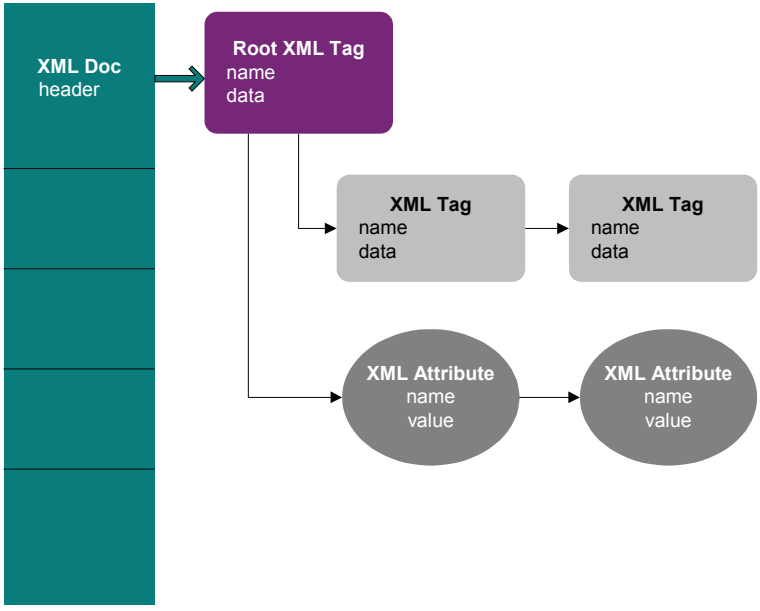


Figure 11-1 *XML Document Object*

XML Tag Object

The *XMLTag* represents an element of the *XMLDoc*. The *XMLTag* is a recursive object that holds a list of child *XMLTag* objects. Each *XMLTag* is inserted to its parent tag in the form of:

```
<parent>parentData
```

```
    <child1>child1Data</child1>
    <child2>child2Data</child2>
</parent>
```

Since the *XMLTag* is recursive, `child1` and `child2` may have a list of child *XMLTag* objects, which will be inserted in the same manner. Each *XMLTag* is a leaf in the tree structure of the *XMLDoc*. There is no limitation on the depth of the tree. The *XMLTag* holds a list of *XMLAttr* objects, and the name and data of the *XMLTag*.

RULES

It is the responsibility of the application to supply well-formed *XMLTag* objects. Well-formed *XMLTag* objects should obey these rules:

- Names can contain letters, numbers, and other characters
- Names must not start with a number or punctuation character
- Names must not start with the letters "xml" (or XML or Xml, and so on)
- Names cannot contain spaces

Note The XML Library will not check that the given tags follow this set of rules.

XML ATTRIBUTE OBJECT

The *XMLAttr* represents an attribute of an *XMLTag*. An *XMLTag* can hold zero to *n* attributes. The *XMLAttr* is identified by its name and a value set to the attribute. The *XMLAttr* will be encoded in the following manner

```
<tag attribute="attribute data">tag data</tag>
```

XML XPATH OBJECT

An XML *XPath* allows addressing an element in an *XMLDoc*. The *XPath* receives an expression in XPath format and returns the corresponding element in the *XMLDoc*. For example, the following expression will return the first child of the root tag which is named "child" with an ID attribute equal to "ch1":

```
/*/child[@id='ch1']
```

XML LIBRARY HANDLES

All XML Library objects are identified using the following handles. These handles must be supplied when calling the XML Library API functions.

- **RvXMLMgrHandle**—the declaration of a handle to the *XMLMgr*. This handle can be received by calling RvSipServerMgrGetXMLMgrHandle(). For more information, see the *RADVISION SIP Server Platform Reference Guide*.

- **RvXMLDocHandle**—the declaration of a handle to an *XMLDoc*. This handle will be given as an OUT parameter when calling RvXMLDocCreate().
- **RvXMLTagHandle**—the declaration of a handle to an *XMLTag*. This handle will be give as an OUT parameter when calling RvXMLTagCreate(), or when calling RvXMLDocGetRootTag() and RvXMLDocGetTagByName().
- **RvXMLAttrHandle**—the declaration of a handle to an *XMLAttr*. This handle will be given as an OUT parameter when calling RvXMLAttrCreate().
- **RvXMLXPathHandle**—the declaration of a handle to an XML *XPath*. This handle will be given as an OUT parameter when calling RvXMLXPathCreate().

XML Library API

This section provides the XML Library API.

Parameters

Listed below are the *XMLDoc*, *XMLTag* and *XMLAttr* parameters.

XMLDoc Parameters

The application can set or examine XMLDoc parameters via Set/Get API functions. The following parameters are available:

Root Tag

The *XMLDoc* holds a handle to the root *XMLTag*. The root tag is the first tag in hierarchy that has no parent tags. The application should retrieve this parameter after constructing a new *XMLDoc* for setting the name and data of the root tag. This parameter can only be accessed with a Get function, and is not modifiable.

Header Name

The header of the *XMLDoc* is the first tag in the *XMLDoc*. This tag is not recursive (does not have any child *XMLTag* objects). To illustrate, in this example, `<?xml version="1.0"?>`, `xml` is the name of the header.

This parameter can be accessed or modified with Get and Set functions. If the application does not set the header, a default value will be used (the default value is same as the above example).

Header Data

The data of the *XMLDoc* header. In the above example, the header data is `version="1.0"`. This parameter can be accessed or modified with Get and Set functions.

XMLMgr

The *XMLDoc* holds a handle to the *XMLMgr* that can be retrieved with a Get function.

XMLTag Parameters

The application can set or examine *XMLTag* parameters via Set/Get API functions. The following parameters are available:

Child Tag

The *XMLTag* holds a list of all its child *XMLTag* objects. A child *XMLTag* is nested inside its parent tag. The child *XMLTag* can be retrieved from the list of child *XMLTag* objects according to its place in the list (first, last, next or previous). The Child Tag parameter can only be accessed via a Get function.

Parent Tag

The *XMLTag* holds a pointer to its parent *XMLTag*. This handle can be accessed via a Get function, and is not modifiable.

Name

The *XMLTag* consists of two parts—name and data. For example, `<tagName>tagData<\tagName>`. In this example, `tagName` is the name of the *XMLTag*. The name will appear in the opening and closing tags.

This parameter can be accessed or modified with Get and Set functions. The application must set the name parameter of an *XMLTag* if not given when creating the *XMLTag*.

Namespace

The namespace part of the name is held in this parameter. It can also be a prefix to a namespace declared in the tag, or its ancestors.

Data

The data of the *XMLTag* in the above example is `tagData`. This parameter is optional and the application is not required to set it. This parameter can be accessed or modified with Get and Set functions.

Attribute

The *XMLTag* holds a list of all its attributes. An *XMLAttr* can be retrieved from this list according to its place (first, last, next or previous). This parameter can be accessed with a Get function.

XMLATTR PARAMETERS

The application can set or examine *XMLAttr* parameters via Set or Get functions. The following parameters are available:

XMLTag

The *XMLAttr* holds a pointer to the *XMLTag* in which it was created. This handle can be retrieved with a Get function, and is not modifiable.

Name

The *XMLAttr* consists of two parts—name and value. For example, `<tag attrName="attrValue"><\tag>`. In this example, `attrName` is the name of the *XMLAttr*. This parameter can be accessed or modified with Get and Set functions. The application must set the Name parameter of the *XMLAttr* if it was not given when creating the *XMLAttr*.

Namespace

The namespace part of the name is held in this parameter. It can also be a prefix to a namespace declared in the tag to which this attribute belongs, or its ancestors.

Value

The value of the *XMLAttr* in the above example is `attrValue`. This parameter can be accessed or modified with Get and Set functions. The application must set the Value parameter of the *XMLAttr* if it was not given when creating the *XMLAttr*.

XML LIBRARY UTILITY FUNCTIONS

The XML Library offers a set of utility functions for creating *XMLDocs*.

Parse

This function is called to parse an *XMLDoc*. It will parse a given buffer into an *XMLDoc* with an hierarchy of XMLTag and XMLAttr objects. Parsing is done with RvXMLDocParse() function.

Encode

When encoding the *XMLDoc* objects, the XML Encoder library will encode all *XMLTag* and *XMLAttr* objects in the *XMLDoc* into a well formed XML document. Encoding can be done using the RvXMLDocEncode() function.

Copy

Copies an *XMLDoc* into another *XMLDoc*.

Searching for an XMLTag

The XML Library enables the application to search for an *XMLTag* by its name. The *XMLTag* can be searched for in the entire *XMLTag* tree (using the RvXMLDocGetTagByName() function), or only within a specific *XMLTag* where the found *XMLTag* will be the child of this tag, (using the RvXMLTagGetChildTagByName() function).

Searching for an XMLAttr

The XML Library enables the application to search for an *XMLAttr* in a specific *XMLTag* by its name (using the RvXMLTagGetAttributeByName() function).

Create Remark

The application can create a remark tag within an existing *XMLTag* of the following form: `<!-- strRemark -->`. A remark can be created using the RvXMLTagRemarkCreate() function.

XPath Parameters

The application will create an *XPath* object when it needs to locate a specific element of a document. It will pass it an expression in XPath format, and then call RvXMLXPathGet().

XPathVariant

This is the result of RvXMLXPathGet(). It is a special data type that can contain any kind of data that may be a result of an evaluated *XPath*.

STD Allocator Interface

The SIP Server STD allocator interface (SipServerAllocatoreIntf) defines a memory allocator interface for various memory allocation purposes used by the SIP Server. Each SipServerAllocatoreIntf can allocate and free different sizes of memory. The SipServerAllocatoreIntf contain two interfaces, one for allocation (RvSipServerStdAllocatorAllocFunc) and one for free (RvSipServerStdAllocatorFreeFunc). In addition, SipServerAllocatoreIntf contains an allocator handle which manages the memory allocation and de-allocation. This handle (RvSipServerStdAllocatorHandle) is passed to both the allocator and free interfaces.

STD List

The SIP Server uses lists as a part of its internal logic and as a part of its interface. For this purpose, the SIP Server STD List object (*StdList*) was defined.

STD List Entities

The STD list (RvSipServerStdList) relates to the following entities:

- STD List—Represent a list object (*StdList*).
- STD List Element—Represent an element in a list. Each *StdListElem* contains a key and data.
- STD List Element Key—Represents an element key (*StdListElementKey*) by which search operations are executed.
- STD List Element Data—Represent the user data (*StdListElementData*) within each *StdListElem*.
- STD List Element Interface—An *StdListElementInterface* represents an interface for allocating, de-allocating and *StdListElem* comparison.

STD List API

The following API functions are provided by STD List:

RvSipServerStdListConstruct()

Constructs an *StdList* based on the given list instance.

RvSipServerStdListDestruct()

Destructs an *StdList*. The destruct operation does not free the list elements. If the list is not empty on destruction, such an operation will cause an assertion failure. Therefore, all *StdListElem* objects must be removed (RvSipServerStdListClear()) from the list before this function is called.

RvSipServerStdListLock()

Locks the *StdList*.

RvSipServerStdListUnlock()

Unlocks the*StdList*.

RvSipServerStdListAddHead()

Creates a new *StdListElem* using the SipServerAllocatoreIntf which passed on RvSipServerStdListConstruct() and adds it to the head of a *StdList*.

RvSipServerStdListAddTail()

Creates new *StdListElem* using the SipServerAllocatoreIntf which passed on RvSipServerStdListConstruct() and adds it to the tail of an *StdList* or after the specified *StdListElem*.

RvSipServerStdListAddElem()

Adds the given *StdListElem* to the *StdList*.

RvSipServerStdListAllocElem()

Asks the StdList SipServerAllocatoreIntf to allocate an *StdListElem*. After such an *StdListElem* was allocated and initialized, it can be added to the *StdList*.

RvSipServerStdListRemoveElem()

Removes and de-allocates the given *StdListElem* using the *StdList* SipServerAllocatoreIntf.

RvSipServerStdListRemove()

Removes and de-allocates an *StdListElem* using StdList SipServerAllocatoreIntf based on a search operation.

RvSipServerStdListIteratorGetFirst()

Initializes the internal *StdList* iterator to the first *StdListElem* and returns this *StdListElem*.

RvSipServerStdListIteratorGetNext()

Assigns the internal *StdList* iterator to the next *StdListElem* of the iterator, and returns this *StdListElem*.

RvSipServerStdListFindElem()

Finds and returns *StdListElemKey* of a given *StdListElem*. The search comparison is based on the specified compare interface, (RvSipServerStdListElemKeyCompareFunc()).

RvSipServerStdListFind()

Finds and returns an *StdListElemData* of a given *StdListElemKey*. The search comparison is based on the specified compare interface, (RvSipServerStdListElemKeyCompareFunc()).

SAMPLE CODE

The following sample code downstrokes how to create an *StdList* and add elements to it. To do so:

1. Declare a RvSipServerStdListInstance.
2. Declare a RvSipServerStdListElemIntf.
3. Initialize all the RvSipServerStdListElemIntf members to 0 using memset().
4. Set the StdAllocatoreIntf and element compare function to RvSipServerStdListElemIntf.
5. Create a *StdList* using RvSipServerStdListConstruct().
6. Create and add new *StdListElem* whose *StdListElemKey* is 1 and its StdListElemData is “data1” using RvSipServerStdListAddHead().

Sample Code

```
/*============================================================*/
RvStatus CreateListExample()
{
      /* Step 1 */
      RvSipServerStdListInstancehInstList;
```

```
/* Step 2 */
RvSipServerStdListElemIntfMyListIntf;


RvSipServerStdListHandlehList;
RvStatus         rv;


/* Step 3 */
memset(&MyListIntf, 0, sizeof(RvSipServerStdListElemIntf));


/* Step 4 */
MyListIntf.allocIntf.pfnAlloc = MyElemAllocFunc;
MyListIntf.allocIntf.pfnFree = MyElemFreeFunc;
MyListIntf.pfnCompare =
                 RvSipServerStdListElemKeyCompareFuncStrings;

/* Step 5 */
rv = RvSipServerStdListConstruct(&hInstList,
                          NULL,
                          &MyListIntf,
                          &hList);
if(RV_OK != rv)
{
  printf("fail to construct a list (rv=%d)\n", rv);
  return(rv);
}


/* Step 6 */
rv = RvSipServerStdListAddHead(hList,
                   (RvSipServerStdListElemKeyHandle)1,

                   (RvSipServerStdListElemDataHandle)"Data1";
if(RV_OK != rv)
{
  printf("fail to add new element to a list (rv=%d)\n", rv);
  return(rv);
}
```

```
        return(rv);
}
/*=================================================================*/
```

12

MEMORY POOL

INTRODUCTION

The SIP Server does not allocate memory dynamically. All memory is allocated during the initialization process and is managed by the SIP Server. 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 Server 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 Server 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.

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 Server memory pool.

COPYING A PAGE INTO A BUFFER

Figure 12-1 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_Success)
      {
        return retStatus;
      }
      memoryString[20] = '\0';

      /* Prints the memory string to screen. */
      printf("The memory string is %s", memoryString);

      return RV_Success;
}
/*=========================================================================*/
```

Figure 12-1 *Copying a Page into a Buffer*

COPYING A BUFFER TO A PAGE

Figure 12-2 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_Success)
          return RV_Failure;

        printf("%s - was added to the page at offset %d\n", source,
                     offset);

        /* Copies information from the page to the dest buffer. */
        retStatus =RPOOL_CopyToExternal(memoryPool,memoryPage,
        offset, dest, size);
        if(retStatus != RV_Success)
          return RV_Failure;
        printf("the dest buffer contains - %s\n",dest);
        return RV_Success;
}
/*=========================================================================*/
```

Figure 12-2 *Copying a Buffer to a Page*

CONSTRUCTING AND DESTRUCTING A MEMORY POOL

Figure 12-3 demonstrates how to construct and destruct a memory pool and how to get a new page for using in calls to SIP Server 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_Success)
        {
               printf("Error getting page from the memory pool");
        }

        /* Calls the SIP Server API function with appPool and
          appPage as parameters. */

        /* Frees appPage page. */
        RPOOL_FreePage(appPool, appPage);

        /* Destructs appPool memory pool. */
        RPOOL_Destruct(appPool);
}
/*============================================================*/
```

Figure 12-3 *Constructing and Destructing a Memory Pool*

Note You can construct the application memory pool when you initialize your application and keep using the memory pool until the application terminates.

13

ADVANCED FEATURES

WORKING WITH DNS

The SIP Server supports working with the DNS mechanism described in RFC 3263, "Session Initiation Protocol (SIP): Locating SIP Servers". Before forwarding a request message, the SIP Server resolves the address of the Request-URI to a concrete transport addresses which it can use in sending the request. Essentially the SIP Server activates an algorithm that selectively uses SRV, NAPTR, A and AAAA DNS queries to map a given SIP address to a prioritized set of transport addresses of the following form:

```
{transport_protocol, IP_address, port}
```

The above list is referred to as the "DNS Address List". The request is sent to the first address on the DNS Address List. In the case of a sending failure, if working as Proxy, the application can choose whether to continue trying sending the request to the next address on the DNS Address List or stop the sending process for the current request.

In the case of a sending failure while working as Back-to-Back User Agent or as Events Server, the SIP Server will continue trying to send the request to the next address on the DNS Address List.

HANDLING SENDING FAILURES

The following are considered as sending failures:

- The transaction was timed out before receiving a final response
- General transport failure
- 503 final response was received

DNS RELATED FUNCTIONS AND EVENTS

When working as Proxy, the SIP Server supplies API functions that enable the application to be notified when a sending request failure occurs. These functions also enable the application to decide whether to continue sending the request to the next address in the DNS Address List or stop the sending process for the current request.

The following API functions can be used on a *ProxyTransc* object that received notification of a failure in sending its outgoing request.

RvProxyCoreTranscDnsContinue()

Requests the SIP Server to continue sending the request to the next address in the DNS Address List. If the DNS Address List is empty, the operation will fail.

RvProxyCoreTranscDnsTerminate()

Requests the SIP Server to stop the sending process for this request. Calling this function causes the termination of the current *ProxyTransc* object.

RVPROXY_CORE_TRANSC_STATE_MSG_SEND_FAILURE

If sending the request failed due to one of the above reasons, the application will be notified about the failure at the RvProxyCoreTranscStateChangeEv() event with the RVPROXY_CORE_TRANSC_STATE_MSG_SEND_FAILURE state.

After receiving this event for a *ProxyTransc*, the SIP Server will continue handling this *ProxyTransc* only upon the decision of the application to continue or stop the sending trials.

The application can access and modify the DNS Address List using DNS Address List API functions. See the *Transport DNS Functions* chapter in the *RADVISION SIP Server SIP Server Platform Reference Guide* for more details about DNS Address List API functions.

With the Back-to-Back User Agent and Events modules, if sending the request failed due to one of the above reasons, the SIP Server will try to continue and send the request to the next address in the list. The number of retries is configurable and can be controlled by the b2bMaxDNSRetries and notifyMaxDNSRetries configuration parameters.

Remarks

Note the following:

- The Transport DNS List object is not protected for a multi-threaded environment at the SIP Server level. It is the responsibility of the application to lock this object to ensure multithreading safety.
- In order to work with this feature, the SIP Server should be compiled with the "Support DNS" configuration. For detailed instructions regarding compilation, see the Release Notes.

Working with IPv6

The SIP Server supports working with IP version 6 (IPv6) addresses in operating systems that have native support for IPv6 such as Solaris, Linux and Windows. The SIP Server can send and receive IPv6 packets (in UDP, TCP and TLS).

To work with IPv6

1. Compile the SIP Server with the "Support IPv6" configuration. (See the Release Notes.)
2. Use IPv6 addresses for the SIP Server network and behavior configuration parameters such as localUdpAddressList, *sipServerDomainsList*, *udpRecordRouteList* and *udpViaList*. For more information about the configuration parameters, see the SIP Server Configuration chapter.

Multihomed Host

The Multihomed Host feature allows sending and receiving requests and responses on different local IP addresses and network interfaces. The application can use a combination of IPv4 and IPv6 local addresses on each one of the supported Transport layers (UDP, TCP and TLS).

The SIP Server allows the application to work with multiple UDP, TCP and TLS addresses, using the following configuration parameters found in the structure, RvSipServerCfg:

- localUdpAddressList—a semicolon-delimited list containing the local UDP address list. The list uses the following syntax: `Address[:port];Address[:port]`
- localUdpAddressListSize—the length of the above list in characters.
- localTcpAddressList—a semicolon-delimited list containing the local TCP address list. The list uses the following syntax: `Address[:port];Address[:port]`

- localTcpAddressListSize—the length of the above list in characters.
- localTlsAddressList—a semicolon-delimited list containing the local TLS address list. The list uses the following syntax: `Address[:port];Address[:port]`
- localTlsAddressListSize—the length of the above list in characters.

For more information about the above parameters, see the SIP Server Configuration chapter.

Receiving Messages

When working with Multihomed IP, the SIP Server can receive requests and responses on any of the configured addresses.

Note When listening to several TCP or TLS addresses, the stackMaxConnections configuration parameter should be increased accordingly.

Sending Messages

This section describes the sending of requests and responses.

Sending Requests

When sending a request, the application can define the local address from which the request will be sent, using the API function of the relevant object. An API function is provided in each of the SIP Server layers, allowing you to define the address from which the requests are sent. The following API functions are provided:

- RvProxyCoreTranscSetLocalAddress()
- RvSipServerB2BDlgSetLocalAddress()
- RvSipServerEventsSubsSetLocalAddress()
- RvSipServerPresSubsSetLocalAddress()
- RvSipServerWinfoSubsSetLocalAddress()
- RvSipServerB2BAFTermSetLocalAddress()

Each of these functions receives five parameters:

- Specified object handle
- Transport type of the local address you wish to set (UDP, TCP, or TLS)
- Address type of the local address you wish to set (IPv4 or 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 Server 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 RvXXXSetLocalAddress() function several times for each requested combination. If, for example, a *B2BDlg* 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 Server. If you configured the SIP Server to listen to a 127.0.0.1 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 a 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 Server uses the socket bound to this address to send the request. In TCP and TLS, when the local outbound address is set, the SIP Server matches a listening address (one of the addresses given for TCP or 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 a random port number.

Sending Responses

When sending a response using UDP transport, the SIP Server uses the same network interface that received the request. When sending a response using TCP or TLS transport, the response will use the same connection on which the request was received. A new connection is constructed only if this connection was closed for some reason.

Working with a Multihomed-host Proxy

A Proxy should be able to handle cases in which a request is received on one interface, and after address resolution, is sent on a different interface. In some cases, a request is forwarded, through the Proxy, from one UA to another UA,

but the two UAs live in different networks and there is no IP level route between them. They can still exchange SIP message, since the Proxy can solve this routing issue using Record-Route.

The SIP Server offers the application two mechanisms to handle these cases:

- Symmetric Record-Route
- Rewrite Record-Route on responses

Symmetric Record-Route Mechanism

Symmetric Record-Route means that if the request was received on interface X and sent using interface Y, the Proxy adds two Record-Route headers to outgoing Requests, in the following order:

- Record-Route:Y
- Record-Route:X

As result of such behavior of the Proxy, each of the UAs builds the same Route-List (in opposite order of course), and each of the UAs will send further requests of the session to the corresponding interface of the Proxy. This also means that the Proxy does not need to analyze the Record-Route in the response.

To enable the Symmetric Record-Route mechanism, the application should:

1. Configure the Proxy to use the Symmetric Record-Route mechanism, by setting the bUseSymmetricRR configuration parameter to RV_TRUE.
2. Configure the Proxy to set Record-Route on requests. The application can also configure the Record-Route mode to be set on requests and enable rewriting the Record-Route in responses (by setting eRecordRoute to RVSIPSERVER_RECORD_ROUTE_MODE_REQ_AND_RESP_REWRITE), but this is not necessary.

Note For more information on the Symmetric Record-Route mechanism refer to section 6.2 of RFC 3608, “Session Initiation Protocol (SIP) Extension Header Field for Service Route Discovery During Registration”. You can also refer to a discussion on the subject at http://www.ietf.org/mail-archive/working-groups/sip/current/msg06589.html.

Rewriting Record-Route on Response

Rewriting the Record-Route header on responses means that the request will be sent with a Record-Route header with the address that corresponds to the outgoing interface from which the request is sent. After the response is received

at the Proxy—before forwarding it to the User Agent client (UAC)—the Proxy will allow the application to rewrite the Record-Route header to contain an address that corresponds to the incoming interface on which the request was received.

In this case, if the request was received on interface X, and forwarded using interface Y, the outoging request will be with Record-Route:Y. Also, before forwarding the response to the User Agent client (UAC), the Record-Route header will be rewritten to be Record-Route:X.

As the result of such behavior of the Proxy, each of the User Agents builds a different Route-List. This also means that the Proxy needs to analyze the Record-Route headers in responses, in order to look for a Record-Route header that was set by the Proxy and needs to be rewritten.

To enable the Rewrite Record-Route on response mechanism, the application should:

1. Configure the Proxy to set the Record-Route headers on requests and enable rewriting the headers on responses, by setting eRecordRoute to RVSIPSERVER_RECORD_ROUTE_MODE_REQ_AND_RESP_REWRITE.
2. Implement the RvProxyCoreObjRewriteRecordRouteInResp() callback of the *ProxyCoreObj* that enables rewriting the Record-route header before forwarding the response.

Configuring the Proxy to Work with Record-Route

Each of the above mechanisms above requires the Proxy to implement the policy of setting Record-Route headers to outgoing requests. It is the responsibility of the application to set the correct values for the Record-Route header during configuration.

The application should add a Record-Route List to set to each defined interface, according to the transport type and the number of interfaces in each type. The number of elements in the Record-route List should be equal to the number of listening addresses (interfaces) in the same transport type, or equal to zero (empty Record-Route List).

Example

Assume that the Server configuration is as follows:

```
udpRecordRouteList = <ip1>;<x.y.com>;<ip3>
tcpRecordRouteList = <ip1;transport=tcp>;< z.w.com;transport=tcp>
```

