Developer Guide for SIP Transparency and Normalization

Cisco Unified Communications Manager Release 9.0(1)

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

Text Part Number: OL-27145-01

Developer Guide for SIP Transparency and Normalization Cisco Unified Communications Manager 9.0(1)

CONTENTS

CHAPTER 11

Preface

This document describes the customization process of the SIP messages on Cisco Unified CM- Session Manager Edition (CUCM-SME). It also describes the details on Lua environment available on CUCM-SME and APIs to support SIP Transparency and Normalization functionality

The preface covers these topics:

Audience

This document provides information for developers, vendors, and customers who are developing applications or products that integrate with Cisco Unified Communications Manager 8.5(1)) using SIP Transparency and Normailzation.

Organization

This document includes the following sections.

Chapter	Description
Chapter 1, “Overview”	Provides an overview of Lua Environment, the interface used for customizing Session Management (SM) behavior for a particular deployment.
Chapter 2, “SIP and SDP Normalization”	Provides introduction to the Lua scripting environment APIs used for manipulating the SIP message and any associated Session Description Protocol (SDP).
Chapter 3, “SIP Messages APIs”	Explains the Lua scripting environments SIP Message APIs that allows messages to be manipulated.
Chapter 4, “SDP APIs”	Explains the APIs associated with Session Description Protocol (SDP) content bodies.

Chapter	Description
Chapter 5, “SIP Pass Through APIs”	Explains the pass through object provided APIs that allows information to be passed from one call leg to the other.
Chapter 6, “SIP Utility APIs”	Explains the APIs that allows data strings to be manipulated
Chapter 7, “SIP URI APIs”	Explains the APIs that allows a parsed SIP URI to be manipulated.
Chapter 8, “Trace APIs”	Explains the how tracing allows the script writer to produce traces from within the script. This must only be used for debugging the script.
Chapter 9, “Script Parameters API”	Explains how the Script parameters allows the script writer to obtain trunk specific or line specific configuration parameter values.
Chapter 10, “SIP Transparency”	Provides introduction to the Lua scripting environment APIs used for SIP Transparency.

Conventions

This document uses the following conventions:

Convention	Description
boldface font	Commands and keywords are in **boldface**.
italic font	Arguments for which you supply values are in *italics*.
[ ]	Elements in square brackets are optional.
{ x | y | z }	Alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in brackets and separated by vertical bars.
string	A nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.
`screen` font	Terminal sessions and information the system displays are in screen font.
`boldface screen` font	Information you must enter is in **`boldface screen`** font.
`italic screen` font	Arguments for which you supply values are in *`italic screen`* font.
Æ	This pointer highlights an important line of text in an example.
^	The symbol ^ represents the key labeled Control—for example, the key combination ^D in a screen display means hold down the Control key while you press the D key.
< >	Nonprinting characters, such as passwords are in angle brackets.

Notes use the following conventions:

Note Means reader take note. Notes contain helpful suggestions or references to material not covered in the publication.

Caution Means reader be careful. In this situation, you might do something that could result in equipment damage or loss of data.

Tip Means the following information might help you solve a problem.

Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, submitting a service request, and gathering additional information, see the monthly What's New in Cisco Product Documentation, which also lists all new and revised Cisco technical documentation, at:

http://www.cisco.com/en/US/docs/general/whatsnew/whatsnew.html

Subscribe to the What's New in Cisco Product Documentation as a Really Simple Syndication (RSS) feed and set content to be delivered directly to your desktop using a reader application. The RSS feeds are a free service and Cisco currently supports RSS Version 2.0.

Developer Support

The Developer Support Program provides formalized support for Cisco Systems interfaces to enable developers, customers, and partners in the Cisco Service Provider solutions Ecosystem and Cisco Partner programs to accelerate their delivery of compatible solutions.

The Developer Support Engineers, an extension of the product technology engineering teams, have direct access to the resources that are necessary to provide expert support in a timely manner.

For additional information on this program, refer to the Developer Support Program Web Site at www.cisco.com/go/developer support/.

Developers who use the SIP Line Messaging Guide are encouraged to join the Cisco Developer Support Program. This new program provides a consistent level of support while leveraging Cisco interfaces in development projects.

Cisco Technical Assistance Center (TAC) support does not include SIP Developer support and is limited to Cisco installation/configuration and Cisco-developed applications. For more information about the Cisco Developer Support Program, contact Cisco at developer-support@cisco.com.

CHAPTER 1

Overview

This guide explains the interface specification used for Customization of SIP messages on Cisco Unified CM- Session Management (SME). It includes details on Lua environment available on Cisco Unified CM-SME and APIs to support SIP Transparency and Normalization functionality.

This chapter describes the following topics:

Introduction

This chapter describes Lua Environment, the interface used for customizing Session Management (SM) behavior for a particular deployment. Lua is a non-proprietary, lightweight scripting language. It is assumed that the reader of this guide has basic familiarity with the Lua scripting language.

Cisco Unified CM provides a set of features called SIP Normalization and Transparency to customize the SIP messages:

- Normalization—It is the process of transforming inbound and outbound messages.
  - For inbound messages, normalization happens after having received the message from the network. The inbound message normalization is used to make the message more palatable to Cisco Unified CM. For example, Cisco Unified CM supports Diversion header for carrying redirecting number information. Some SIP devices connected to Cisco Unified CM use the History-Info header for this purpose. Inbound normalization transforms the History-Info headers into Diversion headers so that Cisco Unified CM recognizes the redirecting information.
  - For outbound messages, normalization happens just prior to sending the message to the network. Thus outbound message normalization is used to make the message more palatable to an external SIP device (e.g. another SIP capable PBX). For example, outbound normalization can be used to transform Diversion headers into History-Info headers so that the external SIP device will recognize the redirecting information.
- Transparency— It allows SIP information to be passed through from one call leg to another even though Cisco Unified CM is a Back to Back User Agent (B2BUA).

The Normalization and Transparency features is exposed by a script which is associated with a SIP trunk or a SIP line. The scripts manifest themselves as a set of message handlers that operate on inbound and outbound SIP messages. For normalization, the script manipulates almost every aspect of a SIP message including:

- The request URI
- The response code and phrase
- SIP headers
- SIP parameters
- Content bodies
- SDP

For transparency, the script passes through almost any information in a SIP message including:

- SIP headers
- SIP parameters
- Content bodies

This guide describes the scripting environment and APIs used to manipulate and pass through SIP message information.

Scripting Environment

The interface for customizing Cisco Unified CM SIP Trunk behavior is provided by a scripting language called Lua. Lua is an open source, lightweight scripting language.

The Lua Environment available for Cisco Unified CM (SM) is a restricted sub-set of Lua. In addition to the basic capabilities provided by Lua, like:

- lexical conventions
- values and types
- variables
- statements
- expressions, etc

Lua also provides some basic libraries, like:

- base
- co-routine
- modules
- string manipulation
- table manipulation
- mathematical functions
- OS facilities
- IO facilities and
- debug capabilities

However, the Cisco SIP Lua Environment available on SM only supports the string library in its entirety and a subset of the base library. The other libraries are not supported.

For the base library, the following is supported:

- ipairs
- pairs
- next
- unpack
- error
- type
- tostring

The Cisco SIP Lua Environment provides a global environment for the scripts to use, it does not expose the default Lua global environment (i.e. _G) to the scripts.

The Lua script provides a set of call back functions called **message handlers** to manipulate SIP messages in the context of SM environment. The name of the message handler indicates the handler that is invoked for a particular SIP message. For example, the script's “inbound_INVITE” message handler is invoked when an inbound INVITE is received by Cisco Unified CM. The message handlers receives a single parameter called **msg** representing a SIP Message. The Lua script uses the APIs defined in Cisco SIP Message library to access and manipulate the **msg** parameter.

The following part of the guide describes the details on the **message handler** construct. The next section has details on the Cisco SIP Message library API’s.

Let's take a look at an example script that simply removes the "Cisco-Guid" header in an outbound INVITE. The script is shown with line numbers on left for ease of describing the script.

Simple Script: M.lua

```
1.  M = {}
2.  function M.inbound_INVITE(msg)
3.      msg:convertHIToDiversion()
4.  end
5.  function M.outbound_INVITE(msg)
6.      msg:removeHeader("Cisco-Guid")
7.  end
8.  return M
```

There are three important parts to the above script:

1. Module Initialization—The first line of the script creates a Lua table called 'M' and initializes it to be empty. This table contains set of callback functions provided by this script. The variable **M** is a Lua table and is also the name of the module.
2. Message Handler Definitions—Lines 2-4 defines an inbound INVITE **message handler**. This callback function is executed when an inbound INVITE is received out on the SIP trunk or SIP line associated with this script. In this example, the message handler invokes an API to convert History-Info headers into Diversion headers.

   Lines 5-6 defines an outbound INVITE **message handler**. This callback function is executed when an outbound INVITE is sent out on the SIP trunk or SIP line associated with this script. In this example, the message handler invokes an API to remove the "Cisco-Guid" header.

   The script can define multiple message handlers. The name of the message handler dictates which message handler is invoked (if any) for a given SIP message.

3. Returning the Module—The last line returns the name of the module. This line is absolutely required. This is how the Cisco SIP Lua Environment finds the message handlers associated with the script.

Message Handlers

The Lua script provides a set of call back functions called message handlers to manipulate SIP messages in the context of SM environment. The name of the message handler indicates which handler is invoked for a particular SIP message.

Naming

The naming rules for message handlers dictate which message handler is invoked for a given SIP message. Various SIP messages by specification, are split into *requests* and *responses*.

- For *requests*–the message handler is named according to the message direction and the SIP request method name. The method name is the one in the 'request line' of SIP message.

  ```
  <direction>_<method>
  ```

  Example

  ```
  inbound_INVITE
  outbound_UPDATE
  ```

- For *responses*–the message handler is named according to the message direction, the response code, and the SIP method . For responses, the method name is obtained from the **CSeq** header of SIP message.

  ```
  <direction>_<response code>_<method>
  ```

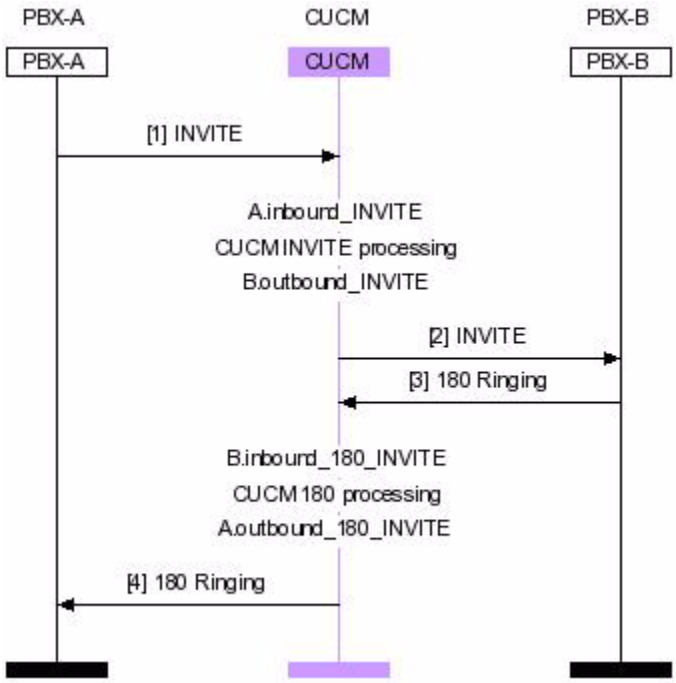
  Example

  ```
  inbound_183_INVITE
  inbound_200_INVITE
  outbound_200_UPDATE
  ```

Use Case

Consider the case where a Cisco Unified CM-SME is connected to PBX-A and PBX-B via two trunks. A script that returns module **A** is attached to trunk connecting to PBX-A. Similarly, a script that returns module **B** is attached to trunk connecting to PBX-B.

The following handlers are executed for an INVITE dialog.



Wild cards

The message handler names also support wild carding. The wild card support is dependent on whether the message is **request** or **response** SIP message.

Request messages

A wildcard **ANY** can be used in place of <method>. The <direction> does not support wild card.

Refer to Table 1-1 on page 1-5 for valid request message handler names

Table 1-1 Valid Request Message Handler Names

Message Handler	Description
M.inbound_INVITE	This message handler is invoked for all inbound INVITE messages including initial INVITEs and reINVITEs.
M.inbound_ANY	This message handler is invoked for all inbound requests.
M.outbound_ANY	This message handler is invoked for all outbound requests.

Response messages

A wildcard **ANY** can be used in place of <method> and/or <response code>. The <direction> does not support wild card. Additionally, a wildcard character **X** can be used in <response code>.

Refer to Table 1-2 on page 1-6 for valid response message handler names.

Table 1-2 Valid Response Message Handler Names

Message Handler	Description
M.inbound_18X_INVITE	This message handler is invoked for all inbound 18X responses including 180, 181, 182, and 183.
M.inbound_ANY_INVITE	This message handler is invoked for all inbound responses for an INVITE request including all 18X, 2XX, 3XX, 4XX, 5XX, and 6XX responses.
M.outbound_ANY_ANY	This message handler is invoked for all outbound responses regardless of the request method.

Rules to Pick Message Handler

Cisco Unified CM uses the following rules to find the **message handler** for a given message:

1. Message handlers are case-sensitive.
2. The *direction* is either *inbound* or *outbound*.
3. The direction is always written as lowercase.
4. The message direction is relative to SM.

   Note The message direction is completely separate from *dialog direction* in SIP.

   Example

   inbound_INVITE is valid handler name; whereas InBound_INVITE is NOT a valid handler name
5. The method name obtained from SIP message is converted to uppercase for the purpose of finding the appropriate message handler in the script.
6. CUCM-SME uses a *longest-match* criterion to find the correct message handler.

   Example

   Assume a script has two message handlers; inbound_ANY_ANY and inbound_183_INVITE. When a 183 response is received by the Cisco Unified CM, the inbound_183_INVITE handler will be executed since it is most explicit match.

Note Although inbound or outbound is supported with ANY (response code) and ANY (method), we do not currently support the ANY (method) wildcard with specific response codes.

In other words, the following message handlers are valid:

```
inbound_ANY_ANY
outbound_ANY_ANY
```

But these are invalid:

```
inbound_401_ANY
outbound_200_ANY
```

etc.

CHAPTER 2

SIP and SDP Normalization

The Lua scripting environment establishes underlying objects for manipulating the SIP message and any associated Session Description Protocol (SDP). The script need not worry about these objects and accesses these objects using the following set of APIs:

- SIP Messages APIs—These APIs allows the script to manipulate the SIP message in a variety of ways.
- SDP APIs—These APIs allow the script to manipulate the SDP in a variety of ways.
- SIP Pass Through APIs—These APIs allow the script to pass information from one call leg to the other.
- SIP Utility APIs— These APIs provide useful utilities for the script to manipulate data including the parsing of a Uniform Resource Identifier (URI) into a SIP URI object.
- SIP URI APIs—These APIs allow the script to manipulate the parsed SIP URI object.
- Trace APIs—These APIs allows the script to enable and disable tracing, determine if tracing is enabled, and to produce traces.
- Script Parameters API—This parameter allows the script writer to obtain trunk or line specific configuration parameter values.

CHAPTER 3

SIP Messages APIs

The Lua scripting environment provides a set of APIs that allows messages to be manipulated. These APIs are explained under the following categories:

Manipulating the Request or Response line

getRequestLine

```
getRequestLine() returns the method, request-uri, and version
```

This method returns three values:

- The method name
- The request-uri, and
- The protocol version.

If this API is invoked for a response, it triggers a run-time error.

Example: Set the values of local variables for the method, request-uri, and the protocol version.

Script

```
M = {}

function M.outbound_INVITE(msg)
    local method, ruri, ver = msg:getRequestLine()
end

return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
```

Output/Result

```
Local variables method, ruri, and version are set accordingly:
method == "INVITE"
ruri == "sip:1234@10.10.10.1"
version == "SIP/2.0"
```

getRequestUriParameter

```
getRequestUriParameter(parameter-name) returns the URI parameter-value
```

Given the string name of a request URI parameter, this function returns the value of the specified URI parameter.

- If the URI parameter is a tag=value format, the value on the right of the equal sign is returned.
- If the URI parameter is a tag with no value, the value will be a blank string (i.e. "").
- If the URI parameter does not exist in the first header value, nil will be returned.
- If this method is invoked for a response message, nil will be returned.

Example: Set a local variable to the value of the user parameter in the request-uri.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local userparam = msg:getRequestUriParameter("user")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1;user=phone SIP/2.0
```

Output/Results

Local variable userparam is set to the string "phone"

setRequestUri

```
setRequestUri(uri)
```

This method sets the request-uri in the request line. If this API is invoked for a response, it triggers a run-time error.

Example: Set the request-uri to tel:1234.

Script

```
M = {}
function M.outbound_INVITE(msg)
    msg:setRequestUri("tel:1234")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
```

Output/Result

```
INVITE tel:1234 SIP/2.0
```

getResponseLine

```
getResponseLine() returns version, status-code, reason-phrase
```

This method returns three values: the protocol version (string), the status-code (integer), and the reason-phrase (string). If this API is invoked for a request, it triggers a run-time error.

Example: Set local variables to the values of the protocol version, the status code, and the reason phrase.

Script

```
M = {}
function M.outbound_200_INVITE(msg)
    local version, status, reason = msg:getResponseLine()
end
return M
```

Message

```
SIP/2.0 200 Ok
.
.
CSeq: 102 INVITE
```

Output/Result

```
Local variables method, ruri, and version are set accordingly:
version == "SIP/2.0"
status == 200
reason == "Ok"
```

setResponseCode

```
setResponseCode(status-code, reason-phrase)
```

This method sets the status-code and reason-phrase. Either value is allowed to be nil. If nil is specified for the value, the existing value stored in the message is used. If this API is invoked for a request, it triggers a run-time error.

Example: Change an outgoing 404 response to a 604 response.

Script

```
M = {}
function M.outbound_404_INVITE(msg)
    msg:setResponseCode(604, "Does Not Exist Anywhere")
end
return M
```

Message

```
SIP/2.0 404 Not Found
.
.
CSeq: 102 INVITE
```

Output/Results

```
SIP/2.0 604 Does Not Exist Anywhere
.
.
CSeq: 102 INVITE
```

Manipulating Headers

allowHeader

```
allowHeaders is a Lua table specified by the script writer
```

Cisco Unified CMs default behavior with respect to the headers that it does not recognize is to ignore such headers. In some cases it is desirable to use such a header during normalization or to pass the header transparently. By specifying the header name in the **allowHeaders** table, the script enables Cisco Unified CM to recognize the header and it is enough for the script to use it locally for normalization or pass it transparently.

Example:

The **allowHeaders** specifies the `History-Info`. Without it, the incoming History-Info headers gets dropped before script processing. Thus the **convertHIToDiversion** does not produce any useful results. This script also allows a fictitious header called `x-pbx-id`. This illustrates that the allowHeaders is a table of header names.

Script

```
M = {}
M.allowHeaders = {"History-Info", "x-pbx-id"}
function M.inbound_INVITE(msg)
    msg:convertHIToDiversion()
end
return M
```

Message

```
INVITE sip:1001@10.10.10.1 SIP/2.0
.
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:1001@10.10.10.1>;index=1.1
```

Result

```
INVITE sip:1001@10.10.10.1 SIP/2.0
.
Diversion: <sip:2401@10.10.10.2>;reason=unconditional;counter=1
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:1001@10.10.10.1>;index=1.1
```

getHeader

```
getHeader(header-name) returns header-value or nil
```

Given the string name of a header, this method returns the value of the header. For multiple headers with the same name, the values a concatenated together and comma separated:

Example: Set a local variable to the value of the Allow header.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local allow = msg:getHeader("Allow")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>;
.
```

```
Allow: UPDATE
Allow: Ack,Cancel,Bye,Invite
```

Output/Results

```
Local variable allow set to the string "UPDATE,Ack,Cancel,Bye,Invite"
```

getHeaderValues

```
getHeaderValues(header-name) returns a table of header values
```

Given the string name of a header, this function returns the comma separated values of the header as an indexed table of values. Lua indexing is 1 based: If there are n header values, the first value is at index 1 and the last value is at index n. Lua's **ipairs** function can be used to iterate through the table

Example: Create a local table containing the values of the History-Info header.