```
localUdpAddressList = ip1;ip2;ip3
localTcpAddressList = ip1;ip2
```

The Server is configured to listen on 3 UDP addresses and 2 TCP addresses. The Server configuration also contains exactly three elements in the UDP Record-Route List and two elements in the TCP Record-Route List. Each one will be set whenever a request is sent using the corresponding interface.

In such a configuration, if a request is sent from IP2 using UDP transport, the Record-Route header that will be set by the Proxy will contain "`x.y.com`" (the corresponding element in the UDP Record-route List). If a request is sent from IP1 in TCP transport, the Record-Route header that will be set by the Proxy will contain "`ip1;transport=tcp`".

Note For more information about the configuration parameters, see the SIP Server Configuration chapter.

General URL Scheme Support

SIP defines and uses various URL schemes such as "sip", "sips", "pres", "im" and "tel". In addition, some implementations define proprietary URL schemes. An address with a general URL scheme (other than a SIP URL) can appear in various locations in SIP messages, such as in a Request-URI, Contact header, Route header, To header and so on. A general framework in the SIP Server supports the receiving and sending of requests and responses with any type of URL scheme.

Sending Requests to General URL Addresses

Whenever a SIP Server object tries to send a request and the destination address is not a SIP URL, the SIP Server requests the application to supply a SIP URL address to which to send the message. Once the application converts the non-SIP URL into a SIP or SIPS URL address, the SIP Server 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 URL address.

In each of the SIP Server layers, a new callback function was added in the form of RvXXXOtherURLToSendEv(). This callback is called whenever the SIP Server tries to send a request to a non-SIP URL address. In this callback, the application is required to supply the SIP URL address for sending the request.

The following callbacks are supplied:

- **RvProxyCoreTranscOtherUrlToSendEv()**—for requests sent by a stateful *ProxyCoreObj*.

- **RvProxyCoreTrxOtherURLToSendEv()**—for requests sent by a stateless *ProxyCoreObj*.
- **RvSipServerB2BDlgOtherUrlToSendEv()**—for requests sent by a *B2BDlg*.
- **RvSipServerDialogOtherUrlToSendEv()**—for requests sent by a *ServerDialog*.
- **RvSipServerEventsNotifyOtherUrlToSendEv()**—for NOTIFY requests sent by *EventsNotify*.
- **RvSipServerPresNotificationOtherUrlToSendEv()**—for NOTIFY requests sent by *PresNotification*.
- **RvSipServerWinfoNotifyOtherUrlToSendEv()**—for NOTIFY requests sent by *WinfoNotify*.
- **RvSipServerB2BAFASOtherURLToSendEv()**—for requests sent by the B2BAFAS-related objects.

Handling Requests Received with General URL Addresses

During request processing, the SIP Server may need to know whether an address belongs to one of the domains for which the SIP Server is responsible. This can happen in one of the following cases:

- When the SIP Server receives a new request, it must determine whether or not to handle this request locally. This decision is based on several criteria, one of which is whether or not the request is destined for one of the SIP Server domains.
- When processing the request as a proxy, in order to resolve the destination set of a request, the proxy is required to examine whether the address (such as the address at the Request-URI) belongs to one of the domains for which the proxy is responsible.
- When handling an incoming registration, one of the phases of validating the new registration is verifying that the address-of-record belongs to one of the domains for which the Registrar is responsible.

The SIP Server examines the relevant address from the request to determine if it represents its local domain. If this address is a general URL, the SIP Server requests the application to determine if it represents its local domain. The SIP Server notifies the application using a set of callbacks in the form of RvXXXOtherURLRcvdEv().

The following callbacks are supplied:

- **RvProxyPolicyMgrOtherUrlRcvdEv()**—called to decide whether or not a new request is for one of the SIP Server domains.
- **RvProxyCoreObjOtherUrlRcvdEv()**—called during the process of requests by a *ProxyCoreObj* when trying to resolve the request address-set.
- **RvProxyRegServerObjOtherURLRcvdEv()**—called by the *RegServerObj* during the process of registration validation.

In order to match further requests with current requests, the SIP Server occasionally performs address comparisons. Two general URL addresses are equivalent if their scheme is the same and their identifiers are equivalent string-wise. For example, the following addresses are not equivalent: “scheme:user@host” and “scheme:user@host?subject=hi”.

The application, however, may wish to ignore parameters while comparing two addresses, as is done for some of the parameters of the SIP URL. Therefore, the SIP Server allows the application to supply a canonical form for each general URL address, via the RvXXXOtherURLRcvdEv() callback. Here for example, the canonical form of “scheme:user@host?subject=hi” could be “scheme:user@host”, which would lead to an equivalence between the two addresses above.

Presence servers occasionally receive a general URL with “pres” schemes. Therefore, the SIP Server, as a Presence Server, canonizes “pres” addresses before calling RvXXXOtherURLRcvdEv(). The canonization done by the SIP Server is obtained by removing all parameters, maintaining only the form of “pres:user@host”. The application can use this canonized form as its canonization for a general URL with “pres” schemes, or it can choose and supply other canonization during the RvXXXOtherURLRcvdEv() callback.

Handling General URL with "Pres" Scheme

A general URL with a “pres” scheme is very similar to a SIP URL, since their core is in the form of “user@host”. Therefore, many implementations consider “sip:user@host” and “pres:user@host” as equivalent and as representing the same logical entity.

The SIP Server allows you to choose its policy via the configuration flag, bEventsDisregardScheme. If TRUE, the “sip”, “sips” and “pres” schemes are disregarded while comparing addresses. For example, “sip:user@host” and “pres:user@host” are equivalent and represent the same logical entity. The default behavior considers schemes in address comparisons, meaning that the configuration flag is set to FALSE by default.

SESSION-TIMER

According to RFC 4028, Session-Timers are mostly of interest to call stateful proxies, to determine whether or not the call is still active. However, a stateful proxy can also participate in the session negotiation. Stateless proxies must not attempt to request Session-Timers since the proxy needs to remember information between the request and response, ruling out stateless proxies. Proxies that ask for Session-Timers need to enable record-route so that the refresh request will pass through them.

PROXY BEHAVIOR

The SIP Server in its transaction stateful proxy mode fully supports RFC 4028. When INVITE or UPDATE requests are received, the proxy checks for validity of the Session-Expires header against the *ProxyCoreObj* ProxyMonSE. If the value of the Session-Expires header is invalid (smaller than *ProxyCoreObj* ProxyMonSE, the *ProxyCoreObj* will assume the RVPROXY_CORE_OBJ_STATE_INVALID_REQUEST state indicating the RVPROXY_CORE_OBJ_REASON_REQUEST_SESSION_EXPIRES_TOO_SMALL reason. If the application rejects the request with a 422 response code, the *ProxyCoreObj* will add a Min-SE header to the response containing its ProxyMinSE value.

Assuming that the request is valid, the Proxy will modify (if needed) the Session-Expires and the Min-SE header of every INVITE or UPDATE request and response according to the roles defined in RFC 4028.

CONFIGURATION PARAMETERS

The configuration parameters of the Session-Timer feature are as follows:

ProxySupportedExtensionList

A list of supported option-tags, separated by commas, that the Proxy layer supports. The list will be added to a Supported header for outgoing messages.

Note To support the Session-Timer feature at the Proxy level, the application must add the “timer” option tag to the list.

ProxySessionExpires

The call duration that the Proxy requested. This value is inserted into any sent INVITE and UPDATE request that the Proxy forwards in a Session-Expires header according the rules of RFC 4028.

Default: 1800 seconds

ProxyMinSE

The minimum value for the session interval that the Proxy is willing to accept. If the application does not set this parameter, the ProxyMinSE value is set to the default value of 90 seconds according to RFC 4028. This value is inserted into any sent INVITE and UPDATE request that the Proxy forwards in a Min-SE header according the rules of RFC 4028.

Default: 90

API FUNCTIONS

RADVISION SIP Server provides the API functions listed below to support and control the Session-Timer mechanism.

GET AND SET FUNCTIONS

The following Get and Set functions are provided for Session-Timer support:

RvProxyCoreObjSetSTPreferenceParams()

Overwrites the preferred configuration Session-Timer parameters of the *ProxyCoreObj*. These parameters may not be equal to the Session-Timer parameters of the *ProxyCoreObj* at the end of the negotiation.

14

SIP SERVER CONFIGURATION

INTRODUCTION

This chapter describes the configuration of the RADVISION SIP Server. You will find information about the SIP Server configuration parameters and how to set the parameters to suit your specific application requirements.

INITIALIZATION

RADVISION SIP Server configuration and memory allocation is performed upon initialization. You initialize the SIP Server using the function, RvSipServerMgrConstruct(), which is defined as follows:

```
/*=============================================================*/
RvStatus RvSipServerMgrConstruct(
       IN        RvInt32                sizeOfCfg,
       INOUT     RvSipServerCfg*        pServerCfg,
       OUT       RvSipServerMgrHandle*  hSipServerMgr);
/*=============================================================*/
```

Alternatively you can initialize the server using the RvSipServerMgrConstructFromFile() function, which is defined as follows:

```
/*=============================================================*/
RvStatus RvSipServerMgrConstructFromFile(
    IN  RvChar*                  szCfgFilename,
    IN  RvChar*                  szLdapCfgFilename,
    IN  RvSipServerMgrEvHandle*  pEvHandler,
    OUT RvSipServerMgrHandle*    hSipServerMgr);
/*=============================================================*/
```

Configuration Structure

The RvSipServerCfg structure that is used in the RvSipServerMgrConstruct() function contains the entire SIP Server configuration, including the configuration of the underling SIP Stack. The configuration parameters of the RvSipServerCfg can be divided into two groups:

- Group A—parameters that the SIP Server cannot figure by itself and the application must supply. For example, maxProxyCoreObj. If not supplied by application, the SIP Server will use its default value. **Note that some of the parameters do not have predefined default values.**
- Group B—parameters that the SIP Server can calculate according the values of other configuration parameters. For example, maxProxyServerTransc.

The following rules apply to parameters of the RvSipServerCfg structure:

1. If you set a parameter from group A to –1, (RV_SIP_SERVER_DEFAULT) the SIP Server will use a default value. For example, maxProxyCoreObj will be set to 200.
2. If you set a parameter from group B to –1, the SIP Server 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 RvSipServerMgrInitCfg() function can help you to initialize the configuration structure, and you should call it before you construct the SIP Server. This function sets a default value to all parameters.

You can then change the values of the group Group A parameters and the remainder of the parameters will be changed accordingly. The RvSipServerCfg is an INOUT parameter of the RvSipServerMgrConstruct() function. The final calculated configuration values that the SIP Server will use are set back in the structure by the SIP Server.

Setting Addresses within RvSipServerCfg

The SIP Server supports both IPv4 and IPv6, and you can work with both addresses simultaneously. You can use any type of address (IPv4 or IPv6) for all network and address configuration parameters. When configuring to work with IPv6, you should put the addresses in square brackets: []. When configuring listening addresses, (localUDPAddressList, localTcpAddressList, localTlsAddressList) IPv6 address should contain the scope ID as well.

For example:

```
[A1B4:AB3A::AF12]%3
```

Configuration File

The configuration file passed to RvSipServerMgrConstructFromFile() contains properties and values that correspond to member variables in the RvSipServerCfg structure. Directions on how to format specific properties is given in the file using comment directives.

Configuration Parameters

The SIP Server configuration parameters and default values are grouped as follows:

- SIP Server Configuration Parameters
- Proxy Configuration Parameters
- Register Configuration Parameters
- Network Configuration Parameters
- TLS Configuration Parameters
- Multithreading Configuration Parameters
- B2B Configuration Parameters
- Events Configuration Parameters
- Log Configuration Parameters
- Stack Configuration Parameters
- LDAP Parameters
- DNS Configuration Parameters
- XML Manager Configuration
- B2BAF Configuration Parameters
- B2BAF Service Library Configuration Parameters

SIP Server Configuration Parameters

The SIP Server Configuration parameters are grouped as follows:

- Behavior and Policy

Behavior and Policy

eDefaultAuthAlgorithm

Indicates the default authentication algorithm that should be used to build the authentication header in case the received request did not contain an authorization header. The values for this parameter are:

- RVSIP_AUTH_ALGORITHM_MD5
- RVSIP_AUTH_ALGORITHM_OTHER

Default Value: RVSIP_AUTH_ALGORITHM_MD5

XML Tag: DefaultAuthAlgorithm

Remarks: Group A parameter

bLoopDetectionRequired

Indicates if the loop detection algorithm should be performed.

Default Value: RV_FALSE.

XML Tag: IsLoopDetectionRequired

Remarks: Group A parameter

sipServerDomainsList

The list of domain names for which the SIP Server is responsible. The domains should be separated by a semicolon (;).

Example: `172.35.6.68;server.radvision.com`

Default Value: empty list

XML Tag: SipServerDomainsList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- The Server will fail to initialize if sipServerDomainsList is empty. The application must initialize this parameter with at least one domain. There is no default domain value.
- Group A parameter

sipServerDomainsListSize

The size of sipServerDomainsList in characters.

Default Value: 0

Remarks: Group A parameter

eSipServerMode

The working mode of the SIP Server. The possible modes are:

- RVSIPSERVER_MODE_STATELESS
- RVSIPSERVER_MODE_TRANSAC_STATEFULL
- RVSIPSERVER_MODE_DIALOG

- RVSIPSERVER_MODE_DYNAMIC

The mode from configuration will determine how to handle an incoming request—as SL Proxy, SF Proxy or by opening a dialog. If working in dynamic mode, by using callbacks the application will receive notification about each new incoming request, and should decide at the callbacks how to handle the incoming request. For more information, see the SIP Server Architecture chapter.

Default Value: RVSIPSERVER_MODE_TRANSAC_STATEFULL

XML Tag: SipServerMode

Remarks: Group A parameter

proxySupportedExtensionList

The list of supported option-tags that the Proxy Core supports, separated by a semicolon (;). The Proxy Core will use this list when checking the validity of incoming Requests. (Proxy-Require option-tags).

Default Value: NULL (empty list)

XML Tag: ProxySupportedExtensionList

Remarks: Group A parameter

proxySupportedExtensionListSize

The size of the sipServerSupportedExtensionList in characters.

Default Value: 0

Remarks: Group A parameter

UniqueServerId

A NULL terminated string which uniquely identifies the SIP Server instance within a farm of SIP Servers. This unique identifier is used when the High Availability mechanism is enabled. The size of the unique identifier must not exceed RVSIPSERVER_MAX_ADDRESS_LEN.

Default Value: empty string

XML Tag: UniqueServerId

Remarks: Group A parameter

maxNumOfDomainNames [Server configuration parameters]

The maximum number of Domain elements that the SIP Server can be responsible for simultaneously.

Default Value: 1

XML Tag: maxNumOfDomainNames

Remarks: Group B parameter

PROXY CONFIGURATION PARAMETERS

The Proxy configuration parameters are grouped as follows:

- Resource Allocation of Objects
- Behavior and Policy

RESOURCE ALLOCATION OF OBJECTS

maxProxyCoreObj

The maximum number of *ProxyCoreObj* objects that the SIP Server created and which exist simultaneously.

Default Value: 200

XML Tag: MaxProxyCoreObj

Remarks: Group A parameter

maxProxyServerTransc

The maximum number of server Proxy transactions that the SIP Server handles simultaneously. Each *ProxyCoreObj* will hold the maximum of one server transaction.

Default Value: 1 per *ProxyCoreObj*

XML Tag: MaxProxyServerTransc

Remarks: Group B parameter

maxProxyClientTransc

The maximum number of client Proxy transactions that the SIP Server handles. Each *ProxyCoreObj* will hold zero or more client transactions. (Zero client transactions for requests processed by the Proxy such as registration; one client transaction for forwarding the request; and more than one client transaction when forking the request).

Default Value: 1 per *ProxyCoreObj* + a small fixed number

XML Tag: MaxProxyClientTransc

Remarks: Group B parameter

maxServerAuthObjects

The maximum number of Server Authentication objects. These objects are required for the authentication of incoming requests.

Default Value: The default value is 50% of the maximum memory that the object requires, since all the elements do not perform authentication at the same time. The value is:

1 per *ProxyCoreObj* +
1 per *RegServerObj* +
1 per *B2B* +
1 per *B2BTransc* +
1 per *PresSubscribe* +
1 per *EventsSubs* +
1 per *WinfoSubs* +
1 per *PublishServerObj*

XML Tag: MaxServerAuthObj

Remarks: Group B parameter

generalSipServerPoolNumOfPages

The SIP Server general pool page number used for object internal memory and for storing messages with the Server. The messages do not always exist on the objects—this depends on the object state.

Default Value: The default value is 50% of the maximum memory that the object requires, since all the elements do not perform authentication at the same time. The value is:

3 per *ProxyCoreObj* +
1 per Proxy client transaction +
1 per element list +
2 per *RegServerObj* +
1 per *ServerAuthObj* +
1 per *PresSubscribe* +
1 per *PresNotification* +
1 per *EventsSubs* +
1 per *EventsNotify* +
1 per *WinfoSubs* +
1 per *WinfoNotify* +
3 per *B2B* +

1 per *B2BTransc* +

1.5 per *LdapTransc* (1.5 * ldapMaxNumLdapTranscPerEngine * numOfEngine) +

small fixed no. for general usage

XML Tag: GeneralSipServerPoolNumOfPages

Remarks:

- The parameter value must be greater than zero
- Group B parameter

generalSipServerPoolPageSize

The general pool page size.

Default Value: 512

XML Tag: GeneralSipServerPoolPageSize

Remarks:

- The parameter value must be greater than zero
- Group A parameter

maxProxyElemListCount

The Proxy holds several lists for various usages. These lists include header lists for different uses, a list of loose routers, lists of response messages that were received, and more. Not all the lists exist simultaneously. The maxProxyElemListCount parameter defines the number of lists that the SIP Server uses.

Default Value:

3 per *ProxyCoreObj* +

2 per *B2B* +

1 per *EventsSubs* +

1 per *PresSubscribe* +

1 per *WinfoSubs* +

1 per *PAPresentity*

XML Tag: MaxProxyElemListCount

Remarks: Group B parameter

maxProxyTrx

The maximum number of *ProxyTrx* objects that the SIP Server handles simultaneously. Each *ProxyCoreObj* will hold the maximum of one *ProxyTrx*.

Default value: 1 per *ProxyCoreObj*

XML Tag: MaxProxyTrx

Remarks: Group B parameter

smallPagesSize

The small pages page size.

Default Value: 192

XML Tag: SmallPageSize

Remarks:

- The parameter value must be greater than zero
- Group A parameter

maxSmallPages

The maximum amount of SIP Server small pages. The objects use these pages for storing addresses, strings and so on.

Default Value:

0.5 * (1 per *PresSubscribe* +
1 per *WinfoSubs* +
2 per *EventsSubs* +
2 per *PublishServer* object +
2 per *PAPresentity* +
1 per *LDAPTransc* object +
100)

XML Tag: MaxSmallPages

Remarks:

- The parameter value must be greater than zero
- Group B parameter

BEHAVIOR AND POLICY

eRecordRoute

An enumeration that indicates whether the Proxy should add a Record Route header to the outgoing requests and whether to rewrite the value of the Record Route when forwarding responses. The modes are:

- RVSIPSERVER_RECORD_ROUTE_MODE_NONE
- RVSIPSERVER_RECORD_ROUTE_MODE_REQ_ONLY
- RVSIPSERVER_RECORD_ROUTE_MODE_REQ_AND_RESP_REWRITE

The Record Route header will cause all next requests of the same session to be routed through the Proxy.

Default Value:

- If the *bUseSymmetricRR* parameter is FALSE, RVSIPSERVER_RECORD_ROUTE_MODE_NONE (the Proxy will not add Record Route headers to forwarded requests).
- If the *bUseSymmetricRR* parameter is TRUE, RVSIPSERVER_RECORD_ROUTE_MODE_REQ_ONLY (the Proxy will add Record Route headers to forwarded requests only).

XML Tag: RecordRouteMode

Remarks: Group A parameter

bAuthenticateRequests

A Boolean indicating whether or not the Proxy should authenticate incoming requests.

Default Value: RV_FALSE

XML Tag: AuthenticateRequests

Remarks: Group A parameter

bRecurseOn3xx

If this flag is on and a 3xx response is received on a client transaction, the Proxy will recurse the 3xx response according to the proxying type. In case of sequential proxying, the contacts from the 3xx response will be added to the end of the DestAddress list. In case of parallel proxying, a new request will be sent to each contact address.

Default Value: RV_FALSE

XML Tag: RecurseOn3xx

Remarks: Group A parameter

bAutoSend100Trying

If this flag is on, a 100 trying response will be sent automatically through the server transaction to the initiating UA for incoming INVITE requests. This flag can be used only when working in stateful mode.

Default Value: RV_TRUE

XML Tag: AutoSend100Trying

Remarks: Group A parameter

maxNumOfForwardingAttempts

The maximum number of attempts to forward a request. The attempts include requests sent according to the resolved address list and 3xx recursion. This parameter is set in order to prevent a forwarding loop.

Default Value: 10

XML Tag: MaxNumOfForwardingAttempts

Remarks:

- A validity check regarding to the Record-Route port is not performed
- Group A parameter

bUseSymmetricRR

If this flag is on, the proxy will use a Symmetric Record-Route mechanism to traverse through different network interfaces. This flag can be used only when using the multihomed feature.

XML Tag: UseSymmetricRR

Default Value: RV_FALSE

Remarks: Group A parameter

proxyMinSE

The minimum value for the session interval that the proxy will be willing to accept.

Default Value: 90

XML Tag: ProxyMinSE

Remarks: Group A parameter

proxySessionExpires

The time at which an element will consider the call to be timed-out, if no successful INVITE or UPDATE refresh occurred beforehand.

Default Value: 1800 seconds

XML Tag: ProxySessionExpires

Remarks: Group A parameter

REGISTER CONFIGURATION PARAMETERS

The Register configuration parameters are grouped as follows:

- Resource Allocation of Objects
- Behavior and Policy

RESOURCE ALLOCATION OF OBJECTS

maxRegisterNum

The maximum number of registration requests that can be handled simultaneously.

Default Value: 20

XML Tag: MaxRegisterNum

Remarks: Group A parameter

BEHAVIOR AND POLICY

maxBindings

The maximum number of bindings that can be handled simultaneously for all registrations. Each binding represents a contact header in the Registration request.

Default Value: 2 per Registration request

XML Tag: MaxBinding

Remarks: Group A parameter

minExpireValue

The minimum expiration time allowed, in seconds. A registration request with an expiration lower than this value will be returned with a 423 error code.

Default Value: 0

XML Tag: MinExpireValue

Remarks: Group A parameter

maxExpireValue

The maximum expiration time allowed, in seconds. A registration request with an expiration larger than this value will be replaced with the value of maxExpireValue.

Default Value: 2^29 (maximum possible value)

XML Tag: MaxExpireValue

Remarks: Group A parameter

defaultExpireValue

The default value for registration with no expiration value.

Default Value: 3600

XML Tag: DefaultExpireValue

Remarks: Group A parameter

bRegAuthRequests

A Boolean indicating whether or not the Register Server layer should authenticate incoming requests.

Default Value: RV_FALSE

XML Tag: regAuthenticate

Remarks: Group A parameter

regServerSupportedExtensionList

A list of supported option-tags that the Register Server module supports, separated by a semicolon (;). The Register Server will use this list when checking the validity of incoming Requests. (Require header option-tags).

Default Value: NULL

XML Tag: regServerSupportedExtensionList

Remarks: Group A parameter

regServerSupportedExtensionListSize

The size of the regServerSupportedExtensionList in characters.

Default Value: 0

Remarks: Group A parameter

NETWORK CONFIGURATION PARAMETERS

The Network configuration parameters are grouped as follows:

- Interface Addresses
- Record Route Information
- Via Information
- LIR Information

INTERFACE ADDRESSES

localUdpAddressList

A list of local UDP addresses and ports on which the Stack is listening. The addresses are separated by a semicolon (;) and use the following syntax: `address[:port];address[:port]`

If no port is set, a default of 5060 is used.

When using an IPv6 address, you must specify the scope-ID of the local address. These parameters will be used in single or multihomed host applications.

Example: `172.20.1.95:5060;[fe80::2bo:doff:fe9b:4835]%3:5060`

Default Value: NULL.

XML Tag: LocalUdpAddresses

Remarks:

- The application must configure at least one local UDP address.
- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- Group A parameter

localUdpAddressListSize

The length of the localUdpAddressList in characters.

Default Value: 0

Remarks: Group A parameter

localTcpAddressList

A list of local TCP addresses and ports on which the Stack is listening. The addresses are separated by a semicolon (;) and use the following syntax: address[:port];address[:port].

If no port is set, a default of 5060 is used. When using an IPv6 address, you must specify the scope-ID of the local address. These parameters will be used in single or multihomed host applications.

Example: `172.20.1.95:5060;[fe80::2bo:doff:fe9b:4835]%3:5060`

Default Value: NULL

XML Tag: LocalTcpAddresses

Remarks

- If stackTcpEnabled = RV_TRUE, the application must define a local address on which to listen
- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited
- Group A parameter

localTcpAddressListSize

The length of the localTcpAddressList in characters.

Default Value: 0

Remarks: Group A parameter

Record Route Information

udpRecordRouteList

A NULL terminated string containing a list of UDP Record-Route hosts and ports. This list will be used in the Record-Route header that the proxy sent when the destination address is UDP, or is resolved to UDP. The added Record-Route value is selected from the list according to the local interface that will be used to send the request.

For example, if the local address that is used to send the request is the second address from the localUdpAddressList, the proxy will use the second udpRecordRouteList element as the Record-Route value. As a result, this list must be synchronized with the localUdpAddressList and must contain the exact amount of values.

The addresses in this list are separated by a semicolon (;) and use the following syntax:

```
<host[:port][;transport=udp]>;<host[:port][;transport=udp]>
```

If no port is set, the Record-Route header will not contain a port. These parameters will be used in single or multihomed host applications.

Example: `<172.20.1.95:5060>;<radvision.com;transport=udp>`

If this parameter is NULL, the SIP Server will use the first address from the sipServerDomainsList as the Record-Route host name.

Default Value: NULL

XML Tag: UdpRecordRouteList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding to the Record-Route host name is not performed. If the Record-Route host name is not one of the SIP Server domains, additional requests will not passes through the SIP Server.
- Group A parameter
- When using the StartupConfig XML file for initialization, please note that the syntax is different:

  ```
  <UdpRecordRouteList>{host[:port][;transport=udp]};{host[:port][;transport=udp]}</UdpRecordRouteList>
  ```

udpRecordRouteListSize

The length of the udpRecordRouteList in characters.

Default Value: 0

Remarks: Group A parameter

tcpRecordRouteList

A NULL terminated string containing a list of TCP Record-Route hosts and ports. This list will be used in Record-Route header that the proxy sent when the destination address is TCP or resolved to TCP.

The added Record-Route value is selected from the list according to the local address that will be used to send the request. For example, if the local address that is use to send the request is the second one from the localTcpAddressList, the proxy will use the second tcpRecordRouteList element as the Record-Route value. As a result, this list must be synchronized with the localTcpAddressList and must contain the exact amount of values.

The addresses in this list are separated by a semicolon (;) and use the following syntax:

```
<host[:port][;transport=tcp];<host[:port][;transport=udp]>
```

If no port is set, the Record-Route header will not contain a port. These parameters will be used in single or multihomed host applications.

Example:

```
<172.20.1.95:5060;transport=tcp>;<radvision.com;transport=tcp>
```

If this parameter is NULL, the SIP Server will use the first address from the sipServerDomainsList as the Record-Route host name.

Default Value: NULL.

XML Tag: TcpRecordRouteList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding the Record-Route host name is not performed. If the Record-Route host name is not one of the SIP Server domains, additional requests will not passes through the SIP Server.
- Group A parameter
- When using the StartupConfig XML file for initialization, please note that the syntax is different:

```
<TcpRecordRouteList>{host[:port][;transport=tcp]};{host[:port][;transport=tcp]}</TcpRecordRouteList>
```

tcpRecordRouteListSize

The length of the tcpRecordRouteList in characters.

Default Value: 0

Remarks: Group A parameter

Via Information

udpViaList

A NULL terminated string containing a list of Via header host names. This list should be used in the Via header field of outgoing requests that the server sent when the destination address is UDP or resolved to UDP. The Via header host name to use is selected from the list according to the local address that will be used to send the request.

For example, if the local address that is used to send the request is the second one from the localUdpAddressList, the server will use the second udpViaList element as the Via host name. As a result, this list must be synchronized with the localUdpAddressList and must contain the exact amount of values.

This address must resolve one of the SIP Server local addresses. The addresses on the list are separated by a semicolon (;) and use the following syntax:

```
host;host
```

If this parameter is NULL, the SIP Server will use the first local IP as host to the Via header added to outgoing requests.

Default Value: NULL

XML Tag: UdpViaList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding to the Via host name is not performed. If the Via host name is not one of the SIP Server local addresses, the SIP Server will not handle response messages.
- Group A parameter

udpViaListSize

The length of the udpViaListSize in characters.

Default Value: 0

Remarks: Group A parameter

tcpViaList

A NULL terminated string containing a list of Via header host names. This list should be used in the Via header field of outgoing requests that the server sent when the destination address is TCP or resolved to TCP. The Via header host name to use is selected from the list according to the local address that will be used to send the request.

For example, if the local address that is used to send the request is the second one from the localTcpAddressList, the server will use the second tcpViaList element as the Via host name. As a result, this list must be synchronized with localTcpAddressList and must contain the exact amount of values.

This address must resolve one of the SIP Server local addresses. The list is separated by a semicolon (;) and uses the following syntax:

```
host;host
```

If this parameter is NULL, the SIP Server will use the first local IP as host to the Via header added to outgoing requests.

Default Value: NULL

Remarks

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding to the Via host name is not performed.
- If the Via host name is not one of the SIP Server local addresses, the SIP Server will not handle response messages.
- Group A parameter

tcpViaListSize

The length of the tcpViaListSize in characters.

Default Value: 0

XML Tag: TcpViaList

Remarks: Group A parameter

LIR Information

maxNumOfLIR

The maximum number of LIR (local address record) that the SIP Server supports simultaneously. The amount is globally defined for all supported transport types (UDP, TCP, TLS).

Default Value: 1

XML Tag: maxNumOfLIR

Remarks: Group B parameter

TLS Configuration Parameters

The TLS configuration parameters are as follows:

localTlsAddressList

A list of local TLS addresses and ports on which the Stack is listening. The addresses are separated by a semicolon (;) and use the following syntax: `address[:port];address[:port]`

When using an IPv6 address, you must specify the scope-ID of the local address. If a port is not set, a default of 5061 is used. This parameter will be used in single or multihomed host applications.

Example: `172.20.1.95:5061;[fe80::2bo:doff:fe9b:4835]%3:5060`

Default Value: NULL.

XML Tag: LocalTlsAddresses

Remarks:

- if bTlsEnabled = RV_TRUE, the application must define a local address on which to listen.
- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- Group A parameter

localTlsAddressListSize

The length of the localTlsAddressList in characters.

Default Value: 0

Remarks: Group A parameter

bTlsEnabled

Indicates whether or not the TLS is enabled. If set to RV_TRUE, compilation should be with tls=on and TCP should be enabled as well.

Default Value: RV_FALSE

XML Tag: StackTlsEnabled

Remarks: Group A parameter

stackNumOfTlsEngines

The maximal number of TLS engines. TLS engines are used to give a set of properties to a TLS connection.

Default Value: 0

XML Tag: StackNumOfTlsEngines

Remarks: Group A parameter

stackMaxTlsSessions

The maximal number of TLS sessions. A TLS session is the TLS equivalent of a TCP connection and contains TLS data required to manage the TLS connection. The SIP Stack will be able to handle a maximum number of stackMaxTlsSessions concurrent TLS connections.

Default Value: stackMaxConnections (Each opened connection can be used for TLS)

XML Tag: StackMaxTlsSessions

Remarks: Group B parameter

tlsRecordRouteList

A NULL terminated string containing a list of TLS Record-Route hosts and ports. The list will be used in the Record-Route header that the proxy sent when the destination address is TLS with secure scheme (sips).

The added Record-Route value is selected from the list according to the local address that will be used to send the request. For example, if the local address that is used to send the request is the second one from the localTlsAddressList, the proxy will use the second tlsRecordRouteList element as the Record-Route value. As a result, this list must be synchronized with localTlsAddressList and must contain the exact amount of values.

The list is separated by a semicolon (;) and uses the following syntax:

```
<host[:port]>;<host[:port]>
```

If no port is set, the Record-Route header will not contain a port. These parameters will be used in single or multihomed host applications.

Example: `<172.20.1.95:5061>;<radvision.com>`

If this parameter is NULL, the SIP Server will use the first address from the sipServerDomainsList as the Record-Route host name.

Default Value: NULL

XML Tag: TlsRecordRouteList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding to the Record-Route host name is not performed. If the Record-Route host name is not one of the SIP Server domains, additional requests will not pass through the SIP Server.
- Group A parameter
- When using the StartupConfig XML file for initialization, please note that the syntax is different:

  ```
  <TlsRecordRouteList>{host[:port]};{host[:port]
  }</TlsRecordRouteList>
  ```

tlsRecordRouteListSize

The length of the tlsRecordRouteList in characters.

Default Value: 0

Remarks: Group A parameter

tlsResolvedRecordRouteList

A NULL terminated string containing a list of Record-Route hosts and ports that resolved to TLS addresses. These values will be used in Record-Route header that the proxy sent to a destination that resolved to a TLS, but without a secure scheme (sips).

The added Record-Route value is selected from the list according to the local address that will be used to send the request. For example, if the local address that is used to send the request is the second one from the localTlsAddressList, the proxy will use the second tlsResolvedRecordRouteList element as the Record-Route value. As a result, this list must be synchronized with the localTlsAddressList and must contain the exact amount of values.

The list is separated by a semicolon (;) and uses the following syntax:

```
<host[:port]>;<host[:port]>
```

If no port is set, the Record-Route header will not contain a port. These parameters will be used in single or multihomed host applications.

Example: `<x.y.com>;<radvision.com>`

If this parameter is NULL, the SIP Server will use the first address from the sipServerDomainsList as the Record-Route host name.

Default Value: NULL

XML Tag: TlsResolvedRecordRouteList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding to the Record-Route host name is not performed. If the Record-Route host name is not one of the SIP Server domains, additional requests will not pass through the SIP Server.
- Group A parameter
- When using the StartupConfig XML file for initialization please note that the syntax is different:

  ```
  <TlsResolvedRecordRouteList>{host[:port]};{hos
  t[:port]}</TlsResolvedRecordRouteList>
  ```

tlsResolvedRecordRouteListSize

The length of the tlsResolvedRecordRouteList in characters.

Default Value: 0

Remarks: Group A parameter

tlsViaList

A NULL terminated string containing a list of Via header host names. This list should be used in the Via header field of outgoing requests that the server sent when the destination address is TLS or resolved to TLS.

The Via header host name to use is selected from the list according to the local address that will be used to send the request. For example, if the local address that is used to send the request is the second one from the localTlsAddressList, the server will use the second tlsViaList element as the Via host name. As a result, this list must be synchronised with the localTlsAddressList and must contain the exact amount of values.

This address must resolve one of the SIP Server local addresses. The list is separated by a semicolon (;) and uses the following syntax:

```
host;host
```

If this parameter is NULL, the SIP Server will use the first local IP as host to the Via header added to outgoing requests.

Default Value: NULL

XML Tag: TlsViaList

Remarks:

- Using IPv4 local address notation (0.0.0.0) or IPv6 local address notation ([0:0:0:0:0:0:0:0]) is prohibited.
- A validity check regarding the Via host name is not performed. If the Via host name is not one of the SIP Server local addresses, the SIP Server will not handle response messages.
- Group A parameter

tlsViaListSize

The length of the tlsViaListSize in characters.

Default Value:0

Remarks: Group A parameter

Multithreading Configuration Parameters

The Multithreading configuration parameters are as follows:

stackNumberOfProcessingThreads

The number of processing threads to be initiated. If this value is set to 0 or a negative number, the SIP Server will be executed in the single-thread mode.

Default Value: 0

XML Tag: StackNumberOfProcessingThreads

Remarks: Group A parameter

stackProcessingQueueSize

The maximum length of the processing queue.

Default Value: 10 +

maxProxyServerTransc +
maxB2BTransc +
maxPresSubs +
maxWinfoSubs +
maxEventsSubs +
maxProxyClientTransc +
maxConnections

XML Tag: StackProcessingQueueSize

Remarks: Group B parameter

stackNumOfReadBuffers

The number of read buffers.

Default Value: (processingQueueSize + numberOfProcessingThreads) * 1/2

XML Tag: StackNumOfReadBuffers

Remarks: Group B parameter

Server Dialog Configuration Parameters

maxStackDialogs

The maximum number of Stack *call-legs* objects (for working above Server Dialog or B2BAF layers)

Default Value: 0

XML Tag: maxStackDialogs

Remarks:

- Group A parameter.
- When working with the B2BAF Add-on Module, this parameter must match RvSipServerB2BAFCfg.numOfB2BAFTerm.

eDialogOwner

eDialogOwner

Indicates which module will handle incoming dialogs from the following list:

- RVSIPSERVER_DIALOG_OWNER_B2B—the B2B Add-on Module will handle the created dialogs.
- RVSIPSERVER_DIALOG_OWNER_APP—the application will handle the created dialogs.
- RVSIPSERVER_DIALOG_OWNER_DYNAMIC—the application can decide which module will handle each created dialog.
- RVSIPSERVER_DIALOG_OWNER_B2BAF—the B2BAF Add-on Module will handle the created dialogs.

Default Value: RVSIPSERVER_DIALOG_OWNER_B2B

XML Tag: DialogOwner

bDisableRefer3515Behavior

Indicates how to handle REFER messages within a call. RFC 3515 defines a new implementation for REFER. According to RFC 3515, REFER messages are handled using Subscription objects. If you set the bDisableRefer3515Behavior parameter to RV_TRUE, the REFER interface will be used. If this parameter is set to RV_FALSE, the application must handle this REFER message as a general request. When using this option, the REFER interface will be disabled. For more information, see the RADVISION SIP Server Back-to-Back Add-on Module Programmer Guide and Reference Guide.

Default Value: RV_FALSE.

XML Tag: bDisableRefer3515Behavior

B2B Configuration Parameters

The B2B configuration parameters are grouped as follows:

- Resource Allocation of Objects
- Behavior and Policy

Resource Allocation of Objects

b2bMaxB2B

The maximum number of *B2B* objects that the SIP Server created and that exist simultaneously.

Default Value: 200

XML Tag: b2bMaxB2B

Remarks: Group A parameter

b2bMaxB2BDlg

The maximum number of *B2BDlg* objects that the SIP Server created and that exist simultaneously.

Default Value: 2 per *B2B*

XML Tag: b2bMaxB2BDlg

Remarks: Group B parameter

b2bMaxB2BTransc

The maximum number of *B2BTransc* objects that the SIP Server created and that exist simultaneously.

Default Value: 2 per *B2B*

XML Tag: b2bMaxB2BTransc

Remarks: Group B parameter

Behavior and Policy

b2bAuthenticationNeeded

A Boolean indicating whether or not the B2B layer should authenticate incoming requests.

Default Value: RV_FALSE

XML Tag: b2bAuthenticateRequests

Remarks: Group A parameter

b2bAutoTry

If this flag is on, a 100 trying response is sent automatically through the server transaction to the initiating UA for incoming INVITE requests.

Default Value: RV_TRUE

XML Tag: b2bAutoSend100Trying

Remarks: Group A parameter

b2bMaxDNSRetries

The maximum number of attempts to forward a request when a DNS error occurs.

Default Value: 0

XML Tag: b2bMaxDNSRetries

Remarks: Group A parameter

b2bSupportedExtensionList

List of supported option-tags that are supported by the B2B layer, separated by a semicolon (;). The B2B will use this list when checking the validity of incoming requests.

Default Value: NULL

XML Tag: b2bSupportedExtensionList

Remarks: Group A parameter

b2bSupportedExtensionListSize

The size of the b2bSupportedExtensionListSize in characters.

Default Value: 0

Remarks: Group A parameter

b2bMinSE

The minimum value for the session interval that the application is willing to accept.

Default Value: 90

XML Tag: b2bMinSE

Remarks: Group A parameter

b2bSessionExpires

The time at which an element will consider the call to be timed-out, if no successful INVITE or UPDATE refresh occurred beforehand.

Default Value: 1800 seconds

XML Tag: b2bSessionExpires

Remarks: Group A parameter

eB2BSTRefresher

The preferred refresher type (UAC or UAS) that will be added to the Session-Expires header. If no refresher was configured, none will be added to the Session-Expires header.

Default Value: RVSIP_SESSION_EXPIRES_REFRESHER_NONE

XML Tag: b2bSessionRefresher

Remarks: Group A parameter

bB2BSupportClientSideAuth

Indicates whether or not the B2B layer supports client side authentication.

Default Value: RV_FALSE.

XML Tag: b2bSupportClientSideAuth

Remarks: Group A parameter

Events Configuration Parameters

The Events configuration parameters are grouped as follows:

- Resource Allocation of Presence Objects
- Resource Allocation of Winfo Objects
- Resource Allocation of Events Objects
- Resource Allocation of Events XML Objects
- Resource Allocation of Publish Objects
- Resources Allocation of Presence Agent Objects
- Behavior and Policy

Resource Allocation of Presence Objects

presMaxNumPresSubs

The number of *PresSubscribe* objects to allocate. This is the number of *PresSubscribe* that the Presence Server can handle simultaneously.

Default Value: 20

XML Tag: presMaxSubs

Remarks: Group A parameter

presMaxNumPresNotification

The number of *PresNotification* objects to allocate. This is the number of *PresNotification* objects that the Presence Server can send simultaneously to the various *PresSubscribe* objects.

Default Value: 2 per *PresSubscribe*

XML Tag: presMaxNotifications

Remarks: Group B parameter

Resource Allocation of Winfo Objects

winfoMaxNumSubs

The number of *WinfoSubs* objects to allocate. This is the number of *WinfoSubs* objects that the Winfo module can handle simultaneously.

Default Value: 20

XML Tag: winfoMaxSubs

Remarks: Group A parameter

winfoMaxNumNotify

The number of *WinfoNotify* objects to allocate. This is the number of *WinfoNotify* objects that the Winfo module can send to the different *WinfoSubs* objects simultaneously.

Default Value: 2 per *WinfoSubs*

XML Tag: winfoMaxNotify

Remarks: Group B parameter

Resource Allocation of Events Objects

eventsMaxNumSubs

The number of *EventsSubs* objects to allocate. This is the number of subscriptions that the Events module can handle simultaneously (for working directly above the Events module).

Default Value: 20

XML Tag: eventsMaxSubs

Remarks: Group A parameter

eventsMaxNumNotify

The number of *EventsNotify* objects to allocate. This is the number of NOTIFY messages that the Events module can send to the different Subscribers (for working directly above the Events module).

Default Value: 2 per *EventsSubs*

XML Tag: eventsMaxNotify

Remarks: Group B parameter

Resource Allocation of Events XML Objects

The following attributes are deprecated. If these attributes are not set, their values are taken from the configuration of the general resource allocation of the XML object configuration. Developers should use the XMLCfg configuration member to configure the resources of XML objects.

For backward compatibility reasons, in the current version, the values set to these parameters will be used in case the values of the XMLCfg were not set. Note that future versions may not longer have this feature.

XMLMaxNumDocs

Deprecated. Please refer to XMLCfg for further information.

XMLMaxNumTags

Deprecated. Please refer to XMLCfg for further information.

XMLMaxNumAttributes

Deprecated. Please refer to XMLCfg for further information.

XMLMaxNumPages

Deprecated. Please refer to XMLCfg for further information.

XMLPageSize

Deprecated. Please refer to XMLCfg for further information.

Resource Allocation of Publish Objects

pubMaxPublishServer

The number of *PublishServer* objects to allocate. This is the number of PUBLISH requests that the Publish module can handle concurrently.

Default value: 20

XML Tag: pubMaxPublishServer

Remarks: Group A parameter

Resources Allocation of Presence Agent Objects

paMaxPAPresentity

The maximum number of *PAPresentity* objects to allocate. This is the number of *PAPresentity* objects that the Presence Agent module can handle concurrently.

Default Value: 20

XML Tag: paMaxPAPresentity

Remarks: Group A parameter

GENERAL BEHAVIOR AND POLICY

bSubsAuthRequests

Indicates whether or not the SUBSCRIBE requests that the Events server handles should be authenticated.

Default Value: RV_FALSE

XML Tag: subsAuthenticate

Remarks: Group A parameter

notifyMaxDNSRetries

The maximum number of attempts to forward a NOTIFY request when a DNS error occurs.

Default Value: 0

XML Tag: notifyMaxDNSRetries

Remarks: Group A parameter

subsDefaultExpireValue

The default expiration value that should be used for accepting a Subscribe Request.

Default Value: 3600 (seconds)

XML Tag: subsDefaultExpire

Remarks: Group A parameter

subsWaitingExpire

The expiration value for a waiting Presence subscription.

Default Value: 0

XML Tag: subsWaitingExpire

Remarks: Group A parameter

subsMinExpire

The minimum expiration value for a waiting Presence subscription.

Default Value: 0

XML Tag: subsMinExpire

Remarks: Group A parameter

allowEventsHeader

The string that will be used for sending an Allow-Events header. The default value of the Allow-Events header is:
`Allow-Events: presence, presence.winfo`.

XML Tag: AllowEventsHeaderValue

Remarks: Group A parameter

allowEventsHeaderSize

The size of the Allow-Events header string.

Default Value: 23

Remarks: Group A parameter

bPubAuthRequests

Indicates whether or not to authenticate incoming PUBLISH requests.

Default value: RV_FALSE

XML Tag: pubAuthenticate

Remarks: Group A parameter

pubMinExpire

The minimum expiration value for incoming PUBLISH requests. PUBLISH requests with an expiration value smaller than pubMinExpire should be rejected with a 423 response, with a MinExpire header with the value of pubMinExpire.

Default Value: 0

XML Tag: pubMinExpire

Remarks: Group A parameter

pubDefaultExpire

The default expiration value that should be used for incoming PUBLISH requests in case they did not contain an Expire header.

Default Value: 3600 seconds

XML Tag: pubDefaultExpire

Remarks: Group A parameter

pubNumPackages

The number of elements at pubInfoPerPackage that includes valid content types per event package and template. If this parameter is zero, the validity check of the content type value will not be performed.

Default value: 0

Remarks: Group A parameter

pubInfoPerPackage

A list of valid content types per package and template. Each element contains the following:

- A NULL terminated string with the event package name
- A NULL terminated string with the event template name (can be NULL, meaning no template).
- A NULL terminated string with valid content types for the element event package and template, separated by a semicolon (;).

Default Value: NULL

XML Tag: pubInfoPerPackage

Remarks: Group A parameter

eventsSupportedExtensionList

A list of supported option-tags, separated by a semicolon (;), that the Events package supports. Each of the Event package modules will use this list to check the validity of incoming Requests. (Require header option-tags). The list is used for incoming SUBSCRIBE requests as well as PUBLISH requests.

Default Value: NULL (empty list).

XML Tag: eventsSupportedExtensionList

Remarks: Group A parameter

eventsSupportedExtensionListSize

The size of the eventsSupportedExtensionList in characters.

Default Value: 0

Remarks: Group A parameter

allowHeader

The string that will be used for sending an Allow header (in a 2xx response for OPTIONS).

Default value: “Allow: PUBLISH” (when working with the Events module).

XML Tag: AllowHeaderValue

Remarks: Group A parameter

allowHeaderSize

The size of the Allow header string.

Default value: 0

Remarks: Group A parameter

bEventsDisregardScheme

Indicates whether or not schemes should be ignored when comparing two addresses. For example, `pres:user@host==sip:user@host`.

Default Value: RV_FALSE

XML Tag: eventsDisregardScheme

Remarks: Group A parameter

paMinTimer

An interval which defines the minimum time that elapsed between NOTIFY requests to a Watcher. When this interval expires, the Presence Agent will verify whether there was a change in the Presentity state, and only if there was a change, a NOTIFY will be sent to all Watchers.

Default: 300 seconds

XML Tag: paMinTimer

Remarks:

- Group A parameter
- A value of zero indicates that this timer is not active.

paForceTimer

An interval which defines the maximum time that elapsed between NOTIFY requests to Watchers. When this interval expires, the Presence Agent will initiate a NOTIFY to all Watchers, regardless of whether or not the state was changed.

Default Value: 600 seconds

XML Tag: paForceTimer

Remarks:

- A value of zero indicates that this timer is not active
- Group A parameter

bPaAddRegInfo

Defines whether or not to add registration information for Presence status. If this flag is configured to RV_TRUE, the Presence Agent will add the information from an accepted registration to the SoftState sent to the Watchers. If this flag is set to RV_FALSE, the information from accepted registrations will not be used for SoftState status.

Default value: RV_FALSE

XML Tag: bPaAddRegInfo

Remarks: Group A parameter

bUsePASC

Defines whether or not to work with the Presence Agent Server component. If RV_TRUE, the PASC (Presence Agent Server Component) will work, PASC callbacks will be called, and no Presence Agent add-on functionality will be implemented. If RV_FALSE, PA add-on will work.

Default Value: RV_FALSE

XML Tag: bUsePASC

Remarks: Group A parameter

bPaAggHardState

Defines whether to aggregate HardState information to SoftState when sending a NOTIFY request with an *XMLDoc* to the Watchers.

Default Value: RV_FALSE

XML Tag: bPaAggHardState

Remarks: Group A parameter

bEnablePartialPresence

Defines whether or not to support Partial Presence as part of the Presence Agent module. When enabled, the Presence Agent module will support publishes and subscribes of type “application/pidf-diff+xml”

XML Tag: bEnablePartialPresence

Default Value: RV_FALSE

LOG CONFIGURATION PARAMETERS

The Log configuration parameters are as follows:

stackCallLegLogFilters

Stack Call-leg module log filters.

Default Value: Set by defaultLogFilters

XML Tag: StackCallLegLogFilters

Remarks: Group A parameter

serverDialogLogFilters

Server Dialog module log filters.

Default Value: Set by defaultLogFilters

XML Tag: ServerDialogLogFilters

Remarks: Group A parameter

b2bLogFilters

B2B module log filters.

Default Value: Set by defaultLogFilters

XML Tag: B2BUALogFilters

Remarks: Group A parameter

defaultSipServerLogFilters

Sets the default logging level for the SIP Server and the underlying SIP Stack modules.

Default Value: 63—full logging information.

XML Tag: DefaultSipServerLogFilters

Remarks: Group A parameter

sipServerMgrLogFilters

SIP Server Manager log filters.

Default Value: Set by defaultLogFilters

XML Tag: SipServerMgrLogFilters

Remarks: Group A parameter

policyLogFilters

Policy module log filters.
Default Value: Set by defaultLogFilters
Xml Tag: PolicyMgrLogFilters
Remarks: Group A parameter

transactionSFLogFilters

Transaction Stateful module log filters.
Default Value: Set by defaultLogFilters
XML Tag: TransactionSFLogFilters
Remarks: Group A parameter

statelessLogFilters

Stateless module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StatelessLogFilters
Remarks: Group A parameter

coreAPILogFilters

Core API module log filters.
Default Value: Set by defaultLogFilters
XML Tag: CoreAPILogFilters
Remarks: Group A parameter

srvAuthLogFilters

Server Authentication module log filters.
Default Value: Set by defaultLogFilters
XML Tag: ServerAuthLogFilter
Remarks: Group A parameter

regServerLogFilters

Register Server module log filters.
Default Value: Set by defaultLogFilters
XML Tag: RegServerLogFilters

Remarks: Group A parameter

proxyListLogFilters

SIP Server List module log filters.

Default Value: Set by defaultLogFilters

XML Tag: ProxyListLogFilter

Remarks: Group A parameter

serverTransportLogFilters

SIP Server List module log filters.

Default Value: Set by defaultLogFilters

XML Tag: ServerTransportLogFilters

Remarks: Group A parameter

pfnPrintLogEntryEvHandler

A function pointer to an application-defined log callback. By registering to this callback, the user can override the default SIP Server logging and control both the log output device and the log message structure.

Default Value: NULL, meaning that the default logging system will be used.

Remarks: Group A parameter

sipServerLogContext

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

stackDefaultLogFilters

SDP Stack module log filters.

Default Value: Set by defaultLogFilters

XML Tag: StackDefaultLogFilters

Remarks: Group A parameter

stackAdsLogFilters

SIP Stack default ADS log filters.

Default Value: Set by defaultLogFilters
XML Tag: StackAdsLogFilters
Remarks: Group A parameter

stackTransmitterLogFilters

Stack Transmitter module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTransmitterLogFilters
Remarks: Group A parameter

stackAdsRListLogFilters

Stack ADS RList module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackAdsRListLogFilters
Remarks: Group A parameter

stackAdsRaLogFilters

Stack ADS Ra module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackAdsRaLogFilters
Remarks: Group A parameter

stackAdsRPoolLogFilters

Stack ADS RPool module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackAdsRPoolLogFilters
Remarks: Group A parameter

stackAdsHashLogFilters

Stack ADS Hash module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackAdsHashLogFilters
Remarks: Group A parameter

stackAdsPqueueLogFilters

Stack ADS PQueue module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackAdsPqueueLogFilters
Remarks: Group A parameter

stackSemaphoreLogFilters

Stack Semaphore module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackSemaphoreLogFilters
Remarks: Group A parameter

stackMutexLogFilters

Stack Mutex module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackMutexLogFilters
Remarks: Group A parameter

stackLockLogFilters

Stack Lock module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackLockLogFilters
Remarks: Group A parameter

stackMemoryLogFilters

Stack Memory module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackMemoryLogFilters
Remarks: Group A parameter

stackThreadLogFilters

Stack Thread module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackThreadLogFilters

Remarks: Group A parameter

stackQueueLogFilters

Stack Queue module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackQueueLogFilters
Remarks: Group A parameter

stackTimerLogFilters

Stack Timer module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTimerLogFilters
Remarks: Group A parameter

stackTimeStampLogFilters

Stack Time Stamp module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTimeStampLogFilters
Remarks: Group A parameter

stackClockLogFilters

Stack Clock module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackClockLogFilters
Remarks: Group A parameter

stackTmLogFilters

Stack ADS RList module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTmLogFilters
Remarks: Group A parameter

stackSocketLogFilters

Stack Socket module log filters.

Default Value: Set by defaultLogFilters
XML Tag: StackSocketLogFilters
Remarks: Group A parameter

stackPortRangeLogFilters

Stack Port Range module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackPortRangeLogFilters
Remarks: Group A parameter

stackSelectLogFilters

Stack Select module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackSelectLogFilters
Remarks: Group A parameter

stackHostLogFilters

Stack Host module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackHostLogFilters
Remarks: Group A parameter

stackTlsLogFilters

Stack TLS module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTlsLogFilters
Remarks: Group A parameter

stackAresLogFilters

Stack Ares module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackAresLogFilters
Remarks: Group A parameter

stackSubsLogFilters

Subscription module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackSubsLogFilters
Remarks: Group A parameter

presLogFilters

Present Server module log filters.
Default Value: Set by defaultLogFilters
XML Tag: PresLogFilters
Remarks: Group A parameter

serverEventsLogFilters

SIP Server Events module log filters.
Default Value: Set by defaultLogFilters
XML Tag: ServerEventsLogFilters
Remarks: Group A parameter

winfoLogFilters

SIP Server Winfo module log filters.
Default Value: Set by defaultLogFilters
XML Tag: WinfoLogFilters
Remarks: Group A parameter

paLogFilters

Presence Agent module log filters.
Default Value: Set by defaultLogFilters
XML Tag: paLogFilters
Remarks: Group A parameter

pubLogFilters

SIP Server Publish module log filters.
Default Value: Set by defaultLogFilters
XML Tag: PubLogFilters

Remarks: Group A parameter

XMLLogFilters

SIP Server XML module log filters.
Default Value: Set by defaultLogFilters
XML Tag: XMLLogFilters
Remarks: Group A parameter

ldapLogFilters

LDAP module log filters.
Default Value: Set by defaultLogFilters
XML Tag: LDAPLogFilter
Remarks: Group A parameter

STACK CONFIGURATION PARAMETERS

The Stack configuration parameters are as follows:

stackMessagePoolNumOfPages

The SIP Stack message pool number of pages.
Default Value: 2.5 messages per server or client transaction
XML Tag: StackMessagePoolNumOfPages
Remarks:

- The parameter value must be greater than zero
- Group B parameter

stackMessagePoolPageSize

The SIP Stack message pool page size.
Default Value: 1536
XML Tag: StackMessagePoolPageSize
Remarks:

- The parameter value must be greater than zero
- Group A parameter

stackGeneralPoolNumOfPages

The SIP Stack number of pages in the general pool.

Default Value: 1.5 per server or client transaction

XML Tag: StackGeneralPoolNumOfPages

Remarks:

- The parameter has a minimum value of 3.
- Group B parameter

stackGeneralPoolPageSize

The SIP Stack size of a page in the general pool.

Default Value: 1024

XML Tag: StackGeneralPoolPageSize

Remarks:

- The parameter has a minimum value of 512.
- Group A parameter

stackRetransmissionT1

T1 determines several timers as defined in RFC 3261. For example, When an unreliable transport protocol such as UDP is used, the user agent (UA) should retransmit messages if a response does not arrive, T1 determines the initial retransmission interval.

Default Value: 500

XML Tag: StackRetransmissionT1

Remarks: Group A parameter

stackRetransmissionT2

T2 determines the maximum retransmission interval as defined in RFC 3261.

Default Value: 4000

XML Tag: StackRetransmissionT2

Remarks:

- The parameter value cannot be less than 4000
- Group A parameter

stackRetransmissionT4

T4 determines several timers as defined in RFC 3261. For example, the time that a UAS waits after receiving an ACK message, before terminating the transaction, when working with an unreliable transport protocol such as UDP.

Default Value: 5000

XML Tag: StackRetransmissionT4

Remarks: Group A parameter

stackGeneralLingerTimer

After a server sends a final response, it cannot be sure 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: stackRetransmissionT1 * 64

XML Tag: StackGeneralLingerTimer

Remarks: Group B parameter

stackInviteLingerTimer

After sending an ACK for an INVITE final response, a client cannot be sure the server received the ACK message. The client should be able to retransmit the ACK upon receiving retransmissions of the final response for inviteLingerTimer milliseconds.

Default Value: 32,000

XML Tag: StackInviteLingerTimer

Remarks: Group A parameter

stackProvisionalTimer

When a provisional response is received, the UA should wait N2 milliseconds before time-out termination.

Default Value: 180,000

XML Tag: StackProvisionalTimer

Remarks: Group A parameter

stackCancelGeneralNoResponseTimer

When sending a CANCEL request to a General request, the UA waits cancelGeneralNoResponseTimer milliseconds before time-out termination. This timer must be a positive number (> 0).

Default Value: stackRetransmissionT1 * 64

XML Tag: StackCancelGeneralNoResponseTimer

Remarks: Group B parameter

stackCancelInviteNoResponseTimer

When sending a CANCEL request to an Invite request, the UA waits stackCancelInviteNoResponseTimer milliseconds before time-out termination. This timer must be a positive number (> 0).

Default Value: stackRetransmissionT1 * 64

XML Tag: StackCancelInviteNoResponseTimer

Remarks: Group B parameter

stackGeneralRequestTimeoutTimer

After sending a General request, the UA waits for a final response stackGeneralRequestTimeoutTimer milliseconds before time-out termination. (During this time the UAgent retransmits the request every T1, 2 * T1,., T2,. milliseconds.)

Default Value: stackRetransmissionT1 * 64

XML Tag: StackGeneralRequestTimeoutTimer

Remarks: Group B parameter

stackProxy2xxRcvdTimer

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: stackRetransmissionT1 * 64

XML Tag: StackProxy2xxRcvdTimer

Remarks: Group B parameter

stackProxy2xxSentTimer

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 terminates.

Default Value: 0

XML Tag: StackProxy2xxSentTimer

Remarks: Group A parameter

stackServerConnectionTimeout

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 stackServerConnectionTimeout to change the default behavior of the SIP Stack. If you set stackServerConnectionTimeout to a value bigger than 0, the SIP Stack will set a timer for each server connection and will close it when this timer expires. If you set the stackServerConnectionTimeout 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.

XML Tag: StackServerConnectionTimeout

Remarks: Group A parameter

stackMaxNumElementsInSingleDnsList

The maximum number of elements in a 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 * stackMaxNumElementsInSingleDnsList elements.

Default Value: 5

XML Tag: StackMaxNumElementsInSingleDnsList

Remarks: Group A parameter

stackSendReceiveBufferSize

Indicates the buffer size that the SIP Stack uses for receiving and sending SIP messages.

Default Value: 1024

XML Tag: StackSendReceiveBufferSize

Remarks:

- The parameter value must be greater than zero
- Group A parameter

bStackOutboundProxyEnabled

Indicates whether the outbound proxy is enabled.

Default Value: RV_TRUE

XML Tag: bStackOutboundProxyEnabled

stackOutboundProxyIpAddress

Indicates the IP address of an outbound proxy to which all requests and responses will be sent.

Default Value: 0—no outbound proxy

XML Tag: StackOutboundProxyIpAddress

Remarks: Group A parameter

stackOutboundProxyPort

Indicates the port of the outbound proxy to which all requests and responses will be sent.

Default Value: 5060

XML Tag: StackOutboundProxyPort

Remarks:

- If you set the outbound proxy port to –1, it will be discovered using the procedures defined in RFC 3263.
- Group A parameter

stackOutboundProxyHostName

The host name of an outbound proxy the SIP Stack uses. For each outgoing request, the DNS will be queried for this host IP address.

Default Value: NULL

XML Tag: StackOutboundProxyHostName

Remarks:

- If you set the outboundProxyIPAddress parameter, the outboundProxyHostName parameter will be ignored
- Group A parameter

stackOutboundProxyTransport

Indicates the transport of the outbound proxy the SIP Stack uses.

Default Value: RVSIP_TRANSPORT_UNDEFINED

XML Tag: StackOutboundProxyTransport

Remarks: Group A parameter

stackTcpEnabled

Indicates whether or not the TCP is enabled. If set to RV_FALSE, no connection will be allocated.

Default Value: RV_FALSE

XML Tag: StackTcpEnabled

Remarks: Group A parameter

stackMaxConnections

Indicates the number of TCP connection sockets to be allocated by the SIP Stack.

Default Value: (maxStackTransactions/2)+1

XML Tag: StackMaxConnections

Remarks: Group B parameter

eStackPersistencyLevel

The persistency level to be used by the SIP Stack and SIP Server objects.

Default Value:
RVSIP_TRANSPORT_PERSISTENCY_LEVEL_UNDEFINED—not using persistent connections.

XML Tag: StackPersistencyLevel

Remarks: Group A parameter

stackUseRportParamInVia

Indicates whether to add the rport parameter to the Via header of outgoing requests. The remote party should fill the rport parameter with the port from where the request was received.

Default Value: RV_FALSE

XML Tag: StackUseRportParamInVia

Remarks: Group A parameter

stackCoreLogFilters

Core module log filters. This filter defines the default logging level for all Core module components.

Default Value: Set by defaultLogFilters

XML Tag: StackCoreLogFilters

Remarks: Group A parameter

stackMsgLogFilters

Message module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackMsgLogFilters
Remarks: Group A parameter

stackTransportLogFilters

Message module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTransportLogFilters
Remarks: Group A parameter

stackTransactionLogFilters

Transaction module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackTransactionLogFilters
Remarks: Group A parameter

stackParserLogFilters

Parser module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackParserLogFilters
Remarks: Group A parameter

stackLogFilters

SIP Stack Manager module log filters.
Default Value: Set by defaultLogFilters
XML Tag: StackLogFilters
Remarks: Group A parameter

stackMsgBuilderLogFilters

Message Builder module log filters.

XML Tag: StackMsgBuilderLogFilters

Default Value: Set by defaultLogFilters

Remarks: Group A parameter

stackAuthenticatorLogFilters

Authenticator module log filters.

Default Value: Set by defaultLogFilters

XML Tag: StackAuthenticatorLogFilters

Remarks: Group A parameter

stackElementPoolNumofPages

The SIP Stack number of pages in the element pool.

Default Value: 37 (maxCallLegs*2 + 0.5*maxSubscriptions + maxTlsSessions + 0.75*maxTransmitters) * 1.1

XML Tag: StackElementPoolNumofPages

Remarks:

- Group B parameter
- The parameter value must be greater than zero

stackElementPoolPageSize

The SIP Stack size of a page in the element pool.

Default Value: 250— for holding an element such as a stand-alone header.

XML Tag: StackElementPoolPageSize

Remarks:

- Group A parameter
- The parameter value must be greater than zero

LDAP Parameters

The LDAP parameters are grouped as follows:

- Resource Allocation of Objects
- Behavior and Policy

Resource Allocation of Objects

ldapMaxNumLdapTranscPerEngine

The maximum number of *LDAPTransc* objects that can be handled simultaneously on each of the configured LDAP engines. Each *LDAPTransc* represents an LDAP query (SEARCH, ADD, REMOVE or UPDATE).

Default Value: maxServerAuthObjects + maxRegisterNum

XML Tag: ldapMaxNumLdapTranscPerEngine

Remarks: Group B parameter

BEHAVIOR AND POLICY

pLdapServiceInfoArr

An array of LDAP service information elements containing the service-related query configuration, server lists, working mode, and so on.

Default Value: NULL

XML Tag: ldapServiceInfo

Remarks: Group A parameter

ldapNumOfServiceInfo

The size of the array given in pLdapServiceInfoArr.

Default Value: 0

Remarks: Group A parameter.

eLocationDbServiceType

Defines the location database service type with which to work. The types are:

- RVSIPSERVER_DB_SERVICE_LOCAL
- RVSIPSERVER_DB_SERVICE_REMOTE_LDAP

Default Value: RVSIPSERVER_DB_SERVICE_LOCAL

XML Tag: locationDbServiceType

Remarks: Group A parameter

DNS CONFIGURATION PARAMETERS

The DNS configuration parameters are as follows:

pStackDnsServers

An array that the application allocated holding DNS servers that the SIP Stack will use to resolve names. You **must** set the size of this array in the stackNumOfDnsServers configuration parameters. 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 zero, the port will be converted to the default “well known” DNS port (53).

Default Value: NULL

XML Tag: StackDnsServer

Remarks: Group A parameter

stackNumOfDnsServers

The number of DNS servers in pStackDnsServers.

Default Value: UNDEFINED

XML Tag: StackNumOfDnsServers

Remarks: Group A parameter

pStackDnsDomains

A list of domains for a 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.

XML Tag: StackDnsDomains

Remarks: Group A parameter

stackNumOfDnsDomains

The size of the array given in pStackDnsDomains.

Default Value: UNDEFINED

XML Tag: StackNumOfDnsDomains

Remarks: Group A parameter

stackMaxDnsServers

The maximum number of DNS servers 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 servers it is expected to use in the stackMaxDnsServers parameter.

Default Value: UNDEFINED

XML Tag: StackMaxDnsServers

Remarks: Group A parameter

stackMaxDnsDomains

The maximum 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 domains it is expected to use in the stackMaxDnsDomains parameter.

Default Value: UNDEFINED

XML Tag: StackMaxDnsDomains

Remarks: Group A parameter

stackMaxDnsBuffLen

The maximum length of the buffer that the SIP Stack uses to read DNS query results arriving on TCP.

Default Value: 1024

XML Tag: StackMaxDnsBuffLen

Remarks: Group A parameter

StackDialPlanSuffix

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

XML Tag: StackResolveTelUrls

Remarks: Group A parameter

StackResolveTelUrls

Indicates that tel: schemes should be resolved when supporting the ENUM/TEL URL scheme.

Default Value: RV_FALSE

XML Tag: StackResolveTelUrls

Remarks:

- If this parameter is set to RV_TRUE, the SIP Server will try to resolve tel URLs according to RFC 3764.

- If this parameter is set to RV_FALSE, the SIP Server will call the RvSipXXXOtherURLAddressFoundEv() callback if the remote address contains a tel URLs scheme.
- If you wish the SIP Server to resolve tel URLs, you should also compile the SIP Stack with the RV_SIP_TEL_URI_SUPPORT compilation flag
- Group A parameter

XML Manager Configuration

The following attributes are part of the RvSipServerXMLCfg XMLCfg configuration structure.

For backward compatibility reasons, un-initialized parameters will get the value of the corresponding parameters of the XML configuration of the Events Add-on Module, if the module is enabled. If the events Add-on Module parameters are not initialized, the default value will be set.

XMLMaxNumDocs

The number of *XMLDoc* objects to allocate. This is the number of *XMLDoc* objects that the XML Encoder can encode simultaneously.

Default Value: MaxPAPresentity * 2 +WinfoMaxNumSubs+5

XML Tag: XMLMaxNumDocs

Remarks: Group A parameter

XMLMaxNumTags

The number of *XMLTag* objects to allocate. This is the maximum number of *XMLTag* objects for all *XMLDoc* objects that the XML Encoder handles simultaneously.

Default Value: 10 per XMLDoc

XML Tag: XMLMaxNumTags

Remarks: Group A parameter

XMLMaxNumAttributes

The number of *XMLAttr* objects to allocate. This is the maximum number of *XMLAttr* objects for all *XMLDoc* objects that the XML Encoder handles simultaneously.

Default Value: 2 per XMLTag

XML Tag: XMLMaxNumAttributes

Remarks: Group A parameter

XMLMaxNumPages

The number of memory pages to allocate for the *XMLDoc* objects. Each *XMLDoc* holds a page for its tags and attributes.

Default Value: 1 per XMLDoc

XML Tag: XMLNumPages

Remarks: Group A parameter

XMLPageSize

The size of memory pages to allocate for the *XMLDoc*.

Default Value: 1024

XML Tag: XMLPageSize

Remarks: Group A parameter

B2BAF Configuration Parameters

maxGeneralCallTransc

The amount of general call transactions required by the B2BAF module.

Default Value: 1200

XML Tag: maxGeneralCallTransc

Remarks: This value needs to match numOfB2BAFTransc.

Note The following attributes are part of the RvSipServerB2BAFCfg configuration structure.

---

b2bafLogFilters

Defines the logging filter of the B2BAF module.

Default Value: 63-full logging information

XML Tag: b2bafLogFilters

numOfB2BAFAS

The maximum number of *B2BAFAS* objects created by the B2BAF module and which exist simultaneously.

Default Value: 200

XML Tag: numOfB2BAFAS

numOfB2BAFTerm

The maximum number of *B2BAFTerm* objects created by the B2BAF module and which exist simultaneously.

Default Value: 3 * numOfB2BAFAS

XML Tag: numOfB2BAFTerm

numOfB2BAFTransc

The maximum number of *B2BAFTransc* objects created by the B2BAF module and which exist simultaneously.

Default Value: 3 * numOfB2BAFTerm

XML Tag: numOfB2BAFTransc

totalNumOfServices

The maximum number of *Service* objects created by the B2BAF module and which exist simultaneously.

Default Value: 3 * numOfB2BAFAS

XML Tag: totalNumOfServices

numOfGeneralPages

The maximum number of memory pages that the B2BAF module and the B2BAF Service Library module allocate simultaneously. These pages are used for storing incoming messages, addressees, and so on.

Default Value: numOfB2BAFTerm + 3 * totalNumOfServices

XML Tag: numOfGeneralPages

generalPageSize

The size of each page (numOfGeneralPages).

Default Value: 1024

XML Tag: generalPageSize

b2bafSupportedExtensionList

The list of supported option-tags that the B2BAF supports, separated by a semicolon (;). The B2BAF service will use this list when checking the validity of incoming Requests. (Require option-tags).

Default Value: NULL (empty list)

XML Tag: b2bafSupportedExtensionList

Remarks: Group A parameter

b2bafSupportedExtensionListSize

The size of the b2bafSupportedExtensionList in characters.

Default Value: 0

Remarks: Group A parameter

b2bafSTMinSE

The minimum value for the session interval that the B2BAF is willing to accept.

Default Value: 90

XML Tag: b2bafSTMinSE

Remarks: Group A parameter

b2bafSTSessionExpires

The time at which an element will consider the call to be timed-out, if no successful INVITE or UPDATE refresh occurred beforehand.

Default Value: 1800 seconds

XML Tag: b2bafSTSessionExpires

Remarks: Group A parameter

b2bafSTSessionRefresher

The preferred refresher type (UAC or UAS) that will be added to the Session-Expires header. If no refresher was configured, none will be added to the Session-Expires header.

Default Value: RVSIP_SESSION_EXPIRES_REFRESHER_NONE

XML Tag: b2bafSTSessionRefresher

Remarks: Group A parameter

B2BAFHAEnable

Enables or disables the HA mechanism supported by the B2BAF. To support the HA mechanism, the application must also implement the High Availability Interface.

Default Value: False

XML Tag: maxBlobBufferSize

Remarks: Group A parameter

maxBlobBufferSize

The maximal size in bytes of the B2BAF checkpoint blob.

Default Value: 3500

XML Tag: maxBlobBufferSize

Remarks: Group A parameter

B2BAF Service Library Configuration Parameters

b2bafServiceLibLogFilters

Defines the logging filter of the B2BAF Service Library module.

Default Value: 63-full logging information

XML Tag: b2bafServiceLibLogFilters

numOfDestAddrPerTransparentConnectService

The maximum amount of destination addresses per RvSipServerB2BAFServiceTransparentConnect service.

Default Value: 3

XML Tag: numOfDestAddrPerTransparentConnectService

15

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, threads, 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.
- Mid-layer Thread API—includes a set of functions that enable the application to attach application threads to the SIP Stack interval thread objects, to use SIP Stack functionality within such threads.

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 RADVISION 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 RADVISION. 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 Toolkit.

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 RTP sessions, or 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*.

MID-LAYER THREAD API

The Mid-layer provides an API that enables the application to attach the threads (that the application created) to the SIP Stack internal thread objects (saved on the local storage of the thread). The application must not use the SIP API functions within this thread if the application thread is not attached.

API FUNCTIONS

The following functions that supply thread control are listed below.

RvSipMidAttachThread()

Indicates that a thread created by the application is about to use the SIP Toolkit API functions. This function should be called within the attached thread. Without calling to this function, calls to API functions may cause a crash. Upon thread destruction, the application must call RvSipMidDetachThread() to remove memory allocated by the SIP Toolkit for this thread.

RvSipMidDetachThread()

Indicates that a thread created by the user is being shut down. This function should be called to remove memory that the SIP Toolkit allocated for this thread (memory that was allocated when calling RvSipMidAttachThread()). This function should be called within the thread to be detached.

16

SIP SERVER LOG

INTRODUCTION

The RADVISION SIP Server Platform includes a Log module that produces output for debugging, monitoring and tracking of the activity of applications built with the SIP Server Platform.

The SIP Server Log output provides detailed information which is important for application development and ongoing system maintenance. The logging provides information about the SIP Server Platform and the underlying SIP Stack. Suggestions for using the SIP Server Log include:

- **Post Mortem Monitoring**

  If you want to know the full history of messages going though the SIP Server, use the SIP Server Log output of all network events related to the SIP Server.

- **Problem Tracking**

  The SIP Server Log provides a chronological record of events to track exactly where and when a problem occurred.

- **Interoperability Tracking**

  The SIP Server Log provides an important vehicle for tracking interoperability. You can see if the events on the remote side are interoperable with the events on the local side.

- **Input for Applications**

  SIP Server Log information is an important source for data. You can save the output to a file and extract information as input to billing and similar applications.

- **Reducing Debugging Costs**

  By using the SIP Server Log as a debugging tool for applications built on top of the SIP Server Platform, you can save valuable programmer time and reduce development costs.

SOURCE IDENTIFIERS

The SIP Server Log uses various source identifiers for monitoring SIP Server behavior. You can set up the SIP Server Log to analyze any of the source identifiers and specify the level of logging required for each one independently. Table 16-1 shows the main SIP Server source identifiers that the Log analyzes. Table 16-2 shows the underlying SIP Stack source identifiers.

Table 16-1 *SIP Server Source Identifiers*

Source Identifier	Description
SRVMGR	SIP Server Manager-related operations, such as construction and destruction of the SIP Server.
POLICY	Policy-related operations, such as core object construction.
STATEFULL	Transaction stateful-related operations, such as parallel proxying of a request.
STATELESS	Stateless-related operations, such as stateless forwarding of a request.
CORE_API	Core API-related operations, such as accessing a core API function and SIP Server core callbacks.
SRV_AUTH_OBJ	Authentication-related operations, such as authentication object state changes.
REG_SERVER	Registration-related operations, such as *RegServerObj* creation and state changes.
SRV_LIST	SIP Server list-related operations, such as pushing an element into a list.
SRV_TRANSPORT	SIP Server Transport-related operations.

Table 16-1 *SIP Server Source Identifiers*

Source Identifier	Description
B2BUA	B2B-related operations, such as the creation of a B2B and state changes.
B2BAF	B2BAF-related operations, such as the creation of a *B2BAFTerm*, *B2BAFServiceBase* operation, and so on.
B2BAFSRVS	B2BAF Service library-related operations, such as creation of a *Service* and the handing of *Service*-related events.
SRV_DIALOG	Sever Dialog-related operations, such as the creation of a *ServerDialog* and Server Dialog events.
PRES_SERVER	Presence-related operations, such as the creation of Presence objects (*PresNotification*, *PresSubscribe*) and state changes.
EVENTS	Events Server-related operations, such as the creation of an *EventsSubs* object.
WINFO	Winfo-related operations, such as a state change of a *WinfoSubs* object.
XML_ENCODER	XML Encoder-related operations, such as the encoding of an XML document (*XMLDoc*).

Table 16-1 *SIP Server Source Identifiers*

Source Identifier	Description
PUBLISH	Publish-related operations, such as the creation of *PublishServer* object.
PA	Presence Agent-related operations, such as initiating and preparing NOTIFY requests to Watchers.
LDAPDB	LDAP-related operations, such as creating a *LDAPTransc* to query the LDAP Server.

Table 16-2 *Underlying SIP Stack Source Identifiers*

Source Identifier	Description
TRANSACTION	SIP Stack transaction-related operations, such as transaction state changes.
MESSAGE	SIP Stack message-related operations, such as header creation and manipulation.
TRANSPORT	SIP Stack transport-related operations, such as DNS IP discovery.
PARSER	SIP Stack parser-related operations, such as header or message parsing.
STACK	SIP Stack Manager-related operations, such as Stack construction and destruction.
MSGBUILDER	SIP Stack message builder related operations, such as building a message from a buffer.

Table 16-2 *Underlying SIP Stack Source Identifiers*

Source Identifier	Description
AUTHENTICATOR	SIP Stack authenticator-related operations, such as starting an authentication.
CALL	SIP *Stack CallLeg*-related operations, such as CallLeg state changes.
SUBSCRIPTION	SIP Stack Subscription-related operations, such as subscription object state changes.
TRANSMITTER	Transmitter-related operations.

A full list of available source identifiers can be found in the RvSipServerModule enumeration in the *RADVISION SIP Server Reference Guide*.

Log File

By default, the SIP Server Log is written to a file called *SipSendLog.txt*.

Log Messages

SIP Server Log messages are printed to the Log file along with information about the message type. The following Log message types are available:

- **EXCEP**—indicates that a fatal error has occurred that prevents the SIP Server from continuing to operate.
- **ERROR**—indicates that a non-fatal error has occurred, such as faulty application behavior, insufficient allocations or illegal network activity.
- **WARN**—warns about a possible non-fatal error.
- **INFO**—describes SIP Server activity.
- **DEBUG**—provides detailed information about SIP Server activity.
- **SYNC**—provides detailed information about object locking activity.
- **ENTER**—a message that is printed to the log each time a Core or ADS function is called.
- **LEAVE**—a message that is printed to the log just before leaving a Core or ADS API function.

Message Structure

The Log contains rows of messages. Each row consists of the following fields:

```
<date> <time> <message type> <log entry source> <log entry>
```

The fields in a message row represent the following:

- 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 on page 373 for a list of message types.
- Log entry source—see Source Identifiers on page 370 for a list of the SIP Server source identifiers analyzed by the Log.
- Log entry—the actual Log message item.

The following is an example of a message row:

```
/*================================================================*/
06/11/02 17:34:20 INFO  - POLICY   - ProxyCoreTransConstruct -
CoreObj=1383734 successfully created ProxyTransc=0 IsServer=0
/*================================================================*/
```

Log Configuration

The RvSipServerCfg configuration structure contains parameters that enable configuring the log for each of the source identifiers. For example, the regServerLogFilters configuration parameter will set the log level for the REG_SERVER source identifier. You can also configure the log level to groups of source identifiers using the following configuration parameters:

defaultSipServerLogFilters

The log level that will apply to all the SIP Server Platform modules including the SIP Server, SIP Stack, ADS and CORE modules.

stackDefaultLogFilters

The log level that will apply to all the SIP Stack modules, overriding the default log level.

stackCoreLogFilters

The log level that will apply to all the CORE modules, overriding the default log level.

stackAdsLogFilters

The log level that will apply to all the ADS modules, overriding the default log level.

After setting the log level to the above groups of source identifiers, the SIP Server Manager 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.

Log Filters

The RvSipServerLogFilters enumeration contains a filter for each of the log message types. The following filters are available:

- RVSIPSERVER_LOG_NO_LOG
- RVSIPSERVER_LOG_DEFAULT_FILTER
- RVSIPSERVER_LOG_DEBUG_FILTER
- RVSIPSERVER_LOG_INFO_FILTER
- RVSIPSERVER_LOG_WARN_FILTER
- RVSIPSERVER_LOG_ERROR_FILTER
- RVSIPSERVER_LOG_EXCEP_FILTER
- RVSIPSERVER_LOG_LOCKDBG_FILTER
- RVSIPSERVER_LOG_ENTER_FILTER
- RVSIPSERVER_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 Server construction using the configuration structure, or at runtime using the RvSipServerMgrSetNewLogFilters() API function.

For more information about setting the Log message types, see Log Filters on page 57 of the Using the SIP Server chapter.

Initializing the Log

Before the construction of the SIP Server, the Log can be configured to different logging levels for each source identifier. You do this by filling the relevant fields in the RvSipServerCfg structure.

Every module has a structure member that represents it. The setting made to defaultSipServerLogFilters in the configuration structure will be applied to all.

Sample Code

The following sample code demonstrates how to initialize the SIP Server with all logging levels set only to INFO, while the stateless module is set to ERROR and EXCEP.

```
/*-----------------------------------------------------------------*/
RvSipServerCfg          cfg;
RvSipServerMgrHandle hServer;
```

```
RvSipServerMgrInitCfg(sizeof(cfg), &cfg);
cfg.defaultSipServerLogFilters = RVSIPSERVER_LOG_INFO_FILTER;
cfg.statelessLogFilters = RVSIPSERVER_LOG_ERROR_FILTER |
RVSIPSERVER_LOG_EXCEP_FILTER;
RvSipServerMgrConstruct(sizeof(cfg),&cfg,&hServer);
/*============================================================*/
```

Changing Log Configuration During Runtime

It is possible to change the logging level of any of the modules during runtime using the RvSipServerMgrSetNewLogFilters() function. You can change the logging setting for any of the source identifiers to be any of the filters combinations.

Sample Code

The following sample code demonstrates how to change the logging level for the Policy module during runtime so only errors and exceptions will be logged.

```
/*============================================================*/
RvSipServerMgrHandle hServer;
RvUint8    filter = 0;

filter = RVSIPSERVER_LOG_ERROR_FILTER |
RVSIPSERVER_LOG_EXCEP_FILTER;
RvSipServerMgrSetNewLogFilters(hServer,
RVSIPSERVER_POLICYMGR,filter);
/*============================================================*/
```

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 Server before the SIP Server is constructed. The configuration member that holds this function is pfnPrintLogEntryEvHandler. Each time the SIP Server 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 Server initialization process.

Note Multi-threaded applications must implement PrintLogEntryEvHandler() that locks the Log before writing and unlocks the Log after writing.

Sample Code

The following sample code demonstrates how to replace the default log function with a user callback function. The user callback function prints the Log to a file named *MyLog.txt* and adds a line number before each line in the Log.

```
/*=============================================================*/
RvSipServerCfg serverCfg;
RvSipServerMgrHandle hServer;
RvStatus AppInitSipServer()
{
FILE *logFile;
RvSipServerInitCfg(sizeof(serverCfg), & serverCfg));

/*Opens the log file for writing*/
logFile = fopen("MyLog.txt","w");

/* Registers the log function in the configuration structure. */
stackCfg.pfnPrintLogEntryEvHandler = AppLogPrintFunction;

/* Saves the file pointer as a log context (optional). */
serverCfg.logContext = logFile;

return RvSipServerConstruct(sizeof(serverCfg), & serverCfg, &
hServer);
}

/* 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;

/* Adds a line number to the log message. */
sprintf(data,"%8d: %s\n", line, formattedText);
line ++;

/* Writes the log to the log file. */
fwrite(data, strlen(data), 1, logFile);
```