Script

```
M = {}
M.allowHeaders = {"History-Info"}
function M.inbound_INVITE(msg)
    local history_info = msg:getHeaderValues("History-Info")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>;
.
History-Info: <sip:UserB@hostB?Reason=sip;cause=408>;index=1
History-Info: <sip:UserC@hostC?Reason=sip;cause=302>;index=1.1
History-Info: <sip:UserD@hostD>;index=1.1.1
```

Output/Results

```
Local variable history_info is  set to
{
  "<sip:UserB@hostB?Reason=sip;cause=408>;index=1",
  "<sip:UserC@hostC?Reason=sip;cause=302>;index=1.1",
  "<sip:UserD@hostD>;index=1.1.1"
}

In other words:
history_info[1] == "<sip:UserB@hostB?Reason=sip;cause=408>;index=1"
history_info[2] == "<sip:UserC@hostC?Reason=sip;cause=302>;index=1.1"
history_info[3] == "<sip:UserD@hostD>;index=1.1.1"
```

getHeaderValueParameter

```
getHeaderValueParameter(header-name, parameter-name) returns the parameter-value
```

Given the names of a header and a header parameter , this method returns the value of the specified header parameter. For multiple headers with same name, only the first header value is evaluated.

- If the header parameter is a tag=value format, the value on the right of the equal sign is returned.

- If the header parameter is a tag with no value, the value will be a blank string (i.e. "").
- If the header parameter does not exist in the first header value, nil will be returned.

Example: Set a local variable to the value of the header parameter named tag.

Script

```
M = {}
function M.outbound_180_INVITE(msg)
    local totag = msg:getHeaderValueParameter("To", "tag")
end
return M
```

Message

```
SIP/2.0 180 Ringing
.
To: <sip:1234@10.10.10.1>;tag=32355SIPpTag0114
```

Output/Results

```
Local variable totag is set to the string "32355SIPpTag0114"
```

getHeaderUriParameter

```
getHeaderUriParameter(header-name, parameter-name) returns the URI parameter-value
```

Given the string name of a header, this method returns the value of the specified URI parameter. For multiple headers with same name, only the first header value is evaluated.

- If the URI parameter is a tag=value format, the value on the right of the equal sign is returned.
- If the URI parameter is a tag with no value, the value will be a blank string (i.e. "").
- If the URI parameter does not exist in the first header value, nil will be returned.

Example: Set a local variable to the value of the uri parameter named user.

Script

```
M = {}
function M.outbound_180_INVITE(msg)
   local userparam = msg:getHeaderUriParameter("To", "user")
end
return M
```

Message

```
SIP/2.0 180 Ringing
.
To: <sip:1234@10.10.10.1;user=phone>;tag=32355SIPpTag0114
```

Output/Results

```
Local variable userparam is set to the string "phone"
```

addHeader

```
addHeader(header-name, header-value)
```

Given the string name of a header and a value, this method appends the value at the **end** of the list of header values.

Example: Add INFO to the Allow header.

Script

```
M = {}
function M.outbound_INVITE(msg)
    msg:addHeader("Allow", "INFO")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>;
Allow: Ack,Cancel,Bye,Invite
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>;
Allow: Ack,Cancel,Bye,Invite,INFO
```

addHeaderValueParameter

```
addHeaderValueParameter(header-name, parameter-name [,parameter-value])
```

Given a header-name, parameter-name, and optional parameter value, this method locates the specified header in the SIP message and adds the specified header parameter and optional parameter value to the header. The original header is replaced by the modified header value in the SIP message. If there are multiple header values for the specified header, only the first header value is modified. No action is taken if no instance of the specified header is located in the SIP message.

Example: Add the color=blue header parameter to the outbound Contact header.

Script

```
M = {}
function M.outbound_INVITE(msg)
    msg:addHeaderValueParameter("Contact", "color", "blue")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
Contact: <sip:1234@10.10.10.2>
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
Contact: <sip:1234@10.10.10.12>;color=blue
```

addHeaderUriParameter

```
addHeaderUriParameter(header-name, parameter-name [,parameter-value])
```

Given a header-name, parameter-name, and an optional parameter-value, this method locates the specified header in the SIP message and adds the specified header URI parameter and optional URI parameter value to the contained URI. The original header is replaced by the modified header value in the SIP message. If there are multiple header values for the specified header, only the first header value is modified. No action is taken if no instance of the specified header is located in the SIP message or if a valid URI is not found within the first instance of the specified header.

Example: Add user=phone parameter to the P-Asserted-Identity uri.

Script

```
M = {}
function M.inbound_INVITE(msg)
    msg:addHeaderUriParameter("P-Asserted-Identity", "user", "phone")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1;user=phone>
```

modifyHeader

```
modifyHeader(header-name, header-value)
```

Given the string name of a header and a value, this method over writes the existing header value with the value specified. This is effectively like invoking removeHeader followed by addHeader.

Example: Remove the display name from the P-Asserted-Identity header in outbound INVITE messages.

Script

```
M = {}
function M.outbound_INVITE(msg)
    -- Remove the display name from the PAI header
    local pai = msg:getHeader("P-Asserted-Identity")
    local uri = string.match(pai, "(<.+>)")
    msg:modifyHeader("P-Asserted-Identity", uri)
```

```
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: "1234" <1234@10.10.10.1>
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <1234@10.10.10.1>
```

applyNumberMask

```
applyNumberMask(header-name, mask)
```

Given a header-name and number mask this method locates the specified header in the SIP message and applies the specified number mask to the URI contained within the header. If a URI is successfully parsed from the header value, the number mask is applied to the user part of the parsed URI. The original header is replaced by the modified header value in the SIP message. If there are multiple instances of the specified header, only the first instance of the header is modified.

No action is taken if any of the following conditions exist.

1. No instance of the specified header is located in the SIP message
2. A valid URI is not found within the first instance of the specified header
3. The specified mask parameter is an empty string

Application of the number mask

The mask parameter defines the transformation to be applied to the user part of the header URI. Wildcard characters are specified in the mask parameter as an upper case X. For example, if the mask "+1888XXXXXXX" is specified, "X" is used as the insertion character of the mask. If this mask is applied to the example user part "4441234", the resulting sting is "+18884441234".

If the number of characters found in the user part to be masked is less than the number of wildcard characters in the mask, the left most wildcard characters will left as "X". Applying the previous mask to the example user part "1234" yields the resulting string "+1888XXX1234". If the number of characters found in the user part to be masked is greater than the number of wildcard characters in the mask, the left most characters of the user part are truncated. For example, if the mask "+1888XXXX" is applied to the user part "4441234", the resulting string is "+18881234".

Example: Apply a number mask the number in the P-Asserted-Identity header for inbound INVITE messages.

Script

```
M = {}
function M.inbound_INVITE(msg)
    msg:applyNumberMask("P-Asserted-Identity", "+1919476XXXX")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:+19194761234@10.10.10.1>
```

convertDiversionToHI

```
convertDiversionToHI()
```

Cisco Unified CM supports sending the Diversion header for handling redirecting information. For example, when a call is forwarded by Cisco Unified CM. In such instances, the SIP message may contain a Diversion header which is used by Cisco Unified CM to send the redirecting number. However, many SIP servers use the History-Info header for this purpose instead of the Diversion header. This API is used to convert the Diversion header into History-Info headers.

Note Some implementations require use of the raw header APIs (e.g. getHeader, addHeader, modifyHeader, and removeHeader) instead of or in addition to this API.

Example: Convert Diversion headers into History-Info headers for outbound INVITE messages. Then remove the Diversion header

Script

```
M = {}
function M.outbound_INVITE(msg)
    if msg:getHeader("Diversion")
    then
        msg:convertDiversionToHI()
        msg:removeHeader("Diversion")
    end
end
return M
```

Message

```
INVITE sip:2400@10.10.10.2 SIP/2.0
.
Diversion: <sip:1002@10.10.10.1>;reason=unconditional;privacy=off;screen=yes
```

Output/Results

```
INVITE sip:2400@10.10.10.2 SIP/2.0
.
History-Info: <sip:1002@10.10.10.1?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:2400@10.10.10.2>;index=1.1
```

convertHIToDiversion

```
convertHIToDiversion()
```

Cisco Unified CM supports receiving the Diversion header for handling redirecting information. For example, when a call is forwarded before reaching Cisco Unified CM. In such a case, the SIP message may contain a Diversion header which is used by Cisco Unified CM to establish the redirecting number. However, many SIP servers use the History-Info header for this purpose instead of the Diversion header.

Note Some implementations may require use of the raw header APIs (e.g. getHeader, addHeader, modifyHeader, and removeHeader) instead of or in addition to this API.

Example

The allowHeaders must specify "History-Info". Without it, the incoming History-Info headers will get dropped before script processing. Thus the call to convertHIToDiversion won't produce any useful results.

Script

```
M = {}
M.allowHeaders = {"History-Info"}
function M.inbound_INVITE(msg)
    msg:convertHIToDiversion()
end
return M
```

Message

```
INVITE sip:1001@10.10.10.1 SIP/2.0
.
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:1001@10.10.10.1>;index=1.1
```

Result

```
INVITE sip:1001@10.10.10.1 SIP/2.0
.
Diversion: <sip:2401@10.10.10.2>;reason=unconditional;counter=1
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:1001@10.10.10.1>;index=1.1
```

removeHeader

```
removeHeader(header-name)
```

Given the string name of a header, this method removes the header from the message.

Example: Remove the Cisco-Guid header from outbound INVITE messages.

```
M = {}
function M.outbound_INVITE(msg)
    msg:removeHeader("Cisco-Guid")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: "1234" <1234@10.10.10.1>
Cisco-Guid: 1234-4567-1234
Session-Expires:  1800
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: "1234" <1234@10.10.10.1>
Session-Expires:  1800
.
```

removeHeaderValue

```
removeHeaderValue(header-name, header-value)
```

Given the string name of a header and a header-value, this method removes the header-value from the message. If the value was the only header-value, the header itself is removed.

Example: Remove X-cisco-srtp-fallback from the Supported header for outbound INVITE messages.

Script

```
M = {}
function M.outbound_INVITE(msg)
    msg:removeHeaderValue("Supported", "X-cisco-srtp-fallback")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: "1234" 1234@10.10.10.1
Supported: timer, replaces, X-cisco-srtp-fallback
Session-Expires:  1800
```

Output/Results

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: "1234" 1234@10.10.10.1
Supported: timer, replaces
Session-Expires:  1800
```

Manipulating SDP

getSdp

```
getSdp() returns string
```

This method returns the SDP as a Lua string. It returns nil if the message does not contain SDP.

Example: Establish a local variable which contains the SDP (or nil if there is no SDP).

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
end
return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
Via: SIP/2.0/TCP 172.18.197.88:5080;branch=z9hG4bK4bae847b2
From: <sip:579@172.18.197.88>;tag=9f12addc-04fc-4f0f-825b-3cd7d6dd8813-25125665
To: <sip:901@rawlings.cisco.com>
Date: Thu, 28 Jan 2010 01:09:39 GMT
Call-ID: cd80d100-b601e3d3-15-58c512ac@172.18.197.88
.
.
Max-Forwards: 69
Content-Type: application/sdp
Content-Length: 214

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24580 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Results

The variable sdp is set to the following string. Note that this string will contain embedded carriage return and line feed characters (i.e. "\r\n").

```
v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24580 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

setSdp

```
setSdp(sdp)
```

This method stores the specified string in the SIP Message as the SDP content body. The Content-Length header of SIP message is automatically updated to the length of the string.

Example: Remove the a=ptime lines from the outbound SDP.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        -- remove all ptime lines
        sdp = sdp:gsub("a=ptime:%d*\r\n", "")
        -- store the updated sdp in the message object
        msg:setSdp(sdp)
    end
end
return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
Via: SIP/2.0/TCP 172.18.197.88:5080;branch=z9hG4bK4bae847b2
From: <sip:579@172.18.197.88>;tag=9f12addc-04fc-4f0f-825b-3cd7d6dd8813-25125665
To: <sip:901@rawlings.cisco.com>
Date: Thu, 28 Jan 2010 01:09:39 GMT
Call-ID: cd80d100-b601e3d3-15-58c512ac@172.18.197.88
.
.
Max-Forwards: 69
Content-Type: application/sdp
Content-Length: 214

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24580 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Results

The Final SIP message after the execution of above script will look like below:

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
Via: SIP/2.0/TCP 172.18.197.88:5080;branch=z9hG4bK4bae847b2
From: <sip:579@172.18.197.88>;tag=9f12addc-04fc-4f0f-825b-3cd7d6dd8813-25125665
To: <sip:901@rawlings.cisco.com>
Date: Thu, 28 Jan 2010 01:09:39 GMT
Call-ID: cd80d100-b601e3d3-15-58c512ac@172.18.197.88
.
.
Max-Forwards: 69
```

```
Content-Type: application/sdp
Content-Length: 214


v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24580 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

removeUnreliableSdp

```
removeUnreliableSdp()
```

This method removes SDP from a 18X message. Before removing the SCDP, it checks to ensure the messages does not require PRACK.

Note The adjustments to the Content-Length header are made automatically when this API results in the SDP actually being removed. The Content-Type header is automatically removed if there is no other content body in the message.

Example: Remove SDP from inbound 180 messages.

Script

```
M = {}
function M.inbound_180_INVITE(msg)
    msg:removeUnreliableSdp()
end
return M
```

Message

```
SIP/2.0 180 Ringing
.
Content-Type: application/sdp
Content-Length: 214

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24580 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Results

The Final SIP message after the execution of above script will look like below:

```
SIP/2.0 180 Ringing
.
Content-Length: 0
```

Manipulating Content Bodies

getContentBody

```
getContentBody(content-type) returns the content-body, content-disposition,
content-encoding, and content-language of the specified content-type
```

Given the string name of a content-type, this function returns the content-body, content-disposition, content-encoding, and content-language. The content-disposition, content-encoding, and content-language are optional and may not have been specified in the message. If the particular header is not specified in the message, nil is returned for its value.

If the content-body is not in the message, nil values are returned for all returned items.

Example: Establish local variables containing the content information for an outbound INFO message.

Script

```
M = {}
function M.inbound_INFO(msg)
    local b = msg:getContentBody("application/vnd.nortelnetworks.digits")
end
return M
```

Outbound Message

```
INFO sip: 1000@10.10.10.1 SIP/2.0
Via: SIP/2.0/UDP 10.10.10.57:5060
From: <sip:1234@10.10.10.57>;tag=d3f423d
To: <sip:1000@10.10.10.1>;tag=8942
Call-ID: 312352@10.10.10.57
Cseq: 5 INFO
Content-Type: application/vnd.nortelnetworks.digits
Content-Length: 72

p=Digit-Collection
y=Digits
s=success
u=12345678
i=87654321
d=4
```

Output/Results

```
Local variable b is set to the string:
```

```
p=Digit-Collection
y=Digits
s=success
u=12345678
i=87654321
d=4
```

addContentBody

```
addContentBody(content-type, content-body [,content-disposition [,content-encoding
[,content-language]]])
```

Given the content-type and content-body, this API will add the content body to the message and make the necessary changes to the content-length header. The script may optionally provide disposition, encoding, and language as well.

Example: Add a plain text content body to outbound UPDATE messages.

Script

```
M = {}
function M.outbound_UPDATE(msg)
    msg:addContentBody("text/plain", "d=5")
end
return M
```

Message Before Normalization

```
UPDATE sip: 1000@10.10.10.1 SIP/2.0
.
Content-Length: 0
```

Message After Normailization

```
UPDATE sip: 1000@10.10.10.1 SIP/2.0
.
Content-Length: 3
Content-Type: text/plain
d=5
```

removeContentBody

```
removeContentBody(content-type)
```

Given the content-type, this API will remove the associated content body from the message and make the necessary changes to the content-length header.

Example: Remove the text/plain content body from outbound UPDATE messages.

Script

```
M = {}
function M.outbound_UPDATE(msg)
    msg:removeContentBody("text/plain")
end
return M
```

Message Before Normailization

```
UPDATE sip: 1000@10.10.10.1 SIP/2.0
.
Content-Length: 3
Content-Type: text/plain

d=5
```

Message After Normailization

```
UPDATE sip: 1000@10.10.10.1 SIP/2.0
.
Content-Length: 0
```

Blocking Messages

- block

block

```
block()
```

This method can block an inbound or outbound unreliable 18X message. For inbound blocking, CUCM behaves as if it never received the message. For outbound message blocking, CUCM doesn't send the message to the network.

Note If PRACK is enabled and therefore the message is actually a reliable 18X, CUCM will effectively ignore the call to block(). The message will not be blocked and an error SDI trace will be produced.

Example: Block outbound 183 messages.

Script

```
M = {}
function M.outbound_183_INVITE(msg)
    msg:block()
end
return M
```

Message

```
SIP/2.0 183 Progress
Via: SIP/2.0/TCP 172.18.199.186:5060;branch=z9hG4bK5607aa91b
From: <sip:2220@172.18.199.186>;tag=7cae99f9-2f13-4a4d-b688-199664cc38a4-32571707
To: <sip:2221@172.18.199.100>;tag=3affe34f-6434-4295-9a4b-c1fe2b0e03b6-21456114
.
.
```

Output/Results

```
The 183 message is not sent to the network.
```

Transparency

- getPassThrough

getPassThrough

```
getPassThrough() returns a per message pass through object
```

This method returns a SIP Pass Through object. It can be invoked for inbound messages. Information added to the pass through object is automatically merged into the outbound message generated by the other call leg. The automatic merging of this information takes place before outbound normalization (if applicable).

Refer to the SIP Pass Through APIs for use of the pass through object and examples.

Maintaining Per-Dialog Context

- getContext

getContext

```
getContext() returns a per call context
```

This method returns a per call context. The context is a Lua table. The script writer can store and retrieve data from the context across various message handlers used throughout the life of the dialog. The script writer does not need to be concerned with releasing the context at the end of the dialog. Cisco Unified CM will automatically release the context when the dialog ends.

Example:

This script uses context to store the History-Info headers received in an INVITE. When the response to that INVITE is sent, the stored headers are retrieved and added to the outgoing responses. The "clear" parameter is used to prevent these headers from being appended to subsequent reINVITE responses.

Script

```
M = {}
M.allowHeaders = {"History-Info"}
function M.inbound_INVITE(msg)
    local history = msg:getHeader("History-Info")
    if history
    then
        msg:convertHIToDiversion()
        local context = msg:getContext()
        if context
        then
            context["History-Info"] = history
        end
    end
end

local function includeHistoryInfo(msg, clear)
    local context = msg:getContext()
```

```
    if context
    then
       local history = context["History-Info"]
       if history
       then
           msg:addHeader("History-Info", history)

           if clear
           then
             context["History-Info"] = nil
           end
       end
    end
end

function M.outbound_18X_INVITE(msg)
    includeHistoryInfo(msg)
end

function M.outbound_200_INVITE(msg)
    includeHistoryInfo(msg, "clear")
end
return M
```

Message Before Normalization

```
INVITE sip:1001@10.10.10.1 SIP/2.0
.
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:1001@10.10.10.1>;index=1.1

SIP/2.0 180 Ringing
.
Content-Length: 0

SIP/2.0 200 OK
.
Content-Type: application/sdp
```

Message After Normalization

```
INVITE sip:1001@10.10.10.1 SIP/2.0
.
Diversion: <sip:2401@10.10.10.2>;reason=unconditional;counter=1
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1
History-Info: <sip:1001@10.10.10.1>;index=1.1


SIP/2.0 180 Ringing
.
Content-Length: 0
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1,
<sip:1001@10.10.10.1>;index=1.1


SIP/2.0 200 OK
.
Content-Type: application/sdp
History-Info: <sip:2401@10.10.10.2?Reason=sip;cause=302;text="unconditional">;index=1,
<sip:1001@10.10.10.1>;index=1.1
.
```

Utilities

isInitialInviteRequest

```
isInitialInviteRequest() returns true or false
```

This method returns true if the message is a request, the method is INVITE, and there is no tag parameter in the To header. Otherwise, false is returned.

Example:

Set a local variable based on whether or not the outbound INVITE is an initial INVITE. In this example, the INVITE is an initial INVITE and therefore, the value would be true for this particular INVITE.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local isInitialInvite = msg:isInitialInviteRequest()
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>
```

Output/Results

```
Local variable isInitialInvite set to 'true'
```

Example:

Set a local variable based on whether or not the outbound INVITE is an initial INVITE. In this example, the INVITE is not an initial INVITE and therefore, the value would be false for this particular INVITE.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local isInitialInvite = msg:isInitialInviteRequest()
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>;tag=1234
```

Output/Results

```
Local variable isInitialInvite set to 'false'.
```

isReInviteRequest

```
isReInviteRequest() returns true or false
```

This method returns true if the message is a request, the method is INVITE, and there is a tag parameter in the To header. Otherwise, false is returned.

Example:

Set a local variable based on whether or not the outbound INVITE is a reINVITE. In this example, the INVITE is a reINVITE and therefore, the value would be true for this particular INVITE.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local isReInvite = msg:isReInviteRequest()
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
To: <sip:1234@10.10.10.1>;tag=112233445566
.
```

Output/Results

```
Local variable isReInvite set to 'true'.
```

Example:

Set a local variable based on whether or not the outbound INVITE is a reINVITE. In this example, the INVITE is not a reINVITE and therefore, the value would be false for this particular INVITE.