```
/ *Flushes it down ... */
fflush(logFile);
}
/*=============================================================*/
```

Note When you replace the default logging, the formatted text includes only the message type, log entry source and log entry.

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.

17

SIP SERVER TEST APPLICATION

INTRODUCTION

The SIP Server Test Application is a ready-to-use program that demonstrates how to implement a SIP Server. The Test Application implements the logical function of a Registrar, Proxy server, Redirect server, Back-to-Back User Agent (B2BUA), Publish Server, Events Server, Back-to-Back Application Framework (B2BAF), and B2BAF Service Library. It provides a rich set of features, such as SIP Server configuration, logging of the application, and resource monitoring.

- The program uses the Tool Command Language/ToolKit (Tcl/Tk) script language for the GUI interface.

Note All references to B2B, Events, LDAP, Accounting, and B2BAF in the Test Application are relevant only to those who acquired the optional SIP Server B2BUA, Events, LDAP, Accounting, and B2BAF Add-on Modules.

TCL/TK

Tcl/Tk is a platform-independent interpreted script language that is used to develop a variety of applications, including GUIs, prototypes and CGI scripts. Tcl also provides an interface into compiled applications such as C and C++. The user application is compiled with Tcl functions, which provide a bi-directional path between Tcl scripts and executable programs.

INSTALLATION

The RADVISION SIP Server Platform includes the SIP Server Test Application. You can use the Test Application either as a working application or as a model of how RADVISION has implemented the SIP Server and API functions.

REMARKS

- Tcl files are located in the *ProxyTestApp\TPX_Tcl\tclSrc* directory, and are not used during runtime.
- You must install Tcl/Tk for working with the SIP Server Test Application. Tcl/Tk version 8.31 can be downloaded from the following website: http://tcl.activestate.com.

SERVERCOMMONAPP LIBRARY

The ServerCommonApp library is an application layer that can be used with different applications to perform the following:

- Load configuration files
- Convert enumerations to strings and strings to enumerations
- Use general utilities used in RADVISION applications

CONFIGURATION

The following files are located in the same directory as the running application files and provide configuration information for the SIP Server Test Application.

sipServerConfig.txt

This file contains parameter information for constructing the SIP Server, including timer configurations, logging levels and network parameters. If the parameter value is not a number or TRUE/FALSE, a comment explains the meaning of the parameter and its optional values.

This file is deprecated as of version 3.0. The *sipServerConfig.xml* file replaces it.

sipServerConfig.xml

This file is a template configuration file to be used as a basis for creating a configuration file for the use of the SIP Server. The name of this file can be passed to the SIP Server during construction time to create the SIP Server with the predefined startup configuration data specified in the file. The attributes in this file correspond to the RvSipServerCfg configuration structure members.

locationDB.txt

This file contains registered users. If the user chooses to save the Location Database Server Component, it is saved in this file and will be loaded the next time the application is run.

securityDB.txt

This file contains the user/password database. The application loads this file at startup.

tpxOptions.val

The graphic user interface (GUI) parameters are saved in this file before shutdown so they can be recalled the next time the application is run.

appConfig.txt

This file contains the Security component, the Location Database component and the Test Application configurations. In this file you can state which configuration filename to use as an argument of RvSipServerMgrConstructFromFile(). By default, the filename is *sipServerConfig.xml*. The application loads this file at startup.

USER INTERFACE

The SIP Server Test Application enables browsing the current active objects in the SIP Server Platform, as well as viewing some of their data members, and manipulating them. This section describes the features of the GUI screens of the Test Application.

Test Application Main Window

Figure 17-1 displays the main window of the Test Application, which provides object control features.

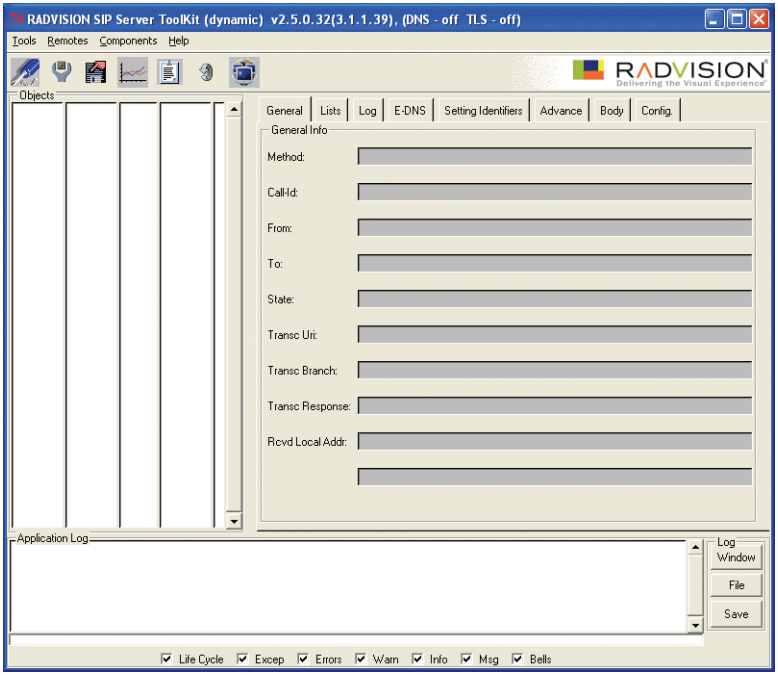


Figure 17-1 *Test Application Main Window*

The window is divided into four areas, each with a different functionality:

- Menu bar and toolbar
- Objects window
- Tab area
- Application log area

Menu Bar and Toolbar

You access some of the features of the SIP Server Platform through the menu bar and the toolbar.

Tools Menu

You can access the following options from the **Tools** menu.

CONFIG

Opens the configuration file. This option is also available from the **Configuration File** button on the toolbar.

Auto Pilot

Opens a window allowing you to set the automated behavior of the SIP Server Test Application. All the values that you select in the **Auto Pilot** window will take effect from the next time that an event occurs, and will have no effect on events that occurred in the past.

What You Can Do in the Auto Pilot Window

You can control the automated behavior of the Test Application by choosing the options described in the following tabs:

- Proxy Tab
- Register Tab
- B2B Tab
- Presence Tab
- Winfo Tab
- Events Tab
- Server Dialog Tab
- Transport Tab
- Publish Tab
- B2BAF Service Tab

PROXY TAB

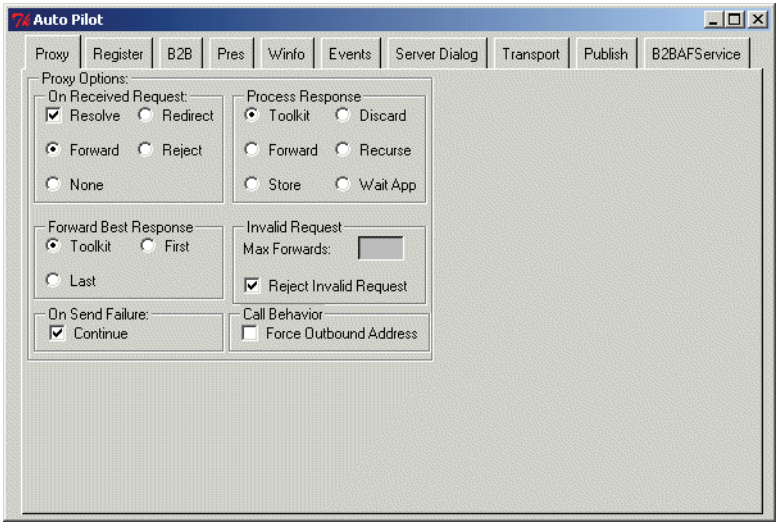


Figure 17-2 *Auto Pilot Window—Proxy Tab*

On Received Request Frame

The parameters in this group determine what to do when a *ProxyCoreObj* receives incoming request.

Resolve

When a *ProxyCoreObj* reaches the REQUEST_VALID state, automatically tries to resolve the address.

Forward

When a *ProxyCoreObj* reaches the ADDRESS_RESOLVED state, automatically forwards the request. The parallel/sequential radio buttons in the **ProxyCoreObj Remote Control** window will determine the proxying mode.

None

When a *ProxyCoreObj* reaches the ADDRESS_RESOLVED state, no automatic action should be performed.

Redirect

When a *ProxyCoreObj* reaches the ADDRESS_RESOLVED state, automatically redirects the request.

Reject

Rejects incoming requests when they are received.

Force outbound Address

Instructs the *ProxyCoreTransc* to send the request to the outbound address regardless of the message content.

Forward Best Response Frame

The parameters in this group control application behavior when the *ProxyCoreObj* did not received a final response for a request forwarded in parallel mode. You can choose from the following:

Toolkit

Forwards the response that the SIP Server recommended. (In general, the lower class response.)

Last

Forwards the last response that the SIP Server received.

First

Forwards the first response that the SIP Server received.

On Send Failure Frame

This parameter determines the automatic behavior when the *ProxyCoreObj* calls the TranscMsgSendFailure() callback or when the *ProxyTrx* reaches the MSG_SEND_FAILURE state.

Continue

If checked, RvProxyCoreTranscDNSContinue() will be called automatically for a stateful *ProxyCoreObj*, and RvProxyCoreTrxSendMsg() will be called for a stateless *ProxyCoreObj*. If not checked, the background of the object that got the callback will change to gray and wait for the user to DNS-continue or DNS-terminate the object. For more information, see E-DNS Tab on page 449.

Process Response Frame

The parameters in this group determine what to do when a *ProxyCoreObj* receives an incoming response.

Toolkit

Accepts the SIP Server recommendation regarding what to do with the response.

Forward

Forwards the incoming response.

Store

Stores the incoming response.

Discard

Discards the incoming response.

Recurse

Recurses on 3xx responses.

Wait App

Waits for the decision of the application.

Invalid Request Frame

The parameters in this group control the behavior of the application when the *ProxyCoreObj* receives an invalid request. You can choose from the following:

Max Forwards

If the invalidation reason was ILLEGAL_MAX_FORWARD, the number in this box will override the number in the original *max forward*.

Reject Invalid Request

If checked, the SIP Server will reject an invalid request. Otherwise, incoming invalid requests will not be rejected.

REGISTER TAB

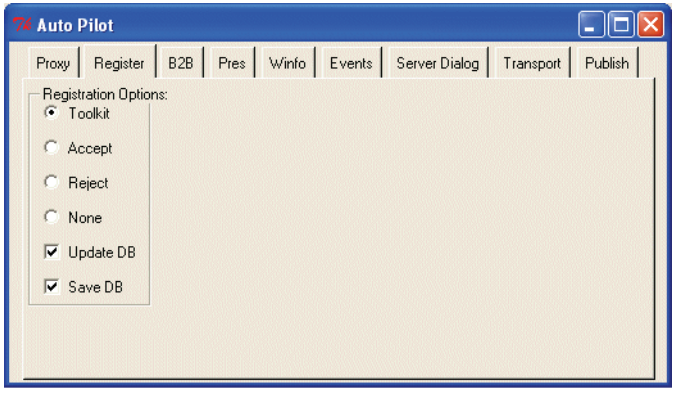


Figure 17-3 *Auto Pilot Window—Register Tab*

Registration Options Frame

The parameters in this group determine the automatic Registrar behavior of the Test Application. You can choose from the following:

Toolkit

Accepts valid incoming registration requests and rejects invalid incoming registration requests.

Accept

Accepts all incoming registration requests.

Reject

Rejects all incoming registration requests.

None

No automatic action will be performed on incoming registration requests.

Update DB

If checked, updates the database with incoming registration requests. If not checked, the database will not be updated even though registrations may be accepted.

Save DB

When checked, saves the Location DB when the application is shut down. When the application is restarted, the Location DB will be reloaded from the saved file.

B2B TAB

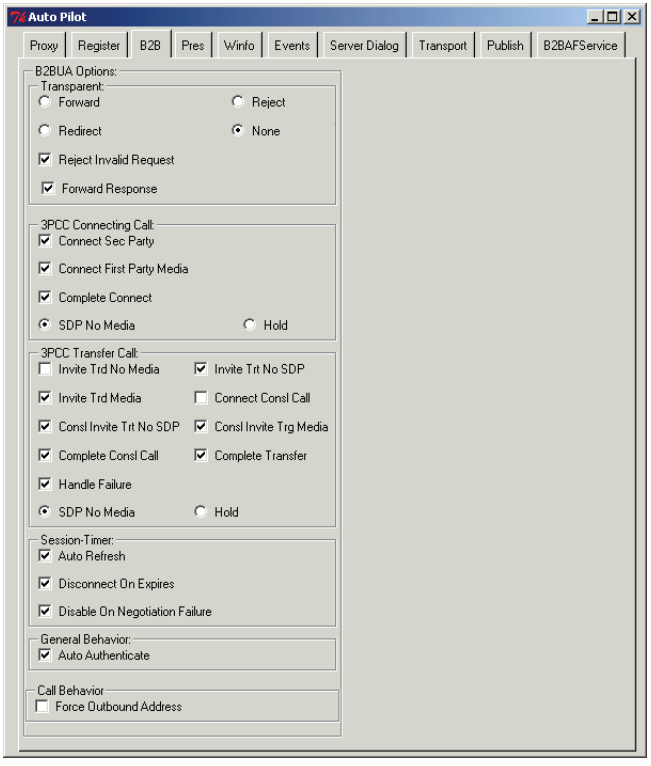


Figure 17-4 *Auto Pilot Window—B2B Tab*

Transparent Frame

The parameters in this group determine the automatic *B2B* behavior of the Test Application while receiving a request. You can choose from the following:

Forward

Once a *B2B* reaches the REQUEST_VALID state, forwards the request to its destination. On an initial request, the SIP Server Test Application looks for the request destination in the Location DB.

Redirect

Redirects all incoming received requests in the context of a *B2B*.

Reject Invalid Request

If checked, the SIP Server will reject an invalid request. Otherwise, incoming invalid requests will not be rejected.

Forward Response

If checked, the SIP Server will forward each received response message. Otherwise, incoming response messages will not be forwarded.

Reject

Rejects all incoming received requests in the context of a *B2B*.

None

No automatic action will be performed on incoming requests.

3PCC Connecting Call Frame

The parameters in this group determine the automatic *B2B* behavior of the Test Application while initiating a 3PCC call between two parties.

Connect Sec Party

If checked, once a 3PCC call sequence is started and a call with the first party is established, sends an INVITE request without an SDP body to the second party.

Connect First Party Media

If checked, once the second party accepted the call (a 2xx response was received), continues with the last part of a 3PCC connect call scenario by sending a re-INVITE request to the first party, containing the media capability of the second party.

Complete Connect

If checked, completes the 3PCC call-connected scenario by sending an ACK request to the remote parties of the call.

SDP No Media

Indicates that an initial INVITE request will contain an SDP body with no media when a 3PCC connect call is started and the initial INVITE request needs to be sent containing an SDP body with no media.

Hold

Indicates that an initial INVITE request will contain a hold SDP body when a 3PCC connect call is started and the initial INVITE request needs to be sent containing an SDP body with no media (for compliancy with UAs that do not support SDP bodies with no media).

Session-Timer Frame

Auto Refresh

Sends a Session-Timer refresh when the session is about to expire and the *B2BDlg* is the refresher. The refresh request type (re-INVITE/UPDATE) to be sent is determined on the **Advance** tab.

Disconnect on Expires

Initiates a BYE request when the Session-Timer expires and the remote party of the *B2BDlg* is the refresher.

Disable On Negotiation Failure

Disables the Session-Timer mechanism when the remote party of the *B2BDlg* does not support the Session-Timer or when it changed the Session-Timer preferences.

General Behavior Frame

Auto Authenticate

Re-sends the request with the request credential.

Call Behavior Frame

Force Outbound Address

Instructs the *B2BDlg* to send a request to the outbound address regardless of the message content.

3PCC Transfer Call Frame

The parameters in this group determine the automatic *B2B* behavior of the Test Application while transferring a call to a third party, in blind or consultation mode, using the 3PCC scenarios.

Invite Trd No Media

If checked, starts the connection of the transferred call. On blind transfer, once the Transferring (TRG) is disconnected, sends a re-INVITE request with an SDP body with no media to the Transferred (TRD). On consultation transfer, once the consultation call is complete, sends a re-INVITE request with an SDP body without media, or a hold SDP body, to the TRD.

Invite Trd Media

If checked, completes the connection of the transferred call. On blind transfer, once the Transferred-To (TRT) accepts the INVITE, sends a re-INVITE request with the media capability of the TRT to the TRD. On consultation transfer, once the TRT accepts the re-INVITE, sends a re-INVITE request with the media capability of the TRT to the TRD.

Consl Invite Trt No SDP

If checked, on consultation transfer, continues connecting the consultation call. Once the TRG accepts the re-INVITE request and an ACK was sent, sends an INVITE request to the TRT without and SDP body.

Complete Consl Call

If checked, on consultation transfer, sends an ACK request to the TRT and the TRG to complete the connection of the transferred call.

Handle Failure

If checked, handles a transfer failure based on the *B2B* returned failure option.

Invite Trt No SDP

If checked, continues connecting the transferred call. On blind transfer, once the TRD accepts the re-INVITE and an ACK was sent to complete the re-INVITE sequence, an INVITE request without an SDP body is sent to the TRT. On consultation transfer, once the TRD accepts the re-INVITE request and an ACK is sent to complete the re-INVITE sequence, a re-INVITE request without an SDP body is sent to the TRT.

Connect Consl Call

If checked, on consultation transfer, starts the connection of the consultation call. Once the TRD was put on hold, sends a re-INVITE request containing an SDP body with no media, or a hold SDP body, to the TRG.

Consl Invite Trg Media

If checked, on consultation transfer, completes the connection of the consultation call. Once the TRT accepts the INVITE, sends a re-INVITE request with the media capability of the TRT to the TRG.

Complete Transfer

If checked, sends an ACK request to the TRT and the TRD to complete the connection of the transferred call.

SDP No Media

Indicates that an initial INVITE request will contain an SDP body with no media when a 3PCC transfer call is started and the initial INVITE request needs to be sent containing an SDP body with no media.

Hold

Indicates that an initial INVITE request will contain a hold SDP body when a 3PCC transfer scenario is started and the initial INVITE request needs to be sent containing an SDP body with no media (for compliancy with UAs that do not support SDP bodies with no media).

Note The above parameters are enabled only if the SIP Server Platform package contains the B2BUA add-on module.

How to Connect a 3PCC Call Between Two Parties

To connect a 3PCC Call

1. Select the **B2BUA** radio button in the **Setting Identifiers** tab to display the **B2BUA** options, and then check **Apply** in the **Setting Identifiers/B2BUA** tab.
2. Enter the source and destination of the first party in the **From** and **To** text boxes, and the source and destination of the second party in the **Sec Party From** and **Sec Party To** text boxes.
3. Select **Auto Pilot** from the **Tools** menu and then select the **B2B** tab in the **Auto Pilot** window.
4. Check **Connect Sec Party**, **Connect First Party Media** and **Complete Connect** in the **Connecting Call** frame of the **B2B** tab in the **Auto Pilot** window.
5. In the same frame, select the **SDP No Media** radio button if the first party supports SDP with no media. Otherwise, select the **Hold** radio button.
6. Select **B2BUA** from the **Remotes** menu to open the **B2BUA Remote Control** window, and select **CreateB2B** to create a new *B2B*.
7. Check **Auto ACK** and **Auto Make** in the **Connect** frame of the **B2BUA Remote Control** window.
8. Press **Connect 2 P** to connect the 3PCC call.

How to Transfer a Call (Blindly) to a Third Party

To transfer a call

1. Select the **B2BUA** radio button in the **Setting Identifiers** tab to display the **B2BUA** options, and then check **Apply** in the **Setting Identifiers/B2BUA** tab.
2. Enter the source and destination of the TRT in the **Trt From** and **Trt To** text boxes.
3. Select **Auto Pilot** from the **Tools** menu and then select the **B2B** tab in the **Auto Pilot** window.
4. Check **Invite Trt Media, Invite Trt No SDP** and **Complete Transfer** in the **Transfer Call** frame of the **B2B** tab in the **Auto Pilot** window.
5. In the same frame, select the **SDP No Media** radio button if the TRT supports SDP with no media. Otherwise, select the **Hold** radio button.
6. In the main window, select a "connected" *B2B* with two active *B2BDlg* objects. Right click on one of the *B2BDlg* objects and select **Select Transferring** from the floating menu.
7. Select **B2BUA** from the **Remotes** menu to open the **B2BUA Remote Control** window, and select the **Blind** radio button in the **Transfer** frame**.**
8. In the same frame, check **Auto ACK** and **Auto Make**.
9. Press the **Transfer** button to transfer the call a third party.

PRESENCE TAB

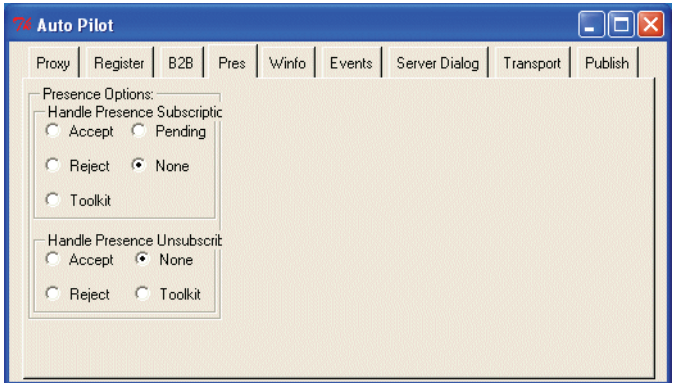


Figure 17-5 *Auto Pilot Window—Presence Tab*

Handle Presence Subscription Frame

This parameter determines the automatic behavior of a *PresSubscribe* while receiving SUBSCRIBE request.

Accept

Accepts an incoming Presence SUBSCRIBE request.

Pending

Answers with a Pending response to an incoming Presence SUBSCRIBE request.

Reject

Rejects the incoming Presence SUBSCRIBE request.

None

No automatic action will be performed on incoming Presence SUBSCRIBE requests.

Toolkit

Accepts incoming valid Presence SUBSCRIBE requests and rejects invalid Presence SUBSCRIBE requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

Handle Presence UnSubscribe Frame

This parameter determines the automatic behavior of the *PresSubscribe* while receiving an Unsubscribe message.

Accept

Accepts an incoming Presence Unsubscribe request.

Reject

Rejects the incoming Presence Unsubscribe request.

None

No automatic action will be performed on the incoming Presence Unsubscribe request.

Toolkit

Accepts incoming valid Presence Unsubscribe requests and rejects invalid Presence Unsubscribe requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

WINFO TAB

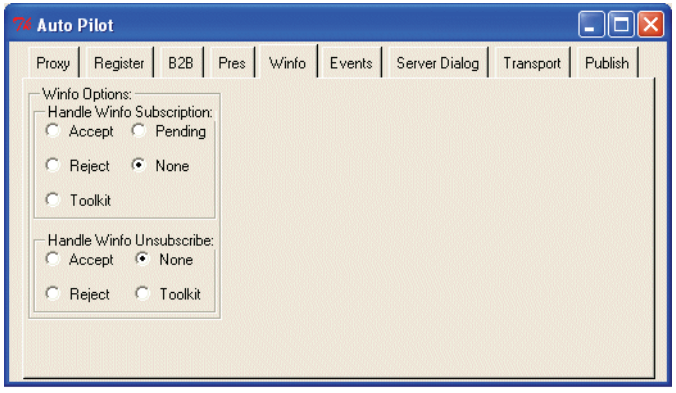


Figure 17-6 *Auto Pilot Window—Winfo Tab*

Handle Winfo Subscription Frame

This parameter determines the automatic behavior of a *WinfoSubs* while receiving a Winfo SUBSCRIBE request.

Accept

Accepts incoming Winfo SUBSCRIBE requests.

Pending

Answers with a Pending response to an incoming Winfo SUBSCRIBE request.

Reject

Rejects the incoming Winfo SUBSCRIBE request.

None

No automatic action will be performed on incoming Winfo SUBSCRIBE requests.

Toolkit

Accepts incoming valid Winfo SUBSCRIBE requests and rejects invalid Winfo SUBSCRIBE requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

Handle Winfo UnSubscribe Request Frame

This parameter determines the automatic behavior of the *WinfoSubs* while receiving an Unsubscribe message.

Accept

Accepts an incoming Winfo Unsubscribe request.

Reject

Rejects the incoming Winfo Unsubscribe request.

None

No automatic action will be performed on the incoming Winfo Unsubscribe request.

Toolkit

Accepts incoming valid Winfo Unsubscribe requests and rejects invalid Winfo Unsubscribe requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

EVENTS TAB

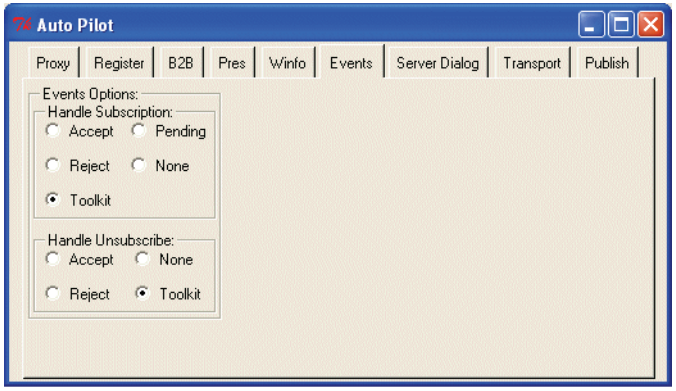


Figure 17-7 *Auto Pilot Window—Events Tab*

Handle Subscription Frame

This parameter determines the automatic behavior of an *EventsSubs* while receiving a SUBSCRIBE request.

Accept

Accepts an incoming SUBSCRIBE request.

Pending

Answers with a Pending response to an incoming SUBSCRIBE request.

Reject

Rejects the incoming SUBSCRIBE request.

None

No automatic action will be performed on incoming SUBSCRIBE requests.

Toolkit

Accepts incoming valid SUBSCRIBE requests and rejects invalid SUBSCRIBE requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

Handle UnSubscribe Frame

This parameter determines the automatic behavior of the *EventsSubs* while receiving an Unsubscribe request.

Accept

Accepts an incoming Unsubscribe request.

Reject

Rejects the incoming Unsubscribe request.

None

No automatic action will be performed on the incoming Unsubscribe request.

Toolkit

Accepts incoming valid Unsubscribe requests and rejects invalid requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

SERVER DIALOG TAB

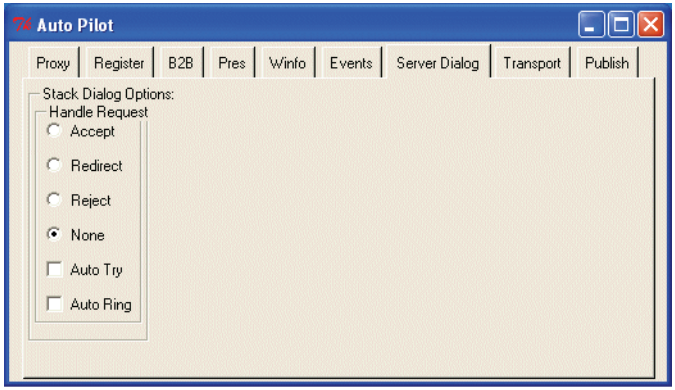


Figure 17-8 *Auto Pilot Window—Server Dialog Tab*

Handle Request Frame

This parameter determines the automatic behavior of a *CallLeg* while receiving a request message.

Accept

Accepts any incoming request when working on dialog level.

Redirect

Redirects any incoming request when working in dialog OWNER_APP mode.

Reject

Rejects any incoming request when working on dialog level.

None

Does not perform anything automatic with incoming requests when working in dialog OWNER_APP mode.

Auto Try

If checked, the SIP Server will initiate a 100 “trying” provisional response to any incoming INVITE request.

Auto Ring

If checked, the SIP Server will initiate a 180 “Ringing” provisional response to any incoming INVITE request.

Note The above parameters are enabled only if the SIP Server Platform package contains the B2BUA add-on module.

TRANSPORT TAB

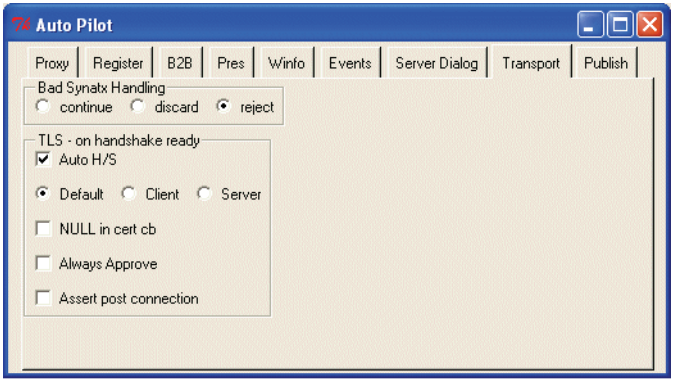


Figure 17-9 *Auto Pilot Window—Transport Tab*

Bad Syntax Handle Frame

Determines the automatic behavior of the SIP Server when handling a message with bad syntax.

Continue

Continues processing the message.

Discard

Discards the message.

Reject

Rejects the request automatically.

TLS—on handshake ready Frame

Sets the automatic behavior of the SIP Server when using TLS connections, in the HANDSHAKE_READY state.

Auto H/S

Automatically performs a TLS handshake in the HANDSHAKE_READY state.

Note Checking **Auto H/S** enables the **Default**, **Client** and **Server** radio buttons as well as the **NULL in cert cb** check box.

Default/Client/Server

Determines the handshake side of the TLS connection.

NULL in cert cb

If checked for the client, assigns NULL to the TLS certificate callback (of the RvSipTransportVerifyCertificateEv()). If checked for the server, this means that the server does not require a mutual handshake.

Always Approve

The application overrides certificate is always approved.

Assert post connection

Asserts the post connection successfully.

PUBLISH TAB

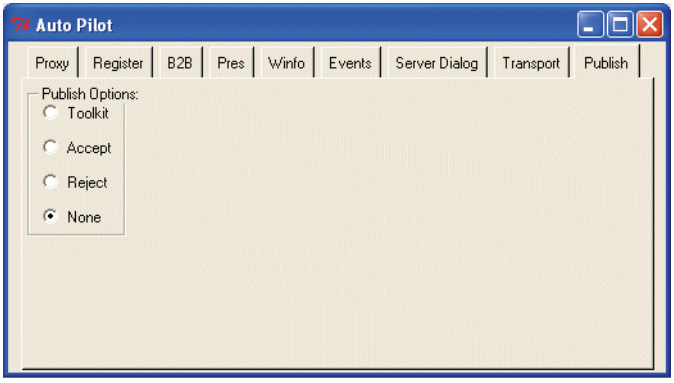


Figure 17-10 *Publish Tab*

Publish Options Frame

Toolkit

Accepts incoming valid Publish requests and rejects invalid requests.

Accept

Accepts incoming Publish requests.

Reject

Rejects incoming Publish requests.

None

No automatic action will be performed on the incoming Publish requests.

Note The above parameters are enabled only if the SIP Server Platform package contains the Events add-on module.

B2BAF Service Tab

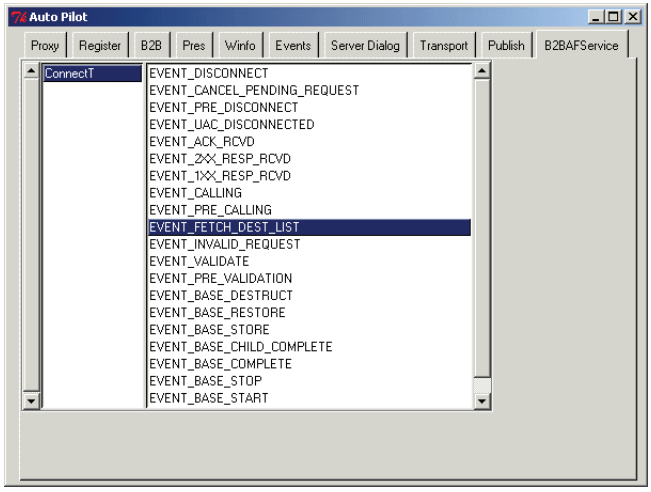


Figure 17-11 *B2BAF Service Tab*

Service column

(Left-hand column) Represents the supported services of the application.

Event column

(Right-hand column) Represents the events supported for the selected service. When an event is selected, the application service will override the parent event handler and will handle the event by displaying the event name. The application user can de-select the event and fire the same event over again, using the Service remote control to continue the parent service flow.

XML

(This option appears after **Auto Pilot** on the **Tools** menu.)

The XML is a GUI interface for the XML Encoder Library, and for the XML Parser. You can create and parse any kind of XML document using this interface.

The XML window is divided into three main parts, as illustrated in Figure 17-12:

- XML Tags
- Text
- Commands

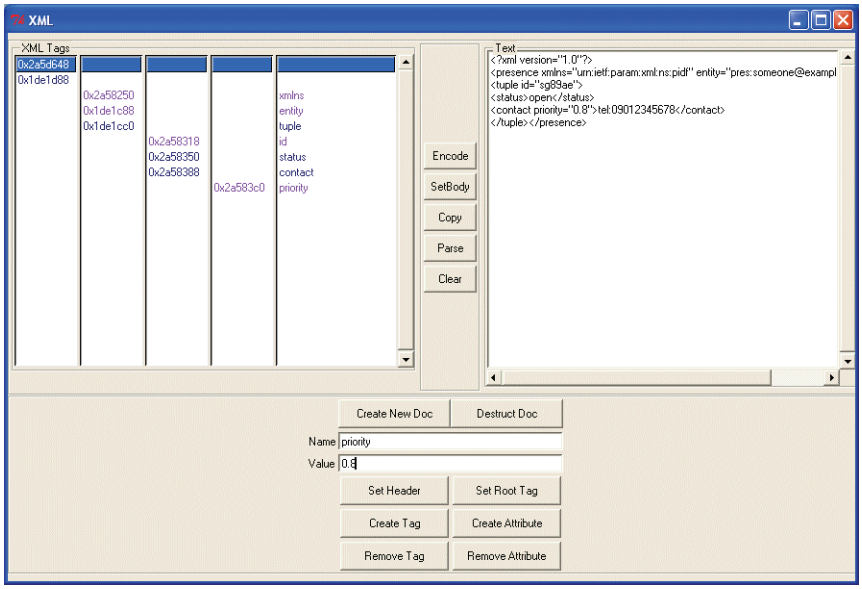


Figure 17-12 *XML Window*

XML Tags

Displays all the XML objects that were created, in the form of a tree. An *XMLDoc* is the first object in the window (in black) with its root *XMLTag* underneath the *XMLDoc* (in blue). All other *XMLTag* objects that were created will appear under their XML father tag. All *XMLAttr* objects will appear under their *XMLTag* (in purple). The **XML Tags** tree is limited to a depth of three.

Text

Displays the encoded XML document, or the text that needs to be parsed.

Command Buttons

Encode

Encodes an XML tree to a document.

Set Body

Copies an encoded document to the **Body** tab in the main window.

Command Buttons

Copy

Copies an *XMLDoc*. After copying the *XMLDoc*, a new *XMLDoc* is created. To copy an *XMLDoc*, you need to select the *XMLDoc* of the document that needs to be copied and press **Copy**.

Parse

Parses an *XMLDoc* into an XML tree. To parse an *XMLDoc*, place the text that needs to be parsed on the Text area of the screen, and press **Parse**.

Clear

Clears the Text area of the screen.

Create New Doc

Creates a new *XMLDoc*. You can edit a few *XMLDoc* objects at the same time if needed. When creating a new *XMLDoc*, two new objects will appear in the **XML Tags** window—the first is the *XMLDoc* (in black) and the second is the root *XMLTag* (in blue).

Destruct Doc

Destructs an *XMLDoc* and all its objects.

Name

A text box for setting the name of an object (*XMLDoc* header, root tag, *XMLTag* or *XMLAttr*).

Value

A text box for setting the value of an object (*XMLDoc* header, root tag, *XMLTag* or *XMLTag*).

Set Header

Uses the text in the **Name** and **Value** text boxes, and sets the *XMLDoc* header with these values.

Set Root Tag

Uses the text in the **Name** and **Value** text boxes, and sets the root *XMLTag* with these values.

Create Tag

Creates a new *XMLTag*, whose parent is the selected *XMLTag*. The new *XMLTag* will appear under its parent at a lower level. The created tag will be set with the name and value from the **Name** and **Value** text boxes.

Create Attribute

Creates a new *XMLAttr* under the selected *XMLTag*. The new *XMLAttr* will appear under its tag at the same level. The created attribute will be set with the name and value from the **Name** and **Value** text boxes.

Remove Tag

Removes an *XMLTag* together with all its child *XMLTag* and *XMLAttr* objects.

Remove Attribute

Removes an *XMLAttr* from an *XMLTag*.

How To Compose an *XMLDoc*

The following steps will show how to create the *XMLDoc* that appears in Figure 17-12 on page 406.

To create an XML document

1. Press the **Create New Doc** button to create a new *XMLDoc*. After the document is created, two new objects will appear in the **XML Tags** window. The first is the *XMLDoc*, and the second is the root tag object.
2. Type “presence” in the **Name** text box. Select the *XMLDoc* object, and press the **Set Root Tag** button.

3. Type “xmlns” in the **Name** text box, and “urn:ietf:params:xml:ns:pidf” in the **Value** text box. Select the root tag object, and press the **Create Attribute** button.
4. Type “entity” in the **Name** text box, and "pres:someone@example.com" in the **Value** text box. Select the root tag object, and press the **Create Attribute** button.
5. Type “tuple” in the **Name** text box (leave the **Value** text box empty). Select the root tag object, and press the **Create Tag** button.
6. Type “id” in the **Name** text box, and “sg89ae” in the **Value** text box. Select the newly created “tuple” tag object, and press the **Create Attribute** button.
7. Type “status” in the **Name** text box, and “open” in the **Value** text box. Select the “tuple” tag object, and press the **Create Tag** button.
8. Type “contact” in the **Name** text box, and “tel:09012345678” in the **Value** text box. Select the “tuple” tag object, and press the **Create Tag** button.
9. Type “priority” in the **Name** text box, and “0.8” in the **Value** text box. Select the newly created “contact” tag object, and press the **Create Attribute** button.
10. Select the *XMLDoc* object, and press the **Encode** button. The encoded *XMLDoc* will appear in the **Text** window.

STATUS

(This option appears after **XML** on the **Tools** menu.)

Displays the **Resources Status** window.

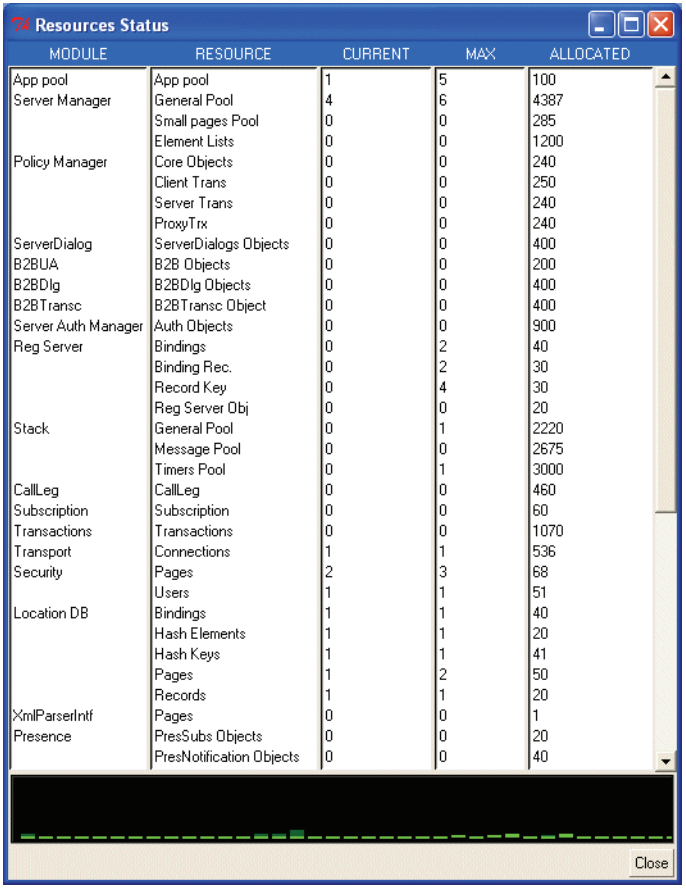


Figure 17-13 *Resources Status Window*

In this window, you can see the number of allocated elements for each resource. This option is also available from the **Status** button on the toolbar.

MISCELLANEOUS

(This option appears after **Status** on the **Tools** menu.)

Opens a window that lets you inject a message into the server and select the editor to be used for opening the log file.

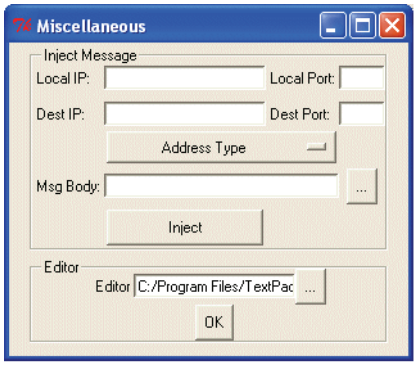


Figure 17-14 *Miscellaneous Window*

Inject Message

Injects a message to the server from a text file. The local IP and port will be used for injecting the message as if it was sent from this address to the destination IP and port. The type of transport used for sending the message can also be selected.

After setting the address, the path to the text file containing the message to be injected should be typed in the **MsgBody** parameter. Pressing the **Inject** button injects the message set to the server.

Editor

Selects the editor to use for opening the log and configuration files.

LOG FILE

(This option appears after **Miscellaneous** on the **Tools** menu.)

Opens the log file. This option is also available from the **Log File** button on the toolbar.

LOG SETTINGS

Opens a window enabling you to change logging levels during runtime.

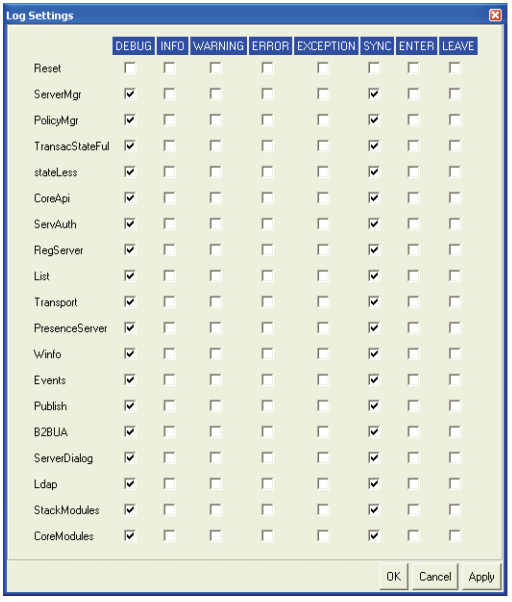


Figure 17-15 *Log Settings Window*

This window is built as a matrix of modules and logging levels. Selecting a button for a certain module and a certain level enables logging for that module and level. The eight topmost check buttons will reset an entire logging level. To save the changes you have made, press **OK** or **Apply**.

This option is also available from the **Log Settings** button on the toolbar.

KILL ALL

(This option appears after **Log Settings** on the **Tools** menu.)

Terminates all the active objects of the SIP Server. Pressing **Ctrl** + **K** on the keyboard also terminates the active objects.

STORE ALL

Stores all connected B2B and Events objects on the specified file name from the B2BUA, Events or Publish remote controls. See Remotes Menu on page 413.

RESTORE ALL

Restores all stored objects from the specified file name from the B2BUA, Events or Publish remote controls. See Remotes Menu on page 413.

EXIT

Exits the application.

TOOLBAR OPTIONS

You can perform additional operations by clicking on the following toolbar icons

Restart

Restarts the Test Application. When the application is restarted, it reloads its configuration files.

Network Configuration

Opens the **Network Configuration** window which displays the network parameters that were configured in the configuration files (such as the **Domain List** and **Local Addresses**).

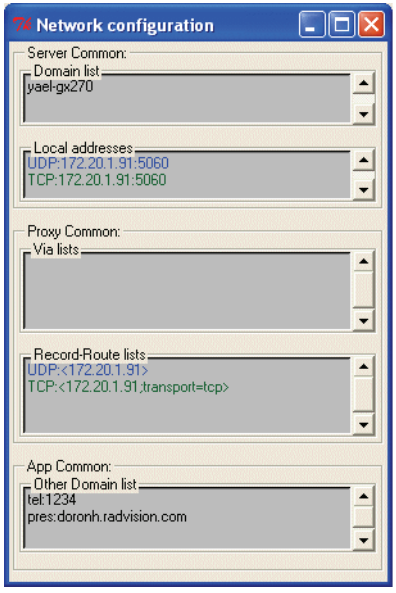


Figure 17-16 *Network Configuration Window*

REMOTES MENU

The SIP Server Test Application remote controls allow you to perform non-automatic actions on a selected object. Each object type has its own dedicated remote control. You can open the object-related remote control in two ways:

- Select the requested remote type from the **Remote** menu bar.

- Double-click a selected object and its related remote control will pop-up.

Note You can press any of the buttons in any of the states. For an action to actually take place on an object, the object must be in a state where calling a function has a meaning. For example, selecting Resolve when a *ProxyCoreObj* is in the ADDRESS_RESOLVED state will have no effect.

The following remote controls are available, as described in the sections below:

- B2BUA Remote Control
- B2BDlg Remote Control
- Core Object Remote Control
- Registrar Remote Control
- Events Remote Control
- Stack Dialog Remote Control
- Connection Remote Control
- Publish Remote Control
- LDAP Remote Control
- Presence Agent Remote Control
- B2BAF Service Remote Control

B2BUA Remote Control

You can perform the non-automatic actions on *B2B*, *B2BDlg* and *B2BTransc* objects by choosing the options described under the **B2BUA Remote Control** window.

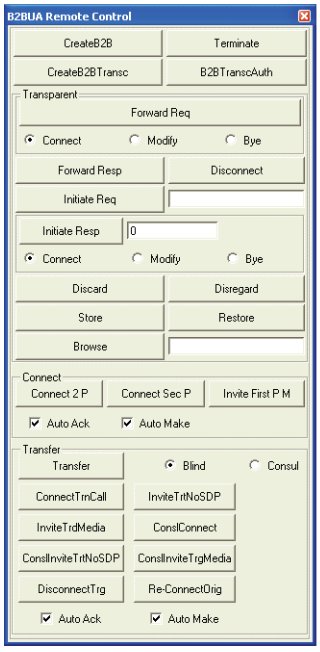


Figure 17-17 *B2BUA Remote Control Window*

CreateB2B

Creates a *B2B*.

Terminate

Terminates the selected *B2B*.

CreateB2BTransc

Creates a *B2BTransc* related to the selected *B2B*. To create a *B2BTransc*, you must select its *B2BDlg* “owner”.

B2BTranscAuth

Re-sends the general request with the request credential.

Transparent Frame

Forward Request

Forwards the request message to its destination. The radio buttons under the **Forward Request** button determine which request will be forwarded (**Connect**, **Modify** or **Bye**). To forward a general request, select the *B2BTransc* regardless of the radio button that is selected.

Forward Response

Forwards the selected response message to its destination. To forward a response message, you must select it in the **Response Message** frame in the **Lists** tab.

Disconnect

Disconnects the *B2B* call (sends a BYE request on all the legs of the *B2B*).

Initiate Request

Initiates any request using the specified method written in the text box to the right of the button on the selected *B2B* or *B2BTransc*.

Initiate Response

Initiates a response using the specified response code written in the text box to the right of the button on the selected *B2B* or *B2BTransc*. The radio button under the **Initiate Response** button determines to which state machine the response will be sent (**Connect**, **Modify**, **Bye** or B2BTransc).

Discard

Discards the selected response message. To discard a received response message, you must select it in the **Response Message** frame in the **Lists** tab.

Disregard

Disregards the validation failure.

Store

Stores the selected *B2B* in the file specified in the text box to the right of the Browse button.

Restore

Restores a *B2B* from a file specified in the text box to the right to the Browse button.

Browse

Browses for a Store or Restore file and displays the selected file in the text box to the right of the button.

Connect Frame

Connect 2 P

Starts the 3PCC connect call scenario. If **Auto Make** is checked, you must first specify the To and From headers of the first and the second parties on the setting identifiers tab under *B2B* section. Otherwise, you must first set the To and From headers of the first party on the **Setting Identifiers** tab under the *B2BDlg* section and create a *B2BDlg*.

Connect Sec P

Invites the second party to participate in the 3PCC call. If **Auto Make** is not checked, you must first set the To and From headers of the second party on the **Setting Identifiers** tab under the *B2BDlg* section and create a *B2BDlg*.

Invite First PM

Continues with the final part of a 3PCC connect call scenario by sending a re-INVITE request to the first party that contains the media capability of the second party.

Auto ACK

If checked, indicates that the *B2B* will automatically send an ACK when a 2xx response is received.

Auto Make

If checked, indicates that the *B2B* will automatically create the call parties.

Transfer Frame

Transfer

Starts the 3PCC call transfer scenario. The radio button to the right of the Transfer button determines which type of transfer will be performed (**Blind** or **Consultation**). To start the transfer scenario, you must first select the *B2BDlg* that will act as the Transferring entity, then right-click the *B2BDlg* and select **Select Transferring** from the floating menu.

ConnectTrnCall

Starts connecting the transferred call.

InviteTrtNoSDP

Continues connecting the transferred call. On blind transfer, if **Auto Make** is checked, you must first specify the To and From headers of the TRT in the **B2B** frame of the **Setting Identifiers** tab. Otherwise, you must first set the To and From headers of the first party in the **B2BDlg** frame of the **Setting Identifiers** tab, create a *B2BDlg*, select and right-click it, and then select **Select Transferred-To** from the floating menu.

InviteTrdMedia

Completes the connection of the transferred call.

ConsIConnect

Starts the connection of the consultation call.

ConsIInviteTrtNoSDP

Continues connecting the consultation call. If **Auto Make** is checked, you must first specify the To and From headers of the TRT in the **B2B** frame of the **Setting Identifiers** tab. Otherwise, you must first set the To and From headers of the first party in the **B2BDlg** frame of the **Setting Identifiers** tab, create a *B2BDlg*, select and right-click it, and then select Select **Transferred-To** from the floating menu.

ConsIInviteTrgMedia

Completes the connection of the consultation call.

DisconnectTrg

Performs the last phase of the consultation transfer and disconnects the TRG.

Re-ConnectOrig

Reconnects the original call between the first and second party on transfer failure.

Auto ACK

If checked, indicates that the *B2B* will automatically send an ACK when a 2xx response is received.

Auto Make

If checked, indicates that a *B2B* will automatically create the TRT.

Adding a CallLeg as a B2BDlg

The *B2B* allows you to add a connected *CallLeg* as one of its *B2BDlg* objects. To do so, select a *CallLeg* and mark it by right-clicking the mouse and selecting the **Mark** option (the object background color will turn orange). After a *CallLeg* has been marked, you need to select a *B2B* and by right-clicking the mouse and selecting the **Add** option, you add the selected *CallLeg* to the *B2B*.

B2BDlg Remote Control

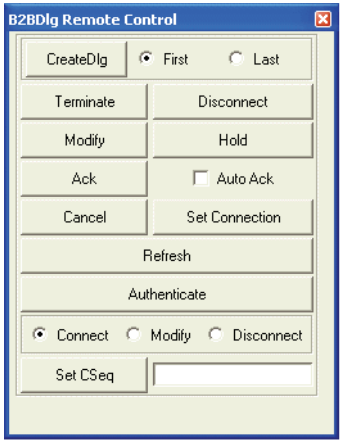


Figure 17-18 *B2BDlg Remote Control Window*

CreateDlg

Creates a *B2BDlg*, The radio button to the right of the **CreateDlg** button determines whether the created *B2BDlg* is the **First** or **Last** on the list.

Terminate

Terminates the selected *B2BDlg*.

Disconnect

Disconnects the *B2BDlg* call (sends a BYE request).

Modify

Initiates re-INVITE request.

Hold

Initiates re-INVITE request with a hold SDP body.

Ack

Initiates an ACK request.

Auto ACK

If checked, indicates that the *B2BDlg* will automatically send an ACK when a 2xx response is received.

Cancel

Initiates a Cancel request.

Set Connection

Sets the selected connection to a persistent *B2BDlg* object.

Refresh

Sends a Session-Timer refresh when the session is about to expire and the *B2BDlg* is the refresher. The refresh request type (re-INVITE/UPDATE) to be sent is determined on the **Advance** tab.

Authenticate

Re-sends the Connect/Modify/Disconnect request with the request credential.

Set CSeq

Sets the CSeq number of the restored *B2BDlg* to the value specified in the text box to the right of the button.

Core Object Remote Control

You can perform the non-automatic actions on a *ProxyCoreObj* by choosing the options described under the **CoreObj Remote Control** window.

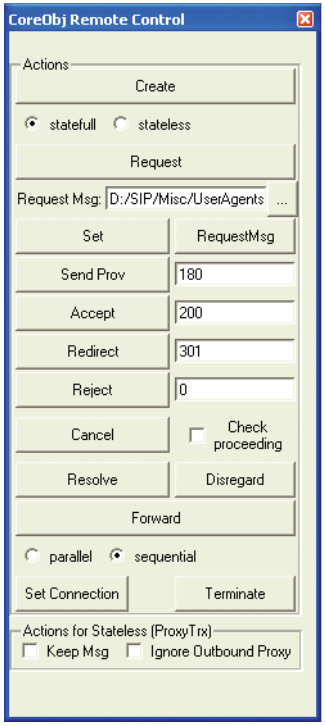


Figure 17-19 *CoreObj Remote Control Window*

Actions Frame

Create

Creates a *ProxyCoreObj*. The radio button below the **Create** button determines whether the created *ProxyCoreObj* is of the stateless or stateful type. This *ProxyCoreObj* will be used to send an out-of-dialog message.

Request

Forms and sends a request from the **Setting Identifiers** tab information. Can be used for stateful *ProxyCoreObj* only.

Set

Sets the message file, as stated next to the Request Msg text box, to the selected *ProxyCoreObj* which you created. To send the message use the **Request Msg** button.

Request Msg

Sends the message you have set.

Send Prov

Initiates a provisional response.

Accept

NA

Redirect

Redirects an object with the response code written in the field to the right of the button.

Reject

Rejects an object with the response code written in the field to the right of the button.

Cancel

Sends a Cancel on an object. For a transaction it, cancels on the selected transaction. For a transaction stateful core object, it cancels on all its client transactions. If the **Check Proceeding** button is checked, the SIP Server checks if a transaction is proceeding before canceling the transaction.

Resolve

Resolves the addresses of a core object consulting with the database. Pressing this button usually moves the core object to the ADDRESS_RESOLVED state.

Disregard

Disregards the incoming request validation failure.

Forward

Forwards a request in parallel or sequential mode. The radio button under the **Forward** button determines the proxying mode (**parallel** or **sequential**).

Note Stateless objects can only be forwarded sequentially.

Set Connection

Sets a connection to a persistent *ProxyCoreObj*. The connection that is set will be either the connection of a marked object (to mark an object, right-click on the object and select **Mark**), or the selected connection (a connection that was selected with the **Select** command).

Terminate

Terminates an object.

Actions for Stateless (ProxyTrx)

Keep Msg

When this flag is on, the *ProxyTrx* will not destruct the message after being sent, thus allowing the application to call the RvProxyCoreTrxSendMsg() function with NULL as the message in the MSG_SEND_FAILURE state.

Ignore Outbound Proxy

When this flag is on, the *ProxyTrx* will ignore its outbound address and send the request according to the request information.

Registrar Remote Control

You can perform the non-automatic actions on a *RegServerObj* by choosing the options described under the **REG Remote Control** window.

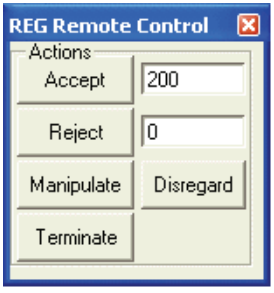


Figure 17-20 *REG Remote Control Window*

Accept

Accepts an object with the response code written in the field to the right of the button.

Reject

Redirects an object with the response code written in the field to the right of the button.

Manipulate

Opens the **Reg Server Manipulation** window that lets you change the binding record of the *RegServerObj* before accepting the registration. (See Reg Server Manipulation Window on page 425.)

Disregard

Disregards the incoming REGISTER request validation failure.

Terminate

Terminates the *RegServerObj*.

REG SERVER MANIPULATION WINDOW

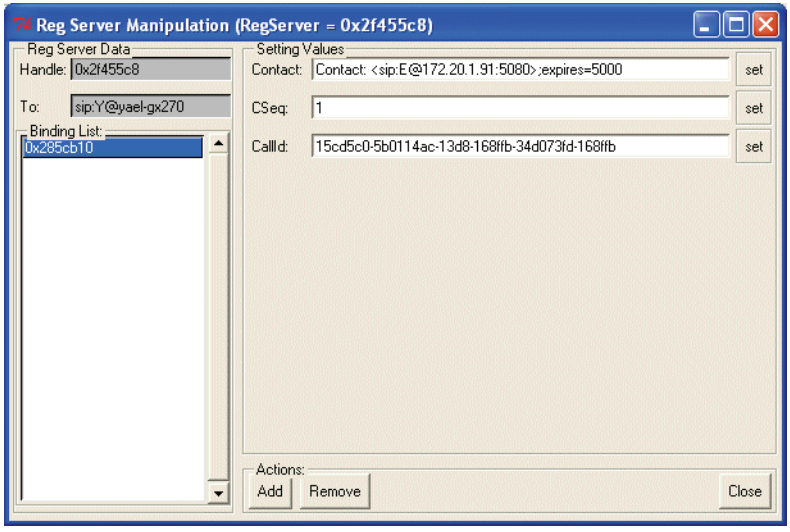


Figure 17-21 *Reg Server Manipulation Window*

The **Reg Server Manipulation** window includes the following parameters and command buttons:

RegServer Data

Handle

Displays the handle of the manipulated *RegServer*.

To

Displays the To header of the Registration request.

Binding List

Displays a list of the records that are bound to the specific Registration request. Selecting any of the bindings will display its data in the **Setting Values** frame.

Setting Values Frame

Contact

Displays the Contact of the selected binding.

CSeq

Displays the CSeq of the selected binding.

Call-ID

Displays the call-ID of the selected binding.

Set

Selecting **Set** for any of the values (Contact, CSeq, Call-ID) will set the value in the selected record.

Command Buttons

Add

Adds a binding containing the triplet inserted as Contact, CSeq and Call-ID.

Remove

Removes the selected binding from the list.

Close

Closes the window. If the *RegServerObj* terminates when this window is open, the window will be closed automatically.

EVENTS REMOTE CONTROL

You can perform the non-automatic actions on a Events subscription by choosing the options described under the **Events Remote Control** window. The **Events Remote Control** controls all the events objects: *PresSubscribe*, *WinfoSubs*, *EventsSubs*, *PresNotification*, *WinfoNotify*, and *EventsNotify*.

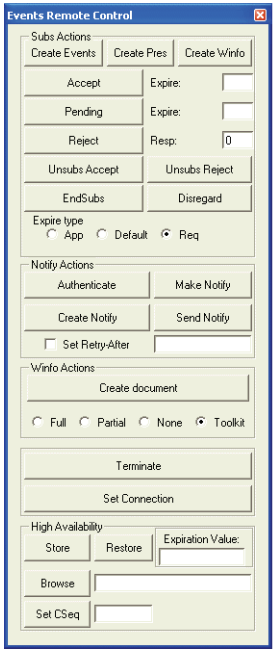


Figure 17-22 *Events Remote Control Window*

Subs Actions Frame

Create Events

Creates a new Events object for restoring an Events object from a file.

Create Pres

Create a new Presence object for restoring a Presence object from a file.

Create Winfo

Create a new Winfo object for restoring a Winfo object from a file.

Accept

Accepts a SUBSCRIBE request (or a Refresh request) handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object, with the expiration value written in the text box next to the button, which will be sent in the 200 response.

Pending

Answers with a Pending response for a SUBSCRIBE request handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object, with the expiration value written in the text box next to the button, which will be sent in the 202 response.

Reject

Rejects the SUBSCRIBE request (or a Refresh request) handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object, with the response code written in the text box next to the button.

UnsubsAccept

Accepts an Unsubscribe request handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object.

UnsubsReject

Rejects an Unsubscribe request handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object.

EndSubs

Ends a subscription handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object by sending a NOTIFY (terminated) request.

Disregard

Disregards a failure found while validating or authenticating a request handled by the selected *PresSubscribe*, *WinfoSubs* or *EventsSubs* object.

Expire Type

Selects the expire value which should be used when accepting or “pending” a subscription. The options are:

- **App**—the value in the text box next to the button will be used.

- **Default**—the value will be taken from the configuration.
- **Req**—the value will be taken from the received request.

Notify Actions Frame

Authenticate

Authenticates a NOTIFY request that received a 401/407 response.

Make Notify

Creates a *PresNotification, WinfoNotify* or *EventsNotify* object and sends a NOTIFY (active) request.

Create Notify

Creates a new *PresNotification, WinfoNotify* or *EventsNotify* object.

Send Notify

Sends the NOTIFY request prepared by the selected *PresNotification, WinfoNotify* or *EventsNotify* object.

Set Retry-After

When checked, a NOTIFY (terminated) with the "probation" or "giveup" reason will be sent with a Retry-After parameter with the value set in the text box next to the check box.

Winfo Actions Frame

Create Document

Creates a Winfo document for a *WinfoNotify* that the application created. The created Winfo document will be displayed in the **Body** tab in the main window. To send the *WinfoNotify* with the document that was created, press the **Set** button in the **Body** tab after selecting the *WinfoNotify*. You can select the type of Winfo document to be created each time a *WinfoNotify* is sent. The types are:

- **Full**—a full Winfo document
- **Partial**—a partial Winfo document
- **None**—no Winfo document
- **Toolkit**—the default behavior of the Winfo module

Terminate

Terminates the selected object (Subscription or Notify).

Set Connection

Sets a connection to a persistent *PresSubscribe*, *WinfoSubs* or *EventsSubs*. The connection that is set will be either the connection of a marked object (to mark an object, right-click on the object and select **Mark**), or the selected connection (a connection that was selected with the **Select** command).

High Availability Frame

Store

Stores the selected Events object in the file specified in the text box to the right of the **Browse** button.

Restore

Restores an Events object from a file specified in the text box to the right to the **Browse** button.

Expiration Value

Sets the expiration value of the restored Events object to the value specified in the text box under the button.

Browse

Browses for a Store or Restore file and displays the selected file in the text box to the right of the button.

Set CSeq

Sets the CSeq number of restored Events objects to the value specified in the text box to the right of the button.

STACK DIALOG REMOTE CONTROL

You can perform the non-automatic actions on a *CallLeg* by choosing the options described under the **StackDlg Remote Control** window.

Figure 17-23 *StackDlg Remote Control Window*

Actions Frame

Respond

Pressing this button sends a response message using the response code written in the field to the right of the button.

Connect

Connects the selected *CallLeg* call using the *CallLeg* identifiers from the **Setting Identifier** tab.

Disconnect

Disconnects the connected *CallLeg* call.

Modify

Sends a modify request (re-INVITE) on the selected *CallLeg*.

Cancel

Cancels the sent request.

General

Initiates a general request using the method written in the field to the right of the button.

Terminate

Terminates the *CallLeg*.

Create

Creates a *CallLeg*.

Connection Remote Control

You can perform the non-automatic actions on a *Connection* object by choosing the options described under the **Connection Remote Control** window.

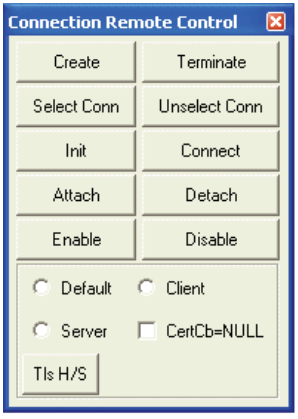


Figure 17-24 *Connection Remote Control Window*

Create

Creates a new *Connection*.

Terminate

Terminates a *Connection*.

Select Conn

Selects a *Connection* that other objects use for persistent connection (when using the **Set Conn** option).

Unselect Conn

De-selects a *Connection*.

Init

Initializes a *Connection*. The parameters for initialization are taken from the **Connection** tab in the main window. To access the **Connection** tab, select the **Setting Identifier** tab in the main window and select the **Conn** option at the bottom-right of the window.

Connect

Connects the selected *Connection*.

Attach

Attaches the application object as the owner of the *Connection*.

Detach

Detaches an application object from a *Connection*.

Enable

Enables a *Connection*.

Disable

Disables a *Connection*.

TLS Connection

Default/Client/Server

Determines the TLS connection handshake side.

CertCb=NULL

Assigns NULL to the TLS Certificate callback.

Tls H/S

Performs a TLS handshake for the TLS connection.

PUBLISH REMOTE CONTROL

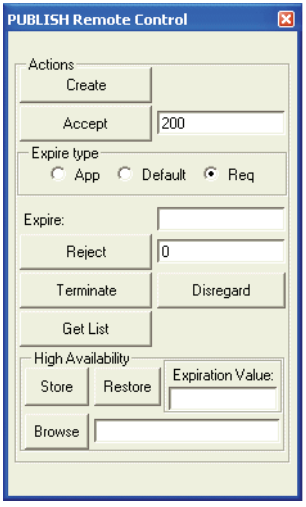


Figure 17-25 *Publish Remote Control Window*

Create

Creates a new PUBLISH request for restoring a Publish object from a file.

Accept

Accepts the PUBLISH request handled by the selected Publish object.

Expire Type

Indicates from where the expiration value should be taken. **App** means it will be taken from the application as written in the **Expire** text box. Default means the default value from the configuration will be taken. **Req** means that the requested expiration value from the PUBLISH request will be taken.

Reject

Rejects the PUBLISH request handled by the selected Publish object, with the response code written in the text box next to the button.

Terminate

Terminates the PUBLISH request.

Disregard

Disregards a failure found while validating or authenticating a request handled by the selected Publish object.

Get List

Gets the list of all Publish objects that have the same resource, package and template as the selected Publish object. The list will be printed to the **Lists** section of the general tabs.

High Availability Frame

Store

Stores the selected Publish object in the file specified in the text box to the right of the **Browse** button.

Restore

Restores a Publish object from a file specified in the text box to the right to the **Browse** button.

Expiration Value

If an expiration value is given, the restored Publish object will expire accordingly.

Browse

Browses for a Store or Restore file and displays the selected file in the text box to the right of the button.

LDAP Remote Control

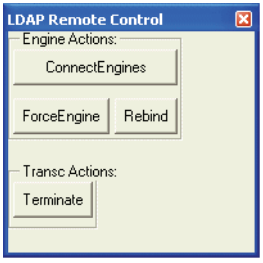


Figure 17-26 *LDAP Remote Control Window*

Engine Actions Frame

Connect Engines

Connects the LDAP add-on engines to their related LDAP server as defined in the configuration.

Force Engine

Disconnects the current LDAP engine and connects the specified one.

Rebind

Reconnects the selected LDAP engine that was disconnected and returns it to the round-robin cycle.

Transc Actions Frame

Terminate

Terminates the selected *LDAPTransc*.

Presence Agent Remote Control

You can perform actions on a *PAPresentity* object by choosing the options described under **PA Remote Control** window.

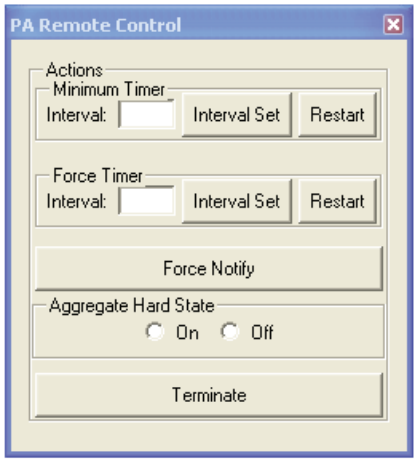


Figure 17-27 *PA Remote Control Window*

Minimum Timer Frame

Interval

Determines the new interval of the Minimum Timer. After setting this value you need to press the **Interval Set** button.

Interval Set

Sets the Minimum Timer interval that appears in the **Interval** text box.

Restart

Restarts the Minimum Timer. This is very useful, for example, after changing the interval so that the Minimum Timer will restart with the new interval that you set.

Force Timer Frame

Interval

Determines the new interval of the Force Timer. After setting this value you need to press the **Interval Set** button.

Interval Set

Sets the Force Timer interval that appears in the **Interval** text box.

Restart

Restarts the Force Timer. Press this button if you want the Force Timer to restart. This is very useful, for example, after changing the interval so that the Force Timer will restart with the new interval that you set.

Force Notify

Sends an updated NOTIFY request according to the last known state of the Presentity. If you mark a specific *PresSubscribe* (by right-clicking it and choosing **Mark**), a NOTIFY request will only be sent to the specific Watcher. If you did not mark a specific *PresSubscribe*, a NOTIFY request will be sent to all the Watchers.

Aggregate Hard State

Determines whether or not the HardState will be aggregated to the NOTIFY requests that are sent to the Watchers. This flag is meaningful if the HardState was changed from its default value, because the default HardState is minimal and will appear anyway in the aggregated document.

Terminate

Terminates the *PAPresentity*.

B2BAF Service Remote Control

You can perform actions on B2BAF *Service* objects by choosing the options described under B2BAF **Service Remote Control** window.

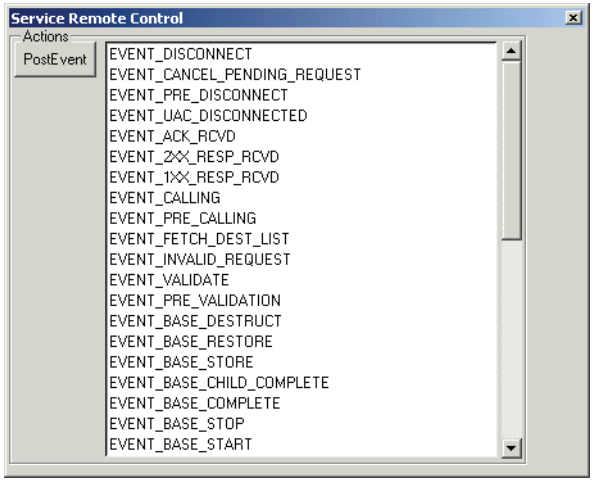


Figure 17-28 *B2BAF Service Remote Control Window*

PostEvent

Posts the selected event to the selected *Service*.

Components Menu

You access the following server components of the SIP Server through this menu. For more information, see Server Components on page 467.

Security

Accesses the Security Server Component. For more information, see the Security Component on page 467.

Location DB

Accesses the Location DB Server Component. For more information, see Location Database Server Component on page 468.

Help Menu

You can access the following form the **Help** menu.

ABOUT

Displays the **About** window. In this window you can find the SIP Server version, the SIP Stack version on which the SIP Server is based, and the available SIP Server add-ons.

OBJECTS WINDOW

You can view and select active objects in the **Objects** window. The different types of objects are listed according to the SIP Server layers and modules. Each object is displayed in a separate color, with its data and type description. Selecting any of the objects updates the data in the entire **Tab** area to reflect the data members held in that object, thus enabling browsing though the data members of the object. Some objects—such as a Proxy transaction, *B2B*, *B2BDlg*, *B2BTransc*, *CallLeg* transaction, *PresNotification*, and *PAPublishElem*—are displayed as a "child" of their owner object. For example if a Proxy transaction is created by a stateful *ProxyCoreObj*, the Proxy transaction will be displayed as a "child" of the *ProxyCoreObj*. Table 17-1 displays the objects and information about them.

Note You can mark each object by selecting the object and then right-clicking it. Marking an object enables setting its connection to other objects.

.

Table 17-1 *Active Object Window*

Object Type	Color	Data Displayed in Line of Object	Type Description
ProxyCoreObj—stateless	Navy Blue	From Header	SL
ProxyCoreObj—transaction stateful	Blue	From Header	SF
RegServerObj	Purple	To Header	REG
ProxyTransc	Steel Blue2	URI	TRN
ProxyTrx	Steel Blue 2	URI	PROXYTRX
B2B	Green3	From Header	B2B
B2BDlg	Pale Green3	From Header	B2BDlg

Table 17-1 *Active Object Window*

Object Type	Color	Data Displayed in Line of Object	Type Description
B2BTransc	Light Sea Green	method	B2BTransc
PresSubscribe	Magenta	Request URI	PSubs
PresNotification	Pink3	N/A	PNotify
WinfoSubs	Dark Orange	Request URI	WSubs
WinfoNotify	Light Salmon	N/A	WNotify
EventsSubs	Sienna3	Request URI	ESubs
EventsNotify	Sienna1	N/A	ENotify
CallLeg	Dark Green	From Header	DLG
CallLeg Transaction	Medium Aqua Marine	Method	DLGTran
Connection	Black	Remote Address	CONN
PublishServer	Orange	Request URI	PUBLISH
PAPresentity	Blue	Presentity	PA
PAPublishElem	Navy Blue	N/A	PAPuElem
LDAPEngine—Security Service	Dark Orchid	State	LDAPE
B2BAF Service	Slate Blue	N/A	B2B Service

TAB AREA

The following tabs are included in the Tabs area:

- **General**
- **Lists**
- **Log**
- **E-DNS**
- **Setting Identifier**

- **Advance**
- **Body**
- **Config**

GENERAL TAB

Figure 17-29 below shows the **General** tab which displays the main data members of the selected object according to the object type.

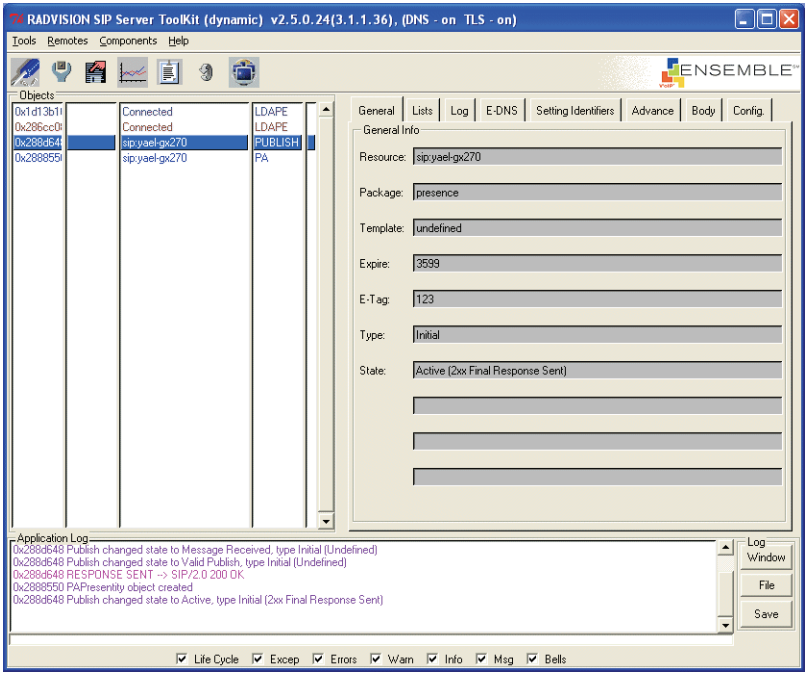


Figure 17-29 *General Tab*

The following data members are displayed for a *ProxyCoreObj*:

- Transaction Method
- Call-Id
- From Header
- To Header
- Object State and Changed State Reason
- Request URI

- Branch
- Received Response
- Received Local Address

The following information is displayed for a *ProxyTransc*:

- Transaction Method
- Call-Id
- From Header
- To Header
- Object State and changed state reason
- Request URI
- Branch
- Received Response
- Connection used by object if persistent
- Outbound Address

The following information is displayed for a *ProxyTrx*:

- Method
- Call-Id
- From Header
- To Header
- Object State and changed state reason
- Request URI
- Branch
- Received Local Address
- Received Response
- Outbound Address

The following data members are displayed for a *RegServerObj*:

- From Header
- To Header
- Expire Header
- Object State and Changed State Reason.

The following information is displayed for a *B2B*:

- UAS From Header
- UAS To Header

- UAC From Header
- UAC To Header
- Object Connect Call State and Changed State Reason
- Object Modify Call State and Changed State Reason
- Object Disconnect Call State and Changed State Reason
- Object Transfer State and Changed State Reason

The following information is displayed for a *B2BTransc*:

- Method
- Object State and Changed State Reason.
- Received Local Address

The following information is displayed for a *B2BDlg*:

- From Header
- To Header
- Object Connect State and Changed State Reason
- Object Modify State and Changed State Reason
- Object Disconnect State and Changed State Reason
- Dialog Direction (incoming or outgoing)
- TRN (Transfer) Role
- Connection used by object if persistent
- Outbound Address

The following information is displayed for a *PresSubscribe*/*WinfoSubs*/*EventsSubs*:

- From Header
- To Header
- Request-URI
- Expire
- Object State
- Connection
- Remaining Time

The following information is displayed for a *PresNotification*/*WinfoNotify*/*EventsNotify*:

- Notify Type
- Object State

- Object Type

The following information is displayed for a *CallLeg*:

- From Header
- To Header
- Call-ID
- Local Contact Address
- Remote Contact Address
- Object State and Changed State Reason
- Object Modify Call State and Changed State Reason
- Object Disconnect Call State and Changed State Reason

The following information is displayed for *CallLeg Transaction*:

- Method
- Object State and Changed State Reason.

The following information is displayed for *Connection*:

- Transport
- Local IP and Port
- Remote IP and Port
- Number of *Connection* owners
- Host name
- *Connection* Object
- *Connection* State and Reason
- TLS Connection State and Reason

The following information is displayed for a *PAPresentity*:

- Presentity
- Object State
- Watchers Count
- Minimum Timer Remaining Time
- Force Timer Remaining Time
- Minimum Timer interval
- Force Timer Interval
- Aggregated HardState flag

The following information is displayed for a *PublishServer*:

- Request URI

- Package
- Template
- Expire
- E-Tag
- Type
- Object State and changed state reason

The following information is displayed for *LDAPEngine*:

- Host
- State
- Type
- Service

The following information is displayed for B2BAF *Service*:

- Service
- State
- LastEvent

LISTS TAB

Figure 17-30 shows the **Lists** tab which displays the list information of the selected object.

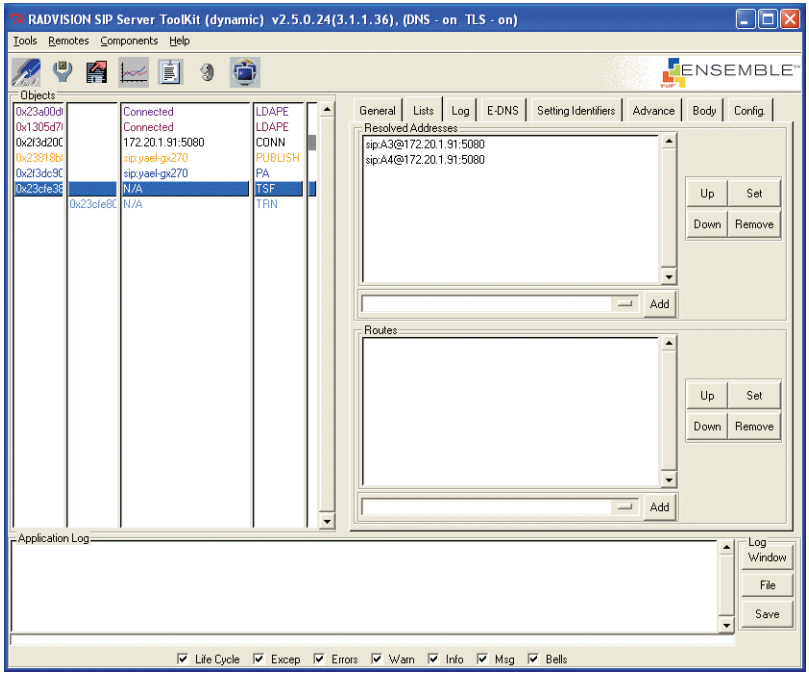


Figure 17-30 *Lists Tab*

The **Lists** tab is relevant for the objects type listed below:

- *ProxyCoreObj*
- *B2B*
- *WinfoSubs*

The following data members are displayed for a *ProxyCoreObj*:

- **Resolved Addresses List**—you can add, remove or change the order of addresses in the Resolved Addresses List. If you want the changed list to replace the original Resolved Addresses List of the *ProxyCoreObj*, press **Set**.
- **Route List**—to add routes to a *ProxyCoreObj*, add the routes to the list of routes and then press **Set**. The list of routs per *ProxyCoreObj* is not displayed upon selection; you can set a new list but you cannot edit a previously set list.

The following data members are displayed for a *B2B*:

- **Response Message List**—this list is only for display and must not be manipulated.
- **Contact Address List**—to add contact addresses to a *B2B*, add the contact address to the list, select a *B2BDlg* on the **B2BDlg List** and then press **Set**. the list of contact address per *B2BDlg* is not displayed upon selection; you can set a new list but you cannot edit a previously set list.

The following data members are displayed for a *WinfoSubs*:

- Each time the *WinfoSubs* is selected, the application queries the Events module for existing Watchers relevant to this subscription. The list of Watchers that were found is displayed. This list will contain the content of a full Winfo document created for this *WinfoSubs*.

LOG TAB

Figure 17-31 shows the **Log** tab which displays incoming and outgoing messages for an object.

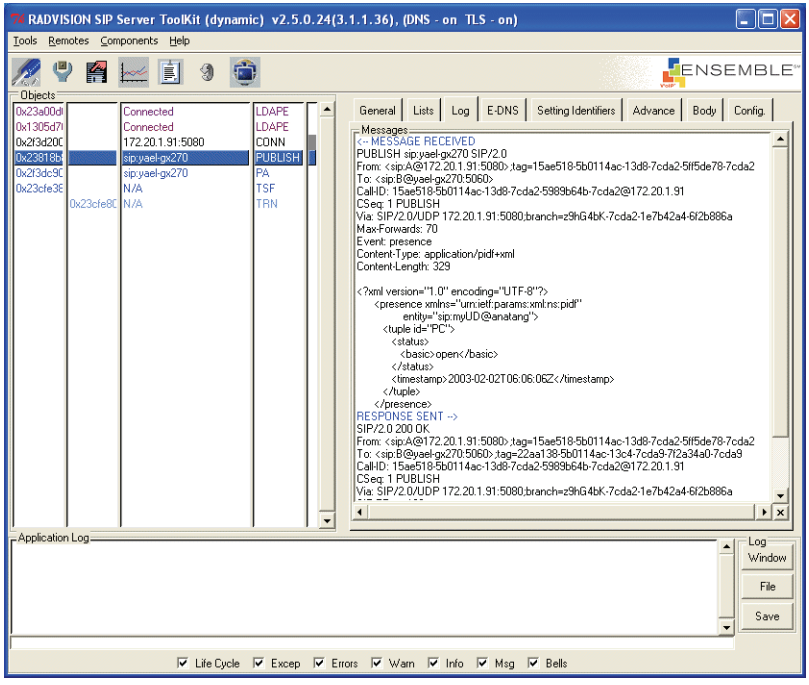


Figure 17-31 *Log Tab*

E-DNS TAB

Figure 17-32 shows the E-DNS tab which displays DNS-related data for a client *ProxyTransc*. If the SIP Server is compiled with the RV_DNS_ENHANCED_FEATURES_SUPPORT flag, this tab will display the data received from the DNS to resolve the host name. This tab is relevant when working with a Transaction Stateful Proxy mode with the DNS feature.

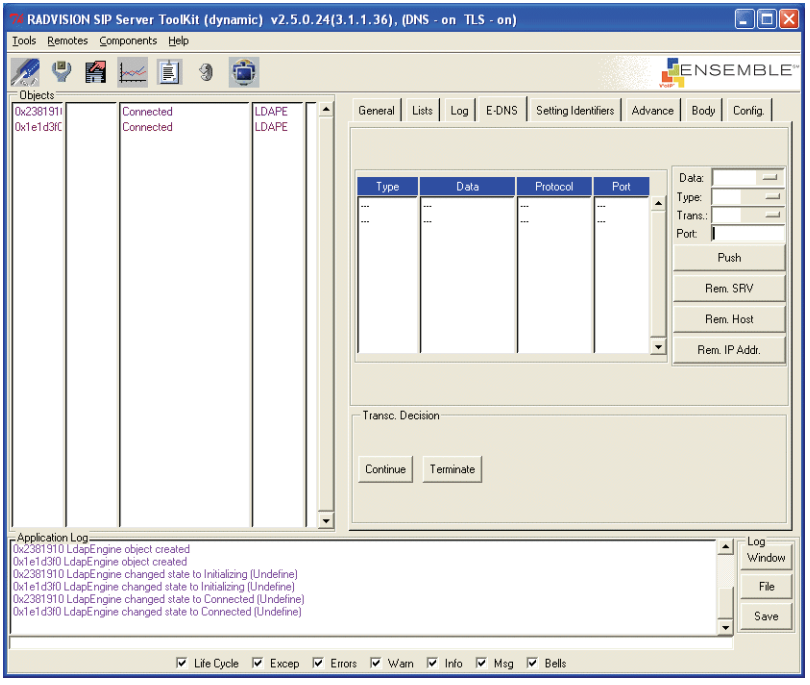


Figure 17-32 *E-DNS Tab*

WHAT YOU CAN DO IN THE E-DNS TAB

- You can add or delete DNS records from the DNS list once a RvProxyCoreTranscMsgSendFailureEv() of a transaction is called or the *ProxyTrx* reaches the MSG_SEND_FAILURE state.
- You can choose to continue with the next address or to terminate the transaction or *ProxyTrx*.
- To insert a new DNS entry into the transaction or *ProxyTrx*, complete the following fields and press **Push**.

Data

The IP address or the host name of the record.

Type

The type of the DNS record to insert—a host record, an SRV record or an IP address record.

Transport

TCP, UDP or other types of transport.

Port

The port number for the DNS record.

SETTING IDENTIFIER TAB

Figure 17-33 shows the **Setting Identifier** tab which allows you to set the object identifiers. The setting will take effect only for objects that are created next and not for existing objects. The fields available for setting are different for the different types of objects.

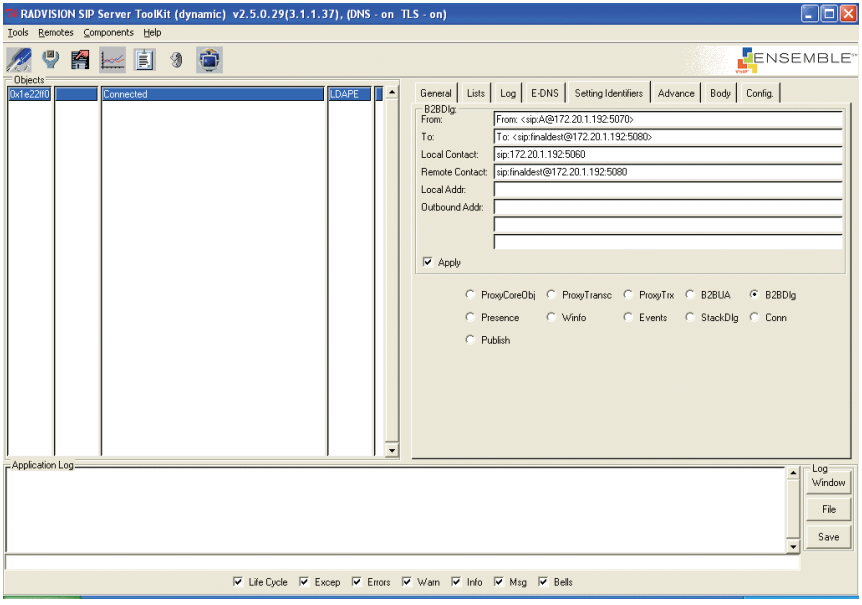


Figure 17-33 *Setting Identifier Tab*

The **Setting Identifier** tab is related to the following objects:

- *ProxyCoreObj*
- *ProxyTransc*

- *ProxyTrx*
- *B2B*
- *B2BDlg*
- *PresNotification*
- *PresSubscribe*
- *WinfoDoc*
- *WinfoNotify*
- *WinfoSubs*
- *EventsNotify*
- *EventsSubs*
- *StackDlg*
- *Connection*
- *PublishServer*

The following identifiers can be set to a *ProxyCoreObj*:

- **Via**—overrides the *ProxyCoreObj* Via header with a specific Via header.
- **Req R.Route**—overrides the *ProxyCoreObj* default Record-Route header with a specific Record-Route header. Otherwise, *ProxyCoreObj* objects will have their default attributes as configured in the SIP Server.
- **Resp R.Route**—rewrites the Record-Route header in the response message.
- **Local Addr**

The following identifiers can be set to *ProxyTransc*:

- From—sets a From header to the *ProxyTransc*. This field is mandatory when the application is initiating a message.
- To—sets a To header to the *ProxyTransc*. This field is mandatory when the application is initiating a message.
- Request-URI—sets a URI header to the *ProxyTransc*. This field is mandatory when the application is initiating a message.
- CSeq—sets a CSeq value to the *ProxyTransc*. This field is mandatory when the application is initiating a message.
- Method—sets the method of the message.
- Call-Id—sets a Call-Id value to the *ProxyTransc*. This field is mandatory when the application is initiating a message.

- Outbound Addr—sets the Outbound Address and port to which the message will be sent.

The following identifiers can be set to a *ProxyTrx*:

- Local Addr—sets the local address from which the message will be sent.
- Dest Addr—sets the destination address to which the message will be sent.
- Via Branch—sets the Branch used in the Via header of the *ProxyTrx*.
- Outbound Addr—sets the Outbound Address and port to which the message will be sent.

The following identifiers can be set to a *B2B***:**

- **From**—sets a From header to the outgoing/First party *B2BDlg*. This field is mandatory when the application is Transparent at the call and is working in non-automatic mode, or when the application behaves as a 3PCC and is working in automatic mode.
- **To**—sets a To header to the outgoing/First party *B2BDlg*. This field is mandatory when the application is Transparent at the call and is working in non-automatic mode, or when the application behaves as a 3PCC and is working in automatic mode.
- **Sec Party From**—sets a From header to the second party *B2BDlg*. This field is mandatory when the application behaves as a 3PCC and is working in automatic mode.
- **Sec Party To**—sets a To header to the second party *B2BDlg*. This field is mandatory when the application behaves as a 3PCC and is working in automatic mode.
- **Trt From**—sets a From header to the Trt *B2BDlg*. This field is mandatory when the application behaves as a 3PCC and is working in automatic mode.
- **Trt To**—sets a To header to the Trt *B2BDlg*. This field is mandatory when the application behaves as a 3PCC and is working in automatic mode.
- **Local Contact Addr**—adds a local contact to the outgoing *B2BDlg*. This field is optional.

- **Outgoing Contact Address**—adds a remote contact address to the outgoing *B2BDlg*. This field is optional.

The following identifiers can be set to **B2BDlg**:

- **From**—sets a From header to the *B2BDlg*. This field is mandatory when the application behaves as a 3PCC and is working in non-automatic mode.
- **To**—sets a To header to the *B2BDlg*. This field is mandatory when the application behaves as a 3PCC and is working in non-automatic mode.
- **Local Contact**—adds a Local Contact to the *B2BDlg*. This field is optional.
- **Remote Contact**—adds a Remote Contact address to the *B2BDlg*. This field is optional.
- **Local Addr**—specifies the local interface that the *B2BDlg* will use to send the request. This field is optional.
- **Outbound Addr**—sets the Outbound Address and port to which the message will be sent.

The following identifiers can be set to **Presence**:

- **Watcher Addr**—sets the address of the Watcher that will be used for composing Winfo documents.
- **Expire**—sets the expiration value of the Presence SUBSCRIBE request.
- **Local Addr—**specifies the local interface that the *PresNotification* will use to send the NOTIFY request. This field is optional.

The following identifiers can be set to **Winfo**:

- **Watcher Addr**—sets the address of the Watcher that will be used for composing Winfo documents.
- **Expire**—sets the expiration value of the Winfo SUBSCRIBE request.
- **Local Addr—**specifies the local interface that the *WinfoNotify* will use to send the NOTIFY request. This field is optional.

The following identifiers can be set to **Events**:

- **Watcher Addr**—sets the address of the Watcher that will be used for composing Winfo documents.
- **Expire**—sets the expiration value of the SUBSCRIBE request.

- **Local Addr—**specifies the local interface that the *EventsNotify* will use to send the NOTIFY request. This field is optional.

The following identifiers can be set to **StackDlg**:

- From—sets a From header to the *CallLeg*. This field is mandatory when the application is working in non-automatic mode.
- To—sets a To header to the *CallLeg*. This field is mandatory when the application is working in non-automatic mode.
- Local Contact Address—adds a Local Contact to the *CallLeg*. This field is optional.
- Outgoing Contact Address—adds a Remote Contact address to the *CallLeg*. This field is optional.

The following identifiers can be set to **Conn**:

- **Transport**—sets the transport of the connection (TCP/TLS)
- **Local IP**—the local IP "from" configuration
- **Local Port**—the local port
- **Remote IP**—the remote IP for this connection
- **Remote Port**—the remote port for this connection
- **Host name**—for TLS

The following identifiers can be set to *PublishServer*:

- E-Tag—sets the E-Tag for each response for a Publish request. If not set, the Publish module will use the existing E-Tag or generate a new one if an E-Tag was not yet generated.
- Min-Expire—sets the minimum expiration time for a Publish request.

ADVANCE TAB

Figure 17-34 shows the **Advance Tab** which displays and edits the Session-Timer. The values will be set for a *B2BDlg* and *ProxyCoreObj* upon creation of an object.

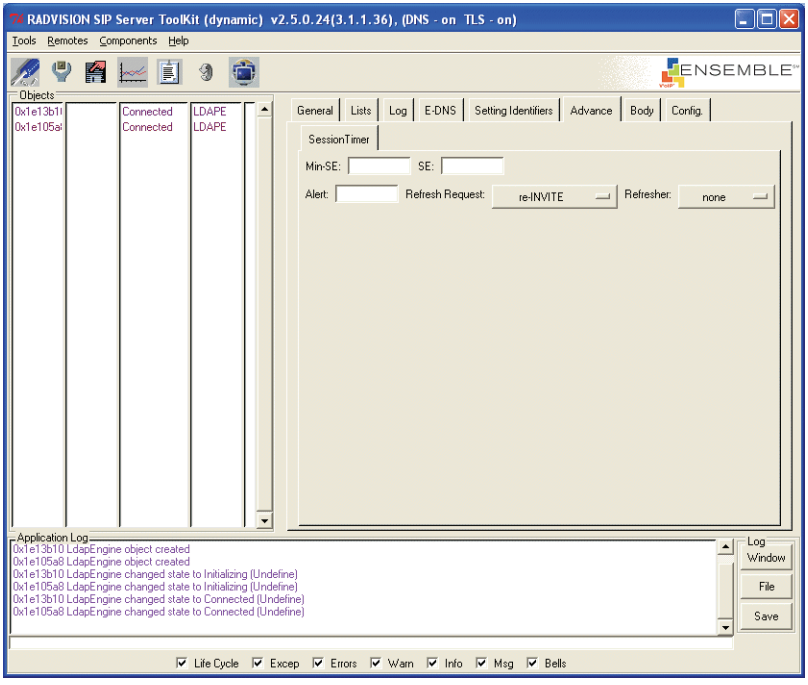


Figure 17-34 *Advance Tab*

Min-SE

Overwrites the Min-SE value of the object from configuration.

SE

Overwrites the SE value of the object from configuration.

Alert

Defines the amount of time (in msec) that the application is notified before the session is expires. This value is relevant only for a *B2BDlg*.