Script

```
M = {}
function M.outbound_ANY(msg)
    local isReInvite = msg:isReInviteRequest()
end
return M
```

Message

```
CANCEL sip:1234@10.10.10.1 SIP/2.0
  .
```

Output/Results

```
Local variable isReInvite set to 'false'.
```

getUri

```
getUri(header-name) returns a string or nil
```

Given a header-name, this method locates the specified header in the SIP message and then attempts to retrieve a URI from that header. If there are multiple instances of the specified header, the URI of the first instance of the header is returned. Lua nil is returned if no instance of the specified header is located in the SIP message or if a valid URI is not found within the first instance of the specified header.

Example: Set a local variable to the uri in the P-Asserted-Identity header.

Script

```
M = {}
function M.inbound_INVITE(msg)
    local uri = msg:getUri("P-Asserted-Identity")
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
.
```

Output/Results

```
Local variable uri is set to "sip:1234@10.10.10.1"
```

CHAPTER 4

SDP APIs

This section covers the APIs associated with Session Description Protocol (SDP) content bodies. Unlike the SIP Message object, the SDP is treated as a basic string. This allows the script writer to leverage the power of the Lua string library. In addition to the Lua string library functions, Cisco also provides some APIs to facilitate SDP manipulation. The Cisco APIs are added directly to the string library.

However, before the script can manipulate the SDP, it must obtain the SDP content body from the Lua SIP Message object using the **getSdp()** API provided by the SIP Message object. The script can then use the string library including Cisco's APIs to manipulate the SDP. On modiifcation, the SDP is written back to the SIP Message object using the **setSdp(sdp)** API provided by the SIP Message object. Refer to the following SIP Messages APIs for further information of these APIs.

```
-- Get the SDP content body from the SIP message
local sdp = msg:getSdp()
-- modification of the SDP happens at this point
-- Update the SDP associated with the SIP message
msg: etSdp(sdp)
```

For the purposes of manipulating the SDP, Cisco provides APIs for getting, modifying, adding, inserting, and removing individual lines as well as entire media descriptions. The remainder of this section will discuss the Cisco APIs.

In order to get and manipulate any line in the SDP, we need a way to identify them. The script writer must be capable of affecting invalid SDPs. Hence, knowledge of the structure of SDP for the purposes of normalization must be minimized. It is difficult, if not impossible, to affect invalid SDP. It is also possible to manipulate valid SDP. With these goals in mind, a string based approach is warranted.

The Lua APIs for manipulating SDP can be classified under two broad classes.

- Line Level APIs—This set of APIs is directed towards being able to get and manipulate lines within the SDP while
- Media Description Level APIs—This set of APIs is geared towards manipulating particular media descriptions.

Manipulating session level lines would leverage the Line Level set of APIs while manipulating the media descriptions would leverage the Media Description Level set of APIs.

Note The set of APIs designed for manipulating lines is not limited to session level lines only.

Line Level APIs

Cisco API's

The following Cisco line level APIs are availabe:

- getLine(start-of-line , line-contains)
- modifyLine(start-of-line, line-contains ,new- line)
- addline(new-line)
- insertLineAfter(start-of-line, line-contains, new-line)
- insertLineBefore(start-of-line, line-contains, new-line)
- removeLine(start-of-line, line-contains)

Parameter Description

Parameter	Description
sdp	The **sdp** parameter is a string that contains the SDP, including all control characters (i.e. '\r' and '\n'). The APIs do not enforce any SDP structure on the sdp parameter, It treats it like a string. The script writer can use the APIs to operate on parts of SDP (for example a audio media description).
start-of-line	The **start-of-line** parameter is a string used to match the start of a line in sdp. In typical scripts, for example, "o=" "a=rtpmap:9" can be used to get/modify/insert/remove the first line in sdp which starts with this string. If there are multiple lines that start with this string , only the first line is considered. This parameter MUST be of the form <char>=<string> . i.e. the second character must be the equal sign. If the parameter does not have this format , a 'nil' is returned .
line-contains	The **line-contains** is a string used as an additional match criterion to search within the line starting with 'start-of-line' parameter. If nil is specified, **line-contains** is not used as a match criterion. In that case, only the **start-of-line** parameter will be used for matching. When **line-contains** is not nil, both the **start-of-line** and **line-contains** will be used for matching. If there are multiple lines in SDP that match (both) the search criteria, only the first match will be returned. If this additional matching, using this parameter is not desired , a value of 'nil' without the quotes needs to be passed in.
new-line	The **new-line** is the string that will be added/inserted in the final sdp (returned sdp)

The APIs above treat the parameters as strings. Therefore , they can operate on both session level and media level lines. There is no difference in behavior based on whether a line is at session level or media level. It is assumed that lines are terminated with **\n**(or **\r\n**) when trying to find the matching line .

No characters in the parameters are considered Lua 'Magic characters'. All characters are considered plain and do not have any special meaning.

getLine

```
sdp:getLine(start-of-line, line-contains) returns string
```

This method returns the first line in sdp that starts with **start-of-line** and also has the string **line-contains**.

Example: Set a local variable to the value of the c-line from the SDP.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local ipv4_c_line = sdp:getLine("c=", "IP4")
    end
end
return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24580 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Result

```
Local variable ipv4_c_line is set to "c=IN IP4 172.18.197.88\r\n"
```

modifyLine

```
sdp:modifyLine(start-of-line, line-contains, new-line)
```

This method finds the first line in 'sdp' that starts with **start-of-line** . If **line-contains** parameter is non-nil, it must be present within that line. The matching line (including the line termination characters) is replaced with **new-line** parameter. The resulting sdp is returned.

If no matching line is found, the original sdp is returned unchanged.

Note It is recommended that the caller terminate the **new-line** with **\r\n** (or **\n**) to comply with SDP rules . No enforcement of line termination characters is done by the utility.

Example: The following code will change a= line for G.722 codec to be G722 without the dot.

Script

```
M = {}

function M.inbound_INVITE(msg)
    local sdp = msg:getSdp()

    if sdp
    then
        local g722_line = sdp:getLine("a=","G.722")

        if g722_line
        then
            --Replace G.722 with G722. The dot is special and must be
            --escaped using % when using gsub.
            g722_line = g722_line:gsub("G%.722", "G722")
            sdp = sdp:modifyLine("a=", "G.722", g722_line)
            msg:setSdp(sdp)
        end
    end
end

return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
m=audio 23588 RTP/AVP 9 0 8 18 101
a=rtpmap:9 G.722/8000
a=ptime:20
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:8 PCMA/8000
a=ptime:20
a=rtpmap:18 G729/8000
a=ptime:20
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Result

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
m=audio 23588 RTP/AVP 9 0 8 18 101
a=rtpmap:9 G722/8000
a=ptime:20
```

```
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:8 PCMA/8000
a=ptime:20
a=rtpmap:18 G729/8000
a=ptime:20
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

addline

```
sdp:addLine(new-line)
```

This method adds **new-line** at the end of SDP. It is recommended that the caller terminate the **new-line** with **\r\n** (or **\n**) to comply with SDP rules . No enforcement of line termination characters is done by this API.

Example: Append an attribute line to the end of the SDP.

Script

```
M = {}

function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()

    if sdp
    then
        sdp = sdp:addLine("a=some-attribute\r\n")

        msg:setSdp(sdp)
    end
end

return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24690 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Result

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.88
```

```
s=SIP Call
c=IN IP4 172.18.197.88
t=0 0
m=audio 24690 RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=some-attribute
```

insertLineAfter

```
sdp:insertLineAfter(start-of-line, line-contains, new-line)
```

This method finds the first line in sdp that starts with **start-of-line** and also has **line-contains** in that line, if **line-contains** parameter is specified . It will insert **new-line** after the found line.

Example: Insert a line into the SDP after the line with "a=" and "G729".

Script

```
M = {}

function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()

    if sdp
    then
        sdp = sdp:insertLineAfter("a=", "G729", "a=ptime:30\r\n")
        msg:setSdp(sdp)
    end
end

return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
m=audio 21702 RTP/AVP 18 101
a=rtpmap:18 G729/8000
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Result

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
```

```
t=0 0
m=audio 21702 RTP/AVP 18 101
a=rtpmap:18 G729/8000^M
a=ptime:30
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

insertLineBefore

```
sdp:insertLineBefore(start-of-line, line-contains, new-line)
```

This method finds the first line in sdp that starts with **start-of-line** and also has **line-contains** in that line, if **line-contains** parameter is specified . It will insert **new-line** before the found line.

Example: Insert a line into the SDP prior to the "s=" line.

Script

```
M = {}

function M.inbound_ANY_INVITE(msg)
    local sdp = msg:getSdp()

    if sdp
    then
        sdp= sdp:insertLineBefore("s=", nil, "e=nobody@cisco.com\r\n")
        msg:setSdp(sdp)
    end
end

return M
```

Message

```
SIP/2.0 200 OK
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
m=audio 17774 RTP/AVP 9 101
a=rtpmap:9 G722/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Result

```
SIP/2.0 200 OK
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
e=nobody@cisco.com
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
```

```
m=audio 17774 RTP/AVP 9 101
a=rtpmap:9 G722/8000
a=ptime:20
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

removeLine

```
sdp:removeLine(start-of-line, line-contains)
```

This method finds the first line in 'sdp' that starts with 'start-of-line' AND also has 'line-contains' in that line if 'line-contains' parameter is specified. The matching line is removed from the sdp.

Example: Remove the line containing both "a=rtpmap" and "G729" from the SDP.

Script

```
M = {}

function M.inbound_INVITE(msg)
    local sdp = msg:getSdp()

    if sdp
    then
        sdp = sdp:removeLine("a=rtpmap:", "G729")
        msg:setSdp(sdp)
    end
end

return M
```

Message

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
m=audio 25328 RTP/AVP 9 0 8 18 101
a=rtpmap:9 G722/8000
a=ptime:20
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:8 PCMA/8000
a=ptime:20
a=rtpmap:18 G729/8000
a=ptime:20
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Output/Result

```
INVITE sip:901@rawlings.cisco.com:5080 SIP/2.0
.
Content-Type: application/sdp

v=0
```

```
o=CiscoSystemsCCM-SIP 2000 1 IN IP4 172.18.197.22
s=SIP Call
c=IN IP4 172.18.197.29
t=0 0
m=audio 25328 RTP/AVP 9 0 8 18 101
a=rtpmap:9 G722/8000
a=ptime:20
a=rtpmap:0 PCMU/8000
a=ptime:20
a=rtpmap:8 PCMA/8000
a=ptime:20
a=ptime:20
a=fmtp:18 annexb=no
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Media Description Level APIs

The following APIs are used to get and manipulate SDP media levels:

SDP Example

In the following SDP example, the session level lined are shown in green while blue denotes media-level lines.

```
v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

In the SDP example provided above, the media-level indices are in blue color. Media level [1] is an audio media description. Media level [2] is a video media description (not shown in its entirety).

All indices are 1 based. Hence, the first media description is at index 1. The second is at index 2 and so on.

The following Medi Description level APIs are availabe:

- getMediaDescription(media-level)
- getMediaDescription(media-contains)
- modifyMediaDescription(media-level, media-description)
- modifyMediaDescription(media-contains, media-description)
- addMediaDescription(media-description)
- insertMediaDescription(media-level, media-description)
- removeMediaDescription(media-level)
- removeMediaDescription(media-contains)

Parameter Description

Parameter	Description
sdp	The sdp object is a string containing the SDP including all control characters (i.e. '**\r**' and '**\n**').
media-level	The media-level parameter is an index used to reference a media description within the SDP. The indices are 1 based. The first media description within the SDP is at media level 1. The second is at media level 2 and so on.
media-contains	The media-contains parameter is a string used to match something (e.g. the media type) within the m-line. For example, audio, video, and message can be used to get, modify, or remove the first occurrence of a media description where the m-line contains an exact match of the media-contains value.
media-description	The media-description parameter is a string contains a complete media description including multiple lines and the necessary embedded control characters: '\r' and '\n'. It is highly recommended that the last line be terminated with "\r\n". Although these APIs are purely text manipulations and no rules about the format of the text are enforced. In fact, some implementations don't actually include the '\r' at the end of each line. Treating this as pure text provides the necessary flexibility to handle either case and even convert easily between the two styles (e.g. it would be trivial to change "\r\n" to "\n" or vice versa). The following string is an example of a complete media description: `"m=audio 18884 RTP/AVP 9\r\na=rtpmap:9 G722/8000\r\n"`

The details of each media related API is described in the section below.

Note All indices (i.e. the media-level) are 1 based

getMediaDescription(media-level)

```
getMediaDescription(media-level) returns media-description
```

This API is used to get a particular media description at the specified media level.

Example: Set a local variable to the value of the first media description.

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local m1 = sdp:getMediaDescription(1)
    end
end
return M
```

Message

```
v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

Output/Result

```
Local variable m1 is set to a string

     m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
```

getMediaDescription(media-contains)

```
getMediaDescription(media-contains) returns media-description
```

This API is used to get a particular media description where media-contains is an exact match for some portion of the m-line.

Note If multiple m-lines match, only the first matching media-description is returned.

Example: Set a local variable to the value of the first media description whose m-line contains the text "audio".

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local audio = sdp:getMediaDescription("audio")
    end
end
return M
```

Message

```
v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
```

```
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

Output/Result

```
Local variable audio is set to a string

     m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
```

modifyMediaDescription(media-level, media-description)

```
modifyMediaDescription(media-level, media-description)
```

This API is used to modify a particular media description at the specified media level.

Example: Remove the lines containing "a=ptime:20\r\n" from the audio media description

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local m1 = sdp:getMediaDescription(1)
        if m1
        then
            m1 = m1:gsub("a=ptime:20\r\n", "")
            sdp = sdp:modifyMediaDescription(1, m1)
            msg:setSdp(sdp)
        end
    end
end
return M
```

Message Before Normalization

```
v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
```

```
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

Message After Normailzation

```
After normalization, the SDP would contain:

     v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=rtpmap:18 G729/8000\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

modifyMediaDescription(media-contains, media-description)

```
modifyMediaDescription(media-contains, media-description)
```

This API is used to modify a particular media description where media-contains is an exact match for some portion of the m-line. Note that if multiple m-lines match, only the first matching media-description is modified.

Example: Remove the lines containing "a=ptime:20\r\n" from the audio media description

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local audio = sdp:getMediaDescription("audio")
        if audio
        then
            audio = audio:gsub("a=ptime:20\r\n", "")
            sdp = sdp:modifyMediaDescription("audio", audio)
            msg:setSdp(sdp)
        end
    end
end
return M
```

Message Before Normalization

```
v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
```

```
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

Message After Normailzation

```
v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 172.18.195.96\r\n
     s=SIP Call\r\n
     c=IN IP4 172.18.195.126\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video…
```

addMediaDescription(media-description)

```
addMediaDescription(media-description)
```

This API is used to add a media description as the last media description.

Example:

The script removes and saves the video media description from inbound INVITEs. The script assumes the video media description is the 2nd media description. For the corresponding outbound responses, the script retrieves the stored video media description, sets the port to zero, and adds the media description into the outbound SDP.

Script

```
M = {}
function M.inbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local video = sdp:getMediaDescription(2)
        if video
        then
            --remove the video media description
            sdp = sdp:removeMediaDescription(2)
            --store video media description
            local context = msg:getContext()
            context["video"] = video
            msg:setSdp(sdp)
        end
    end
end

function M.outbound_ANY_INVITE(msg)
    local sdp = msg:getSdp()
    --assume other side is expecting video before audio when there is
    --a video m-line in the SDP
    if sdp
    then
```

```
        local context = msg:getContext()
        local video = context["video"]
        if video
        then
            --set port to zero in SDP answer
            video = video:gsub("video %d* RTP", "video 0 RTP")
            sdp = sdp:addMediaDescription(video)
            msg:setSdp(sdp)
        end
    end
end
return M
```

Message Before Normalization

```
INVITE SDP . . .

     v=0\r\n
     o=- 2000 3 IN IP4 10.10.10.100\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.100\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video 32068 RTP/AVP 112 113 114\r\n
     b=TIAS:4000000\r\n
     a=rtpmap:112 H264/90000\r\n
     a=fmtp:112
profile-level-id=4D8028;packetization-mode=1;max-mbps=243000;max-fs=8100\r\n
     a=rtpmap:113 H264/90000\r\n
     a=fmtp:113 profile-level-id=42400D;packetization-mode=1;max-mbps=11880;max-fs=396\r\n
     a=rtpmap:114 H264/90000\r\n
     a=fmtp:114 profile-level-id=42400D;packetization-mode=0;max-mbps=11880;max-fs=396\r\n
     a=sendrecv\r\n
     \r\n


     200 Ok SDP . . .

     v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 10.10.10.1\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.200\r\n
     t=0 0\r\n
[1]  m=audio 16007 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
     \r\n
```

Message After Normalization

```
INVITE SDP . . .

     v=0\r\n
     o=- 2000 3 IN IP4 10.10.10.100\r\n
     s=SIP Call\r\n
```

```
     c=IN IP4 10.10.10.100\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
     \r\n

     200 Ok SDP . . .

     v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 10.10.10.1\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.200\r\n
     t=0 0\r\n
[1]  m=audio 16007 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video 0 RTP/AVP 112 113 114\r\n
     b=TIAS:4000000\r\n
     a=rtpmap:112 H264/90000\r\n
     a=fmtp:112
profile-level-id=4D8028;packetization-mode=1;max-mbps=243000;max-fs=8100\r\n
     a=rtpmap:113 H264/90000\r\n
     a=fmtp:113 profile-level-id=42400D;packetization-mode=1;max-mbps=11880;max-fs=396\r\n
     a=rtpmap:114 H264/90000\r\n
     a=fmtp:114 profile-level-id=42400D;packetization-mode=0;max-mbps=11880;max-fs=396\r\n
     a=sendrecv\r\n
     \r\n
```

insertMediaDescription(media-level, media-description)

```
insertMediaDescription(media-level, media-description)
```

This API is used to add a media description as the at the specified media level.

Example: Inserting a message media description so that it is the first media description in the SDP

Script

```
M = {}
function M.outbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local message = "m=message 1234 sip:alice@10.10.10.100\r\n"
        sdp = sdp:insertMediaDescription(1, message)
        msg:setSdp(sdp)
    end
end
return M
```

Message Before Normalization

```
INVITE SDP . . .
```

```
     v=0\r\n
     o=- 2000 3 IN IP4 10.10.10.100\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.100\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
```

Message After Normalization

```
INVITE SDP . . .

     v=0\r\n
     o=- 2000 3 IN IP4 10.10.10.100\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.100\r\n
     t=0 0\r\n
[1]  m=message 1234 sip:alice@10.10.10.100\r\n
[2]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
```

removeMediaDescription(media-level)

```
removeMediaDescription(media-level)
```

This API is used to remove the media description at the specified media level.

Example:

Please refer to the example provided by addMediaDescription(media-description). It also uses removeMediaDescription.

removeMediaDescription(media-contains)

```
removeMediaDescription(media-contains)
```

This API is used to remove a particular media description where media-contains is an exact match for some portion of the m-line.

Note If multiple m-lines match, only the first matching media-description is remove.

Example:

The script removes and saves the video media description from inbound INVITEs. For the corresponding outbound responses, the script retrieves the stored video media description, sets the port to zero, and adds the media description into the outbound SDP.

Script

```
M = {}
function M.inbound_INVITE(msg)
    local sdp = msg:getSdp()
    if sdp
    then
        local video = sdp:getMediaDescription("video")
        if video
        then
            --remove the video media description
            sdp = sdp:removeMediaDescription("video")
            --store video media description
            local context = msg:getContext()
            context["video"] = video
            msg:setSdp(sdp)
        end
    end
end

function M.outbound_ANY_INVITE(msg)
    local sdp = msg:getSdp()
    --assume other side is expecting video before audio when there is
    --a video m-line in the SDP
    if sdp
    then
        local context = msg:getContext()
        local video = context["video"]
        if video
        then
            --set port to zero in SDP answer
            video = video:gsub("video %d* RTP", "video 0 RTP")
            sdp = sdp:addMediaDescription(video)
            msg:setSdp(sdp)
        end
    end
end
return M
```

Message Before Normalization

```
INVITE SDP . . .

     v=0\r\n
     o=- 2000 3 IN IP4 10.10.10.100\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.100\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video 32068 RTP/AVP 112 113 114\r\n
     b=TIAS:4000000\r\n
     a=rtpmap:112 H264/90000\r\n
     a=fmtp:112
profile-level-id=4D8028;packetization-mode=1;max-mbps=243000;max-fs=8100\r\n
     a=rtpmap:113 H264/90000\r\n
     a=fmtp:113 profile-level-id=42400D;packetization-mode=1;max-mbps=11880;max-fs=396\r\n
     a=rtpmap:114 H264/90000\r\n
```

```
     a=fmtp:114 profile-level-id=42400D;packetization-mode=0;max-mbps=11880;max-fs=396\r\n
     a=sendrecv\r\n
     \r\n

     200 Ok SDP . . .

     v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 10.10.10.1\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.200\r\n
     t=0 0\r\n
[1]  m=audio 16007 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
     \r\n
```

Message After Normalization

```
INVITE SDP . . .

     v=0\r\n
     o=- 2000 3 IN IP4 10.10.10.100\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.100\r\n
     t=0 0\r\n
[1]  m=audio 18884 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:18 G729/8000\r\n
     a=ptime:20\r\n
     a=fmtp:18 annexb=no\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
     \r\n

     200 Ok SDP . . .

     v=0\r\n
     o=CiscoSystemsCCM-SIP 2000 3 IN IP4 10.10.10.1\r\n
     s=SIP Call\r\n
     c=IN IP4 10.10.10.200\r\n
     t=0 0\r\n
[1]  m=audio 16007 RTP/AVP 9 18 101\r\n
     a=rtpmap:9 G722/8000\r\n
     a=ptime:20\r\n
     a=rtpmap:101 telephone-event/8000\r\n
     a=fmtp:101 0-1\r\n
[2]  m=video 0 RTP/AVP 112 113 114\r\n
     b=TIAS:4000000\r\n
     a=rtpmap:112 H264/90000\r\n
     a=fmtp:112
profile-level-id=4D8028;packetization-mode=1;max-mbps=243000;max-fs=8100\r\n
     a=rtpmap:113 H264/90000\r\n
     a=fmtp:113 profile-level-id=42400D;packetization-mode=1;max-mbps=11880;max-fs=396\r\n
     a=rtpmap:114 H264/90000\r\n
     a=fmtp:114 profile-level-id=42400D;packetization-mode=0;max-mbps=11880;max-fs=396\r\n
     a=sendrecv\r\n
     \r\n
```

CHAPTER 5

SIP Pass Through APIs

Cisco Unified CM is a Business to Business User Application (B2BUA) with respect to SIP call processing. The pass through object provides a set of APIs that allows information to be passed from one call leg to the other.

The following SIP Pass Through APIs are availabe:

- addHeader
- addHeaderValueParameter
- addHeaderUriParameter
- addRequestUriParameter
- addContentBody

addHeader

```
addHeader(header-name, header-value)
```

Given the string name of a header and value, this method adds the information to the pass through object for inclusion in the triggered outbound message.

Example

Without transparency, Cisco Unified CM ignores the inbound **x-nt-corr-id** header since it is unknown to it. Effectively, it is stripped off of the inbound INVITE and not included in the outbound INVITE.

To enable pass through of this header, it must be included in the **allowHeaders** table at the beginning of the script. Then it must be explicitly added to the pass through object during message handling.

Script

```
M = {}
M.allowHeaders = {"x-nt-corr-id"}
function M.inbound_INVITE(msg)
    local ntcorrid = msg:getHeader("x-nt-corr-id")
    if ntcorrid
    then
        pt = msg:getPassThrough()
        pt:addHeader("x-nt-corr-id", ntcorrid)
    end
end
return M
```

Inbound Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
x-nt-corr-id: 000002bf0f15140a0a@000075447daf-a561119
.
```

Outbound Message

```
INVITE sip:1234@10.10.10.2 SIP/2.0
.
x-nt-corr-id: 000002bf0f15140a0a@000075447daf-a561119
.
```

addHeaderValueParameter

```
addHeaderValueParameter(header-name, parameter-name [,parameter-value])
```

Given the name of a header , a parameter name, and a parameter value, this method adds the information to the pass through object for inclusion in the triggered outbound message.

The header name and parameter name are required arguments. The parameter-value is optional.

Note By default, **Contact** header value parameters, are passed through independent of any script logic, except the following, which are considered call-leg specific, and are Cisco Unified CM generated as appropriate:

- audio
- video

Example

Without transparency, Cisco Unified CM ignorea the inbound **x-tag** in the **From** header since it is unknown to it. Effectively, it is stripped off of the inbound INVITE and not included in the outbound INVITE.

To enable pass through of this header parameter, it must be explicitly added to the pass through object during message handling.

Script

```
M = {}
function M.inbound_INVITE(msg)
    local xtag = msg:getHeaderValueParameter("From", "x-tag")
    if xtag
    then
        pt = msg:getPassThrough()
        pt:addHeaderValueParameter("From", "x-tag", xtag)
    end
end
return M
```

Inbound Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
From: <sip:1000@10.10.10.58>;tag=0988bf47-df77-4cb4;x-tag=42
.
```

Outbound Message

```
INVITE sip:1234@10.10.10.2 SIP/2.0
.
From: <sip:1000@10.10.10.1>;tag=abcd;x-tag=42
.
```

addHeaderUriParameter

```
addHeaderUriParameter(header-name, parameter-name [,parameter-value])
```

Given the name of a header, a URI parameter name, and a parameter value, this method adds the information to the pass through object for inclusion in the triggered outbound message.

The header name and parameter name are required arguments. The parameter value is optional.

Example:

Without transparency, Cisco Unified CM ignores the inbound **cca-id** parameter in the Contact header URI since it is unknown to it. Effectively, it is stripped off of the inbound INVITE and not included in the outbound INVITE.

To enable pass through of this header URI parameter, it is explicitly added to the pass through object during message handling.

Note The parameter, in this example, takes on a different name in the outbound message (i.e. originating **cca-id** in the outbound message versus **cca-id** in the inbound message).

Script

```
M = {}
function M.inbound_INVITE(msg)
    local occaid = msg:getHeaderUriParameter("Contact", "cca-id")
    if occaid
    then
        pt = msg:getPassThrough()
        pt:addHeaderUriParameter("Contact", "originating-cca-id", occaid)
    end
end
return M
```

Inbound Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
Contact: <sip:1000@10.10.10.58;cca-id=LSC.dsn.mil>
.
```

Outbound Message

```
INVITE sip:1234@10.10.10.2 SIP/2.0
.
Contact: <sip:1000@10.10.10.1;originating-cca-id=LSC.dsn.mil>
.
```

addRequestUriParameter

```
addRequestUriParameter(parameter-name [,parameter-value])
```

Given a URI parameter name and a parameter value, this method adds the information to the pass through object for inclusion in the triggered outbound message.

The parameter name is a required argument. The parameter value is optional.

Note By default, Request-Uri parameters in an initial INVITE, are passed through independent of any script logic, except the following, which are considered call-leg specific, and are CUCM generated as appropriate:

- phone-context
- trunk-context
- tgrp
- user

Example: The inbound leg creates and passes through a parameter to be placed into the outbound Request URI.

Script

```
M = {}
function M.inbound_INVITE(msg)
    pt = msg:getPassThrough()
    pt:addRequestUriParameter("from-network", "service-provider")
end
return M
```

Inbound Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
```

Outbound Message

```
INVITE sip:1234@10.10.10.2;from-network=service-provider SIP/2.0
```

addContentBody

```
addContentBody(content-type, content-body [,content-disposition [,content-encoding
,[content-language]]])
```

Given a content-type, content-body, content-disposition, content-encoding, and content-language, this method adds the information to the pass through object for inclusion in the triggered outbound message.

The content-type and content-body are required arguments. The content-disposition, content-encoding, and content-language are optional parameters. If blank or nil is specified for any of these values, the header will not be included as part of the content.

Example:
Without transparency, Cisco Unified CM ignores the inbound INFO message and content body. Using transparency, Cisco Unified CM extracts the proprietary content body sent by a Nortel PBX, extract the DTMF digits from that content body, create a new dtmf-relay content body and pass that through.

Script

```
M = {}
function M.inbound_INFO(msg)
    local body = msg:getContentBody("application/vnd.nortelnetworks.digits")
    if body
    then
        local digits = string.match(body, "d=(%d+)")
        if digits
        then
            pt = msg:getPassThrough()
            body = string.format("Signal=%d\r\nDuration=100\r\n", digits)
            pt:addContentBody("application/dtmf-relay", body)
        end
    end
end
return M
```

Inbound Message

```
INFO sip: 1000@10.10.10.1 SIP/2.0
Via: SIP/2.0/UDP 10.10.10.57:5060
From: <sip:1234@10.10.10.57>;tag=d3f423d
To: <sip:1000@10.10.10.1>;tag=8942
Call-ID: 312352@10.10.10.57
CSeq: 5 INFO
Content-Type: application/vnd.nortelnetworks.digits
Content-Length: 72

p=Digit-Collection
y=Digits
s=success
u=12345678
i=87654321
d=4
```

Outbound Message

```
INFO sip: 1000@10.10.10.58 SIP/2.0
Via: SIP/2.0/UDP 10.10.10.1:5060
From: <sip:1234@10.10.10.1>;tag=ef45ad
To: <sip:1000@10.10.10.58>;tag=1234567
Call-ID: 475623@10.10.10.1
CSeq: 5 INFO
Content-Type: application/dtmf-relay
Content-Length: 26

Signal=4
Duration=100
```

CHAPTER 6

SIP Utility APIs

The Lua environment provides an APIs that allows data strings to be manipulated. The following SIP Utility API is availabe:

- parseUri

parseUri

```
parseUri(uri)
```

Given a URI string, this function parses the specified URI into a sipUri object and returns the sipUri object to the calling script. Null is returned if a valid URI is not specified.

Example:

Script

```
MM = {}
function M.inbound_INVITE(msg)
    local uriString = msg:getUri("P-Asserted-Identity")
    if uriString
    then
        local uri = sipUtils.parseUri(uriString)
    end
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
.
```

Output/Results

```
Local variable uriString is set to "sip:1234@10.10.10.1"
Local variable uri is a sipUri object containing the parsed form of uriString
    "uri:getUser() is "1234"
    "uri:getHost() is "10.10.10.1"
```

CHAPTER 7

SIP URI APIs

The Lua environment provides a set of APIs that allows a parsed SIP URI to be manipulated. The following SIP URI APIs are availabe:

- applyNumberMask
- getHost
- getUser
- encode

applyNumberMask

```
applyNumberMask(mask, mask-char)
```

Given a number mask, this function applies the specified number mask to the user part of the parsed URI and stores the modified user in the parse sipUri object. A string representation of the user part of the modified URI is returned.

Application of the number mask

The mask parameter defines the transformation to be applied to the user part of the URI.. The upper case "X" specifies the wildcard portion of the number mask. For example, if the mask "+1888XXXXXXX" and mask is applied to the example user part "4441234", the resulting sting is "+18884441234".

If the number of characters found in user part to be masked is less than the number of wildcard characters in the mask, the left most wildcard characters will left as "X". Applying the previous mask to the example user part "1234" yields the resulting string "+1888XXX1234". If the number of characters found in the user part to be masked is greater than the number of wildcard characters in the mask, the left most characters of the user part are truncated. For example, if the mask "+1888XXXX" is applied to the user part "4441234", the resulting string is "+18881234".

Example: Set a local variable to the value of the user part of URI in the P-Asserted-Identity after having applied a number mask to the user part of the URI

Script

```
M = {}
function M.inbound_INVITE(msg)
    local uriString = msg:getUri("P-Asserted-Identity")
    if uriString
    then
        local uri = sipUtils.parseUri(uriString)
        if uri
```

```
        then
            local user = uri:applyNumberMask("+1919476XXXX")
        end
    end
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
.
```

Output/Results

```
Local variable user is set to "+19194761234"
```

getHost

```
getHost()
```

This function retrieves the host portion of the parsed sipUri object and returns it to the caller as a string.

Example: Set a local variable to the host portion of the URI in the P-Asserted-Identity header.

Script

```
M = {}
function M.inbound_INVITE(msg)
    local uriString = msg:getUri("P-Asserted-Identity")
    if uriString
    then
        local uri = sipUtils.parseUri(uriString)
        if uri
        then
            local host = uri:getHost()
        end
    end
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
.
```

Output/Results

```
Local variable host is set to "10.10.10.1"
```

getUser

```
getUser()
```

This function retrieves the user portion of the parsed sipUri object and returns it to the caller as a string.

Example: Set a local variable to the user portion of the URI in the P-Asserted-Identity header

Script

```
M = {}
function M.inbound_INVITE(msg)
    local uriString = msg:getUri("P-Asserted-Identity")
    if uriString
    then
        local uri = sipUtils.parseUri(uriString)
        if uri
        then
            local user = uri:getUser()
        end
    end
end
return M
```

Message

```
IINVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
.
```

Output/Results

```
Local variable user is set to "1234"
```

encode

```
encode()
```

This function encodes the parsed sipUri object into a string and returns it to the caller. Any changes made to the parsed sipUri object prior to encoding will be reflected in the resulting string.

Example: Parse the URI from the P-Asserted-Identity header, apply a number mask, and then encode the resulting URI

Script

```
M = {}
function M.inbound_INVITE(msg)
    local uriString = msg:getUri("P-Asserted-Identity")
    if uriString
    then
        local uri = sipUtils.parseUri(uriString)
        if uri
        then
            uri:applyNumberMask("+1919476XXXX")
            uriString = uri:encode()
        end
    end
end
return M
```

Message

```
INVITE sip:1234@10.10.10.1 SIP/2.0
.
P-Asserted-Identity: <sip:1234@10.10.10.1>
```

Output/Results

```
Local variable uriString is set to "<sip:+19194761234@10.10.10.1>"
```

CHAPTER 8

Trace APIs

Script tracing allows the script writer to produce traces from within the script. This must only be used for debugging the script. It must be disabled when the script starts working. There are two ways to enable script tracing.

1. Via configuration—refer to the Script Trace checkbox associated with the SIP Trunk or SIP Profile.
2. Via scripting—refer to the **trace.enable()** API.

The following table indicates whether or not traces are produced by the script based on all combinations of the two settings described above:

Configuration	Scripting	Trace Produced
Enabled	Enabled	Yes
Enabled	Disabled	No
Disabled	Enabled	No
Disabled	Disabled	No

The trace library provides the following APIs:

- trace.format
- trace.enable
- trace.disable
- trace.enabled

trace.format

```
trace.format(format-string, p1, p2, ...)
```

Given a format string and a list of parameters, this function inserts the parameters into the format string and writes the output to the SDI trace file, if tracing is enabled for the trunk associated with the script and the script has enabled tracing via **trace.enable()**.

Example

```
M = {}
trace.enable()
function M.inbound_INVITE(msg)
    local callid = msg:getHeader("Call-Id")
    trace.format("Call-Id header is %s", callid)
```

```
end
return M
```

trace.enable

```
trace.enable()
```

This allows the script to enable tracing locally.

Note Tracing must also be enabled via configuration for the SIP Trunk or SIP Profile using the script.

Example

Refer to the trace.format example above.

trace.disable

```
trace.disable()
```

This allows the script to disable tracing locally.

Note If the script trace configuration flag for the associated SIP Trunk or SIP Profile is enabled, script trace will still be produced by Trace.output even if disabled here.

Example:

```
M = {}
trace.disable()
function M.inbound_INVITE(msg)
    local callid = msg:getHeader("Call-Id")
    trace.format("Call-Id header is %s", callid)
end
return M
```

trace.enabled

```
trace.enabled() turns a boolean
```

True is returned if script tracing has been enabled on the device associated with this instance of the script or if the script has enabled the tracing itself. If tracing has not been enabled, it returns false.

Example

```
M = {}
trace.enable()
function M.inbound_INVITE(msg)
    if trace.enabled()
    then
        local callid = msg:getHeader("Call-Id")
        trace.format("Call-Id header is %s", callid)
    end
end
return M
```

CHAPTER 9

Script Parameters API

Script parameters allows the script writer to obtain trunk specific or line specific configuration parameter values.

The scriptParameters library provides the following API:

- getValue

getValue

```
getValue(parameter-name)
```

Given a parameter name, this function returns the value of the parameter. If a parameter with the specified name exists and has a value then its returns the same value.

If the name exists but there is no associated value, then blank string is return (i.e. "" without the double quotes). If there is no such parameter with the specified name, nil is returned.

Example: The CCA-ID parameter is configured against the SIP trunk using this script for outbound INVITEs.

Script

```
M = {}
local ccaid = scriptParameters.getValue("CCA-ID")
function M.outbound_INVITE(msg)
    if ccaid
    then
        local contact = msg:getHeader("Contact")
        local replace = string.format("%s;CCA-ID=%s>", "%1", ccaid)
        contact = string.gsub(contact, "(<sip:.*)>", replace)
        msg:modifyHeader("Contact", contact)
    end
end
return M
```

Message Before Normallization

```
INVITE sip:1234@10.10.10.58 SIP/2.0
.
Contact: <sip:1000@10.10.10.1>
.
```

Message After Normallization

```
INVITE sip:1234@10.10.10.58 SIP/2.0
.
Contact: <sip:1000@10.10.10.1;CCA-ID=LSCSUB.dsn.mil>
```

CHAPTER 10

SIP Transparency

Cisco Unified Communications Manager (Unified CM) is a Back to Back User Agent (B2BUA). Therefore, any SIP to SIP call consists of 2 SIP dialogs. It is often useful to pass information from one dialog to the other during the life of the dialogs. This includes call setup, mid call, and end of call messaging. Using the pass through object described earlier, it is possible to trigger transparent pass through of information on from one SIP dialog (representing 1 of the call legs) to the other.

Currently, transparency is limited to INVITE dialogs on SIP trunks or SIP lines. SUBSCRIBE dialogs, PUBLISH, out-of-dialog REFER, out-of-dialog unsolicited NOTIFY are not supported, and MESSAGE are not supported.

Supported Features

The following messages support transparency:

- Initial INVITE and associated responses
  - INVITE response
  - 180 response
  - 183 response
  - 200 response
  - 4XX, 5XX, 6XX responses
- reINVITE and associated responses
- UPDATE message (transparency for responses to UPDATE is not supported)
- INFO message (transparency for responses to INFO is not supported)
- BYE message (transparency for response to BYE is not supported)

The following messages do not support Transparency:

- ACK
- PRACK and associated responses
- INVITE with replaces and associated responses
- REFER and associated responses

Typically Cisco Unified CM processes the following information (i.e. parameters, headers, and content bodies) locally. That is relative to a particular call leg. Hence, the SIP information that is understood is consumed and not passed across to the other call leg (which may not even be SIP anyway). This allows Cisco Unified CM to support various protocol inter-workings such as SIP to H.323, SIP to MGCP, etc. SIP information which is not understood by Cisco Unified CM is typically ignored.

In the following section, information which is understood and consumed by Cisco Unified CM is said to be **known** and the information not understood and consumed by Cisco Unifiecd CM is said to **unknown**.

The following information can be passed through transparently:

- Parameters
- Unknown headers
- Unknown content-bodies

Known headers

The following is the list of known headers:

- Accept
- Accept-Contact
- Accept-Resource-Priority
- Alert-Info
- Allow
- Allow-Events
- Also
- Authorization
- Bridge-Participant-ID
- Call-ID
- Call-Info
- CC-Diversion
- CC-Redirect
- Contact
- Content-Disposition
- Content-ID
- Content-Length
- Content-Type
- CSeq
- Date
- Diversion
- Event
- Expires
- From
- Geolocation

- Geolocation-Error
- Join
- Max-Forwards
- Min-Expires
- Min-SE
- MIME-Version
- P-Asserted-Identity
- P-Preferred-Identity
- Privacy
- Proxy-Authenticate
- Proxy-Authorization
- Proxy-Require
- RAck
- Reason
- Recv-Info
- Refer-To
- Referred-By
- Reject-Contact
- Remote-Party-ID
- Replaces
- Request-Disposition
- Requested-By
- Require
- Resource-Priority
- Retry-After
- RSeq
- RTP-RxStat
- RTP-TxStat
- Server
- Session
- Session-Expires
- SIP-ETag
- SIP-If-Match
- Subject
- Subscription-State
- Supported
- Target-Dialog
- To

- Unsupported
- User-Agent
- Via
- Warning
- WWW-Authenticate
- X-Cisco-EMCCInfo
- X-Cisco-FallbackID
- X-Cisco-ViPR-Ticket

Known Content-bodies

- application/sdp

If the script attempts to pass through a known header or content-body, it will trigger an execution error.

A script writer will quickly figure out that there is a way to pass known data through without it being consumed by or interfering with Cisco Unified CM processing. Effectively, the script can get the value for a known header and place into an unknown header. The same can be done with content bodies. The 181 Transparency example below does just that with the Reason header. It gets the Reason header value and passes it through as an X-Reason