Refresh Request

Defines the refresh request type to be used when the Session expires. This value is relevant only for a *B2BDlg*.

Refresher

Overwrites the refresher value from configuration. This value is relevant only for a *B2BDlg*.

BODY TAB

Figure 17-35 shows the **Body** tab which displays and edits the body that will be added to an outgoing request. The body text will be set to the outgoing message of the selected object. The objects of the B2BUA and Events add-on modules use this tab. The Presence Agent uses this tab to set/display SoftState, HardState or Aggregated documents.

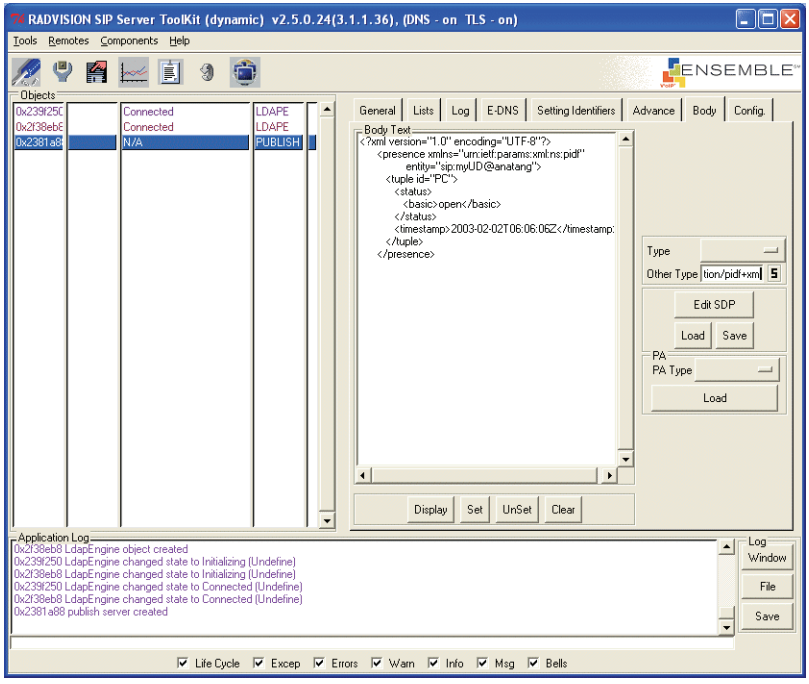


Figure 17-35 *Body Tab*

Body Text

Area for typing, editing and displaying body text.

Type

For selecting the Content-Type header of the body from a list of suggested Content-Type headers. If the header does not appear in the list, type the header in the **Other Type** text box and press the **S** button to set it.

Edit SDP

Opens the SDP Editing Screen, displayed in Figure 17-36.

PAType

For selecting the type of document from a list of suggested types. The types are:

- HardState—relevant only for a *PAPresentity*
- AggDoc—relevant only for a *PAPresentity*
- SoftState—relevant only for a *PAPublishElem*

These types can be set or displayed via the **Set** and **Display** buttons.

Load

Loads a text file in the **Body Text** windows—for Presence Agent objects only.

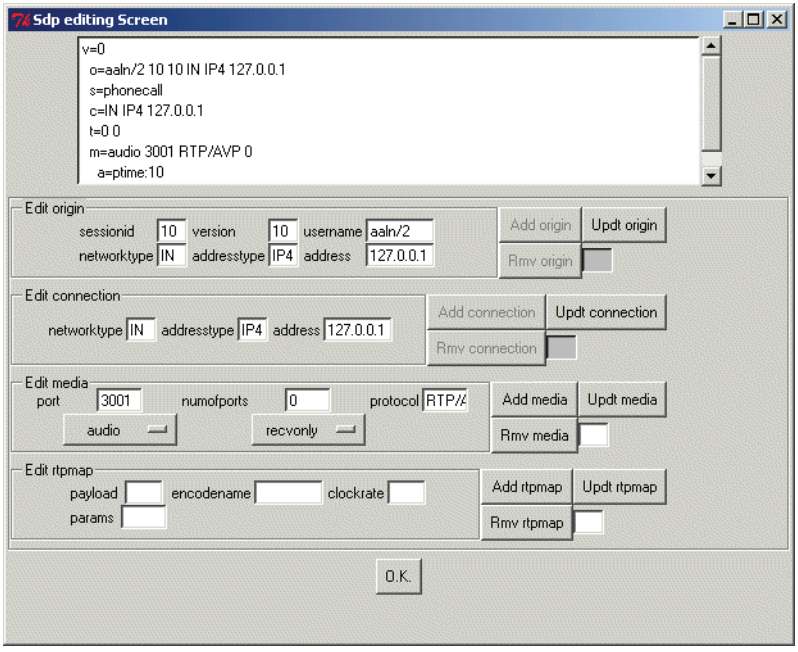


Figure 17-36 *SDP Editing Screen*

The **SDP Editing Screen** lets you edit the body with the action buttons related to the SDP parameter. This way, the application assures that the edited SDP syntax is valid.

There are four kinds of SDP parameters that can be manipulated (added, removed or updated) by the screen—**origin**, **connection**, **media** and **rtpmap**.

Load

Loads a text file in the **Body Text** window. For PA objects only.

PA Type

For selecting the type of document from a list of suggested types. The types are:

- HardState—relevant only for a *PAPresentity*
- AggDoc—relevant only for a *PAPresentity*
- SoftState—relevant only for a *PAPublishElem*

These types can be set or displayed via the **Set** and **Display** buttons.

Save

Saves the body text to a file.

Display

Displays the body text of a selected object in the **Body Text** window.

Set

Sets the body text to the selected object.

UnSet

Sets the body of the selected object to a NULL string.

Clear

Clears the **Body Text** window.

CONFIGURATION TAB

Figure 17-37 shows the **Config** tab in which you can override the default configuration for an object. Settings selected here will override settings from the configuration file.

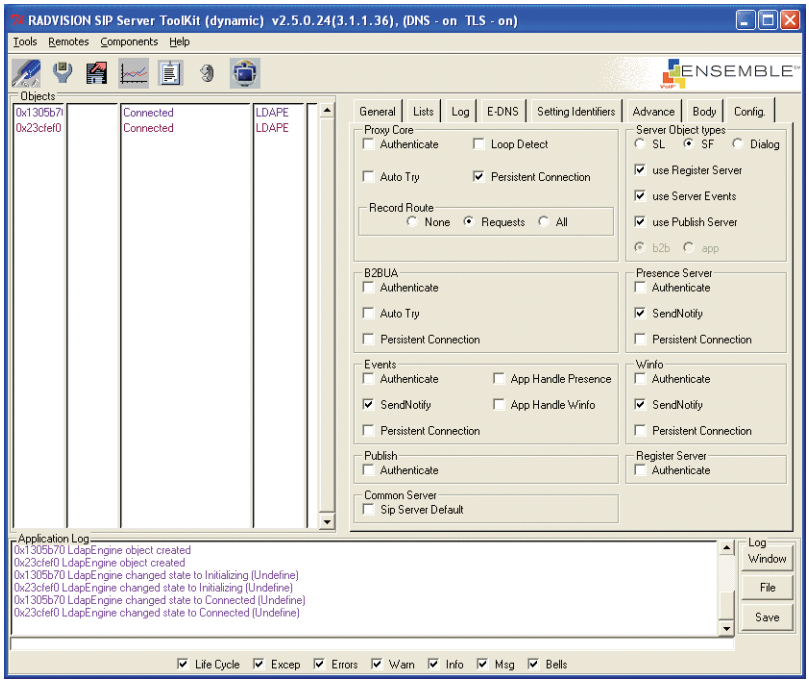


Figure 17-37 *Config Tab*

The parameters that you select in this tab will take effect for the next objects created, until another change is performed.

WHAT YOU CAN DO IN THE CONFIG TAB

You can re-configure an object by choosing the options described below.

Proxy Core Frame

Authenticate

When checked, incoming requests are authenticated.

Auto Try

When checked, a core object sends a 100 response on incoming requests.

Loop Detect

When checked, the SIP Server activates its loop detection mechanism.

Persistent Connection

When checked, the next object that is created will be with a persistent connection.

Record Route

Specifies the Record-Route level that this *ProxyCoreObj* should use:

- **None**—does not add a Record-Route header to requests
- **Request**—adds a Record Route header only to requests
- **All**—adds a Record Route header to requests and allows rewriting the Record-Route in responses.

This parameter is set from configuration and cannot be modified.

B2BUA Frame

Authenticate

When checked, incoming requests of *B2B* objects are authenticated.

Auto Try

When checked, a *B2B* sends a 100 response on incoming INVITE requests.

Persistent Connection

When checked, the next object that is created will be with a persistent connection.

Events Frame

Authenticate

When checked, incoming SUBSCRIBE requests are authenticated.

Send Notify

When checked, Presence Notification requests will be sent by the Presence Server.

Persistent Connection

When checked, the next object that is created will be with a persistent connection.

App Handle Presence

When checked, the application will handle *PresSubscribe* requests directly above the Events API.

App Handle Winfo

When checked, the application will handle *PresSubscribe* requests directly above the Winfo API.

Publish Frame

Authenticate

When checked, incoming PUBLISH requests are authenticated.

Common Server Frame

Sip Server Default

If checked, settings in this frame will have no effect and the newly created objects will have their default attributes as configured in the SIP Server configuration file.

Server Object Types Frame

The parameters in this group let you create different types of new objects for incoming requests:

SL

Creates a new stateless *ProxyCoreObj*. This parameter is available only if the SIP Server is configured to dynamic mode.

SF

Creates a new transaction stateful *ProxyCoreObj* (when possible). This parameter is available only if the SIP Server is configured to dynamic mode.

Dialog

Creates a new *Server Dialog* for initial INVITE requests as *B2B* or *CallLeg* (depending on the *Server Dialog* owner type described bellow). This parameter is available only if the SIP Server is configured to dynamic mode.

Use Register Server

When checked, a REGISTER request for a local domain will be handled by the Register Server module, using a *RegServerObj*.

Use Server Events

When checked, a SUBSCRIBE request for a local domain with event presence will be handled by the Events Server module.

Use Publish Server

When checked, a PUBLISH request for a local domain will be handled by the Publish Server module.

B2B

Creates new *B2B* for Initial INVITE requests. This parameter is available only if the SIP Server is configured to dynamic or dialog mode.

APP

Creates new *CallLeg* for Initial INVITE requests. This parameter is available only if the SIP Server is configured to dynamic or to dialog mode.

Presence Server Frame

Authenticate

When checked, incoming SUBSCRIBE requests are authenticated.

Send Notify

When checked, Presence Notification requests will be sent by the Presence Server.

Persistent Connection

When checked, the next object that is created will be with a persistent connection.

Winfo Frame

Authenticate

When checked, incoming SUBSCRIBE requests are authenticated.

Send Notify

When checked, Presence Notification requests will be sent by the Presence Server.

Persistent Connection

When checked, the next object that is created will be with a persistent connection.

Registrar Server Frame

Authenticate

When checked, incoming REGISTER requests are authenticated.

Application Log Area

The **Application Log** area is used for recording and displaying events. (This log is a tool to help monitor the SIP Server Platform on the application level, and is not part of the SIP Server Platform.) There are five different logging options, each useful for a different purpose. You can turn each option on or off by selecting or deselecting its radio button. The logging options are listed in Table 17-2

Table 17-2 *Logging Options*

Name	Function	Color
Life Cycle	Notifies of creation, termination and state changes of objects.	Purple
Excep	Notifies of SIP Server exception	
Error	Notifies of SIP Server errors.	Red

Table 17-2 Logging Options

Name	Function	Color
Warn	Notifies of SIP Server warnings.	
Info	Delivers important information from the application developer to the user.	Black
Msg	Displays the start line of each incoming and outgoing massage.	Magenta
Bells	Sounds a bell on incoming registrations and message send failure events.	(–)

What You Can Do in the Log Area

- You can clear the log window by selecting the **Window** button in the **Log Clear** frame.
- You can clear the log file by selecting the **File** button in the **Log Clear** frame.

SERVER COMPONENTS

You access the GUI interfaces to the supplied implementations for the Location DB and Security Server components by selecting **Components** on the menu bar.

SECURITY COMPONENT

Selecting **Security** from the **Components** menu opens the **Security Components** window.

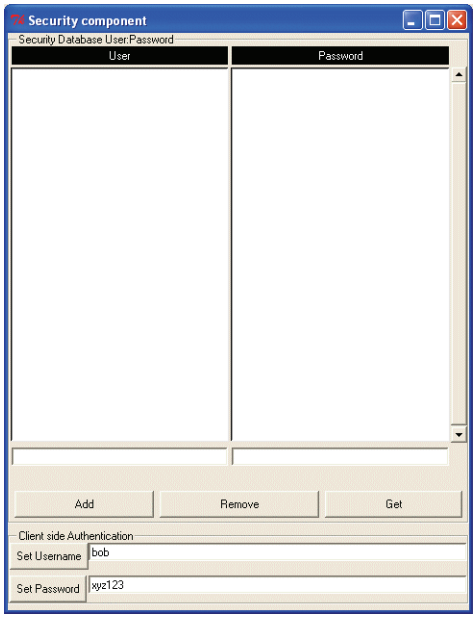


Figure 17-38 *Security Components Window*

You can view the user names and their matching passwords (as they were inserted into the Security database) in the respective columns. The security database is saved each time the application is shut down and loaded each time the application is started. The *securityDB.txt* file contains the security database.

WHAT YOU CAN DO IN THE SECURITY COMPONENT WINDOW

You can perform the following actions in the **Security Components** window:

Add

Inserts a name and a password in the text boxes under the user and password columns.

Remove

Removes a selected user.

Get

Gets the password for a user (from the database, rather than the GUI). Enter the user name in the text box under the **User** column and press **Get**. If the user is in the database, the password is displayed. Otherwise, “NOT FOUND” is displayed.

Client Side Authentication

Sets the SIP Server username and password for client-side authentication. The username and password will be used for authenticating a request which requires authentication.

Location Database Server Component

Selecting **Location DB** from the **Components** menu opens the **Location DB Components** window.

In this window you can view users that are registered to the SIP Server with their binding information.

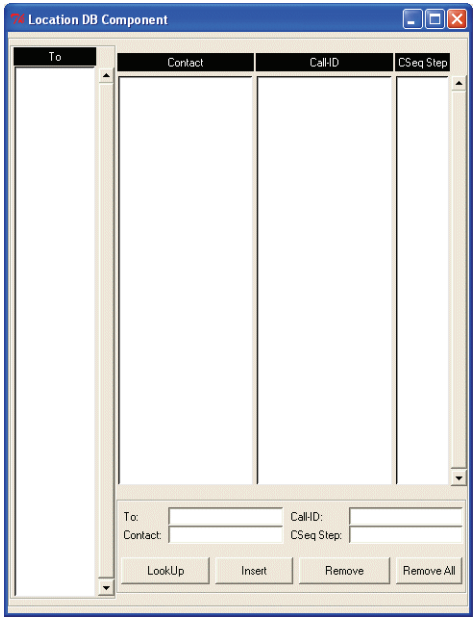


Figure 17-39 *Location DB Component Window*

The Location DB can be saved when the application is shut down and loaded when the application is started again. To enable this function check the **Save DB** option in the **Auto Registration** frame in the **App. Defaults** tab.

What You Can Do in the Location DB Component Window

The following buttons let you to interact with the database manually:

Lookup

Looks up in the database the user displayed in the **To** box.

Insert

Inserts the user in the **To** box into the database, using the values in the **call-ID**, **cSeq** and **contact** boxes.

Remove

Removes from the database the user displayed in the **To** box.

Remove All

Removes all records from the database.

18

SIP Server Sample Applications

Sample Applications

The SIP Server Platform includes the following sample applications.

- **SimpleGeneralTransaction**—A basic implementation of how to send an out-of-dialog general request using the SIP Server Platform. The SimpleGeneralTransaction initiates and sends general request such as "OPTIONS" to an address of your choice. (You need to configure the sample before running it.) This sample also demonstrates how to handle an out-of-dialog incoming general request. If the request is valid, the SimpleGeneralTransaction will respond with a 180 provisional response and then accept the incoming request.

  You can find the sample code for the SimpleGeneralTransaction under *samples\sipServer\SimpleGeneralTransaction.*

- **SimpleRedirectServer**—A basic implementation of a redirect server that was built using the SIP Server Platform. The SimpleRedirectServer redirects incoming requests to a set of predefined destinations.

  You can find the sample code for the SimpleRedirectServer under *samples\sipServer\SimpleRedirectServer.*

- **SimpleRegistrar**—A basic implementation of a Registrar server that was built using the SIP Server Platform. The SimpleRegistrar enables working with authentication. In order to enable the authentication, select “use authentication” when asked.

  If you want the requests to be authenticated successfully by the SimpleRegistrar, use the following parameters in the registering user agent (UA):

  - User: “User”
  - Password: “Password”
  - Realm: “radvision.com”
  - The algorithm used for authentication is MD5

  You can find the sample code for the SimpleRegistrar under *samples\sipServer\SimpleRegistrar*.
- **SimpleStatelessProxy**—A basic implementation of a stateless Proxy that was built using the SIP Server Platform. The SimpleStatelessProxy enables forwarding incoming requests in stateless mode.

  You can find the sample code for the SimpleStatelessProxy under *samples\sipServer\SimpleStatelessProxy*.
- **SimpleStatefulProxy**—A basic implementation of a stateful Proxy that was built using the SIP Server Platform. The Proxy enables registering to a database and then forwarding incoming requests in a parallel or sequential mode. You can choose between parallel and sequential mode in the Start menu.

  You can find the sample code for the SimpleStatefulProxy under *samples\sipServer\SimpleStatefulProxy*.
- **SimpleTlsServer**—A basic implementation of a stateful Proxy that supports TLS, which was built using the SIP Server Platform. The proxy enables registering to a database and then forwarding incoming requests. This sample also demonstrates how to:

  - Initiate TLS engines, which are responsible for TLS connection management.
  - Initiate a TLS handshake on a connected connection.

- Use the “sips:” scheme to conduct a secure SIP session.

Note: The supplied certificates are valid for local addresses 127.0.0.1.

You can find the sample code for the SimpleTlsServer under *samples\sipServer\SimpleTlsServer*.

- **AdvancedTlsServer**—A proxy that support TLS, which was built using the SIP Server Platform. The proxy enables registering to a database and then forwarding incoming requests. This sample also demonstrates how to:
  - Initiate TLS engines, which are responsible for TLS connection management.
  - Initiate a TLS handshake on a connected connection.
  - Use the “sips:” scheme to conduct a secure SIP session.
  - Use a user-made verification callback and inspect the certificate.
  - Override the SIP Stack decision on post connection assertions.

    Note that in this sample a deliberately mis-issued certificate is used.

    You can find the sample code for the AdvancedTlsServer under *samples\sipServer\AdvancedTlsServer*.

SAMPLE CODE

Each of the above sample codes is fully documented and each application contains a *README.txt* file that explains the basic concepts of the application and how to use the application.

INDEX

A

accounting 32
advanced features
  DNS 287
    related functions, events 288
    sending failures 287
  general URL scheme support 294
    handling general URL with "pres" 296
    handling received requests 295
    sending requests 294
  IPv6 289
  multihomed host 289
    receiving messages 290
    sending messages 290
  multihomed-host
    proxy 291
  session-timer 297
    API functions 298
    configuration parameters 297
    proxy behavior 297
API
  mid-layer management API 360
  mid-layer select 364
  mid-layer thread 367
  mid-layer timer 362

B

B2BAF 26
bad syntax
  API 160
  events (callbacks) 159
  fixing bad syntax messages 159
  handling bad syntax messages 156
  header 155
  parameter 155
body object 166
body part object 166

C

client connection state machine 193
compact form 153
configuration
  initialization 299
configuration file 301
configuration parameters 301
  B2B configuration parameters 323
  B2BAF configuration parameters 355
  B2BAF service library configuration parameters 358
  DNS configuration parameters 351
  events configuration parameters 326
  LDAP configuration parameters 350
  log configuration parameters 334
  multithreading configuration parameters 321
  network configuration parameters 312
  proxy configuration parameters 304
  register configuration parameters 310
  server dialog configuration parameters 322
  SIP Server configuration parameters 301
  stack configuration parameters 342
  TLS configuration parameters 317
  XML Manager configuration 354
content-type header 168

D

database access 32
DLA 220
  API 221

DNS 287
  related functions, events 288
  sending failures 287
dynamic local address (DLA) 220
  API 221

E

enhanced DNS 223
  API 227
  caching 236
    compilation flags 236
  call-leg layer 234
  changing DNS list before sending message 235
  configuring DNS parameters 223
  DNS list API 230
  DNS/ENUM record 225
  DNS/SRV tree 224
  manipulating DNS list object 230
  publish-client layer 235
  RADVISION implementation 225
  register-client layer 235
  state machine 227
  subscriptions layer 234
  support for call-legs, subscriptions, register-clients 234
enhanced DNS state machine 227
Events Server 27
  authentication 28
  event package name 28
  subscribe duration 28

G

general URL scheme support 294

H

headers
  content-type 168
  stand-alone headers, stand-alone headers 148

I

infrastructure components 32
  accounting 32
  database access 32
IPv6 289
  addresses 217

L

Location Database Server Component 239
  API 244
  configuration parameters 246
  interface 239
  replacing implementation 240
log 369
  changing configuration during runtime 376
  compilation log control 378
  configuring 374
  controlling 376
  file 373
  initializing 375
  message structure 374
  messages 373
  source identifiers 370

M

Memory Pool 281
  API 281
  constructing, destructing 284
  copying buffer to page 283
  copying page into buffer 282
  definitions 281
  using 282
mid-layer 359
  threading considerations 359
mid-layer management API 360
  constructing, destructing 360
  manager handle 360
mid-layer select API 364
  events 365
  functions 364
  select() loop 366
mid-layer thread API 367
mid-layer timer API 362
  control functions 363
  events 363
  handle 363
  threading considerations 362

utility functions 363
multihomed host 289, 290
multihomed-host proxy 291
multipart MIME 165
multithreaded mode 63

P

persistent connection
handling 177
pres scheme 296
Presence Server 29
Proxy Core 65
entities 66
ProxyCoreObj 66
ProxyCoreTransc 66
ProxyPolicyMgr 66
ProxyTrx 67
handles 68
sample code 112
transaction stateful mode 67
workflow diagrams 106
Proxy Core API
Proxy Core Object API 70
control 73
events 75
parameters 70
Proxy Core Policy Manager API 69
control 69
parameters 69
Proxy Core Transaction API 81
control 84
parameters 81
Proxy Core Transmitter API 85
control 88
events 89
parameters 86
Proxy Core Object state machine 90
Proxy Server 5
address resolution 11
authentication 22
determining Target-set 11
DNS resolution 12
forking 19
loop detection, max forwards 23
loose-routing 18
message flow diagrams 12
message spiraling 24
outbound proxy 24
policy 25
request validation 10
stateful message forwarding 12
stateful, stateless proxies 6
symmetric record-route 17
Proxy Transaction state machine 101
Proxy Transmitter state machine 102

R

RADVISION SIP Server Platform 34
architecture 46
capabilities 39
Advanced B2BUA 42
Events Server 43
general 40
Presence Server 43
proxy 41
redirect 42
registrar 42
SDP Stack 43
standards compliance 39
Winfo module 44
non-Sip infrastructure
non-SIP infrastructure 44
package 47
raw buffer, message monitoring 219
Redirect Server 3
Register Server 117
entities 117
handles 118
workflow diagrams 128
Register Server API 120
control 123
events 123
parameters 120
Register Server Manager 119
callback function structure 120
Register Server Object state machine 125
Registrar Server 3

S

sample applications 471
  sample codes 473
SDT layer 269
  STD allocator interface 276
    STD list 276
    STD List API 276
    STD list entities 276
  XML Library 269
Security Server Component 249
  API 263
  configuration parameters 266
  creating application security component 259
  implementation 263
  initializing 264
  interface 255
  server authentication object 249, 250
    API 250
    control 251
    state machine 252
server
  connections 195
Server Authentication state machine 252
server components 467
server connection reuse 200
  API functions, events 203
  authorization of 204
server connection state machine 198
session-timer 297
SIP messages 143
  adding new headers 147
  API 141
  body object 166
  body part object 166
  body string 174
  compact form 153
    forcing 154
    get 153
    set 153
  content-type header 168
  creating 151
  creating multipart body 172
  encoding multipart body 174
  handles 138
  headers 141
  headers in body part object 169
  memory management 154
  multipart body structure 166
  multipart MIME 165
  objects 137
    header 140
    message 139
    message manager 138
  parsing multipart body 170
  parsing, encoding of 144
  reading, modifying of 143
  removing from message 150
  stand-alone headers 148
    creating 149
    setting in message 149
  syntax errors 155
    bad syntax API 160
    bad syntax events (callbacks) 159
    bad syntax header 155
    bad syntax parameter 155
    bad syntax start line 156
    fixing bad syntax messages 159
    handling bad syntax messages 156
SIP objects
  body 166
  body part 166
SIP Server 1, 51
  API conventions 52
  contents 51
  creating an application 54
  definition 1
  logging 53
  memory management 53
  threading model 63
    multithreaded mode 63
  working mode 62
SIP Server API
  data types 52
  function parameters 52
  functions 53
  return status codes 53
SIP Server application 54
  configuration 57
  event processing 59
  getting module handles 60
  initialization 54
  registering callbacks 61

SIP Server configuration
structure 300
SIP Sever application 48
source identifiers 370
SRV 224
state machines
Proxy Core Object state machine 90
Proxy Transaction state machine 101
Proxy Transmitter state machine 102
Register Server Object state machine 125
Server Authentication state machine 252

T

Tcl/Tk 379
test application 379
application log area 465
components menu 439
config tab 460
configuration 380
E-DNS tab 449
general tab 442
help menu 439
installation 379
lists tab 446
log tab 448
main window 382
menu bar, toolbar 382
objects window 440
remotes menu 413
server components 467
setting identifier tab 451
tab area 441
Tcl/Tk 379
toolbar options 413
tools menu 382
user interface 381
TLS transport
TLS and SIP 206
TLS configuration parameters 216
TLS connection 210
TLS connection API 210
TLS connection establishment 205
TLS connection events 211
TLS connection state machine 215
TLS connection states 213
TLS engine 206
TLS engine API 207
TLS Stack objects 206
transport layer 177
client connection state machine 193
connection API 186
connection control functions 188
connection creation, initialization 187
connection events 189
connection parameters 186
connection states 190
connection capacity percent
connection capacity percent 179
connections 186
dynamic local address (DLA) 220
API 221
IPv6 addresses 217
and SIP Stack 218
and transport address structure 218
compiling SIP Stack with IPv6 218
initializing SIP Stack with IPv6 218
scope ID 218
syntax 217
persistency levels 179
persistency level configuration 181
SIP Stack object persistency APIs, events 182
transaction persistency 180
transaction user persistency 181
undefined persistency 180
persistent connection handling 177
application connections 179
connection hash 178
connection owner 178
connection termination rule 178
raw buffer monitoring
raw buffer events 219
raw buffer, message monitoring 219
server connection reuse 200, 203, 204
server connections 195
API 196
closing 198
events 196
server connection state machine 198
states 197
TCP transport 205
TLS transport 205
TLS Stack objects 206

transport layer API
connection objects 185
handles 186
transport manager 186

X

XML Library 269
handles 271
objects
XML Attribute object 271
XML Document object 270
XML Manager object 270
XML Tag object 270
XML XPath object 271
utility functions 274
XML Library API 272
parameters 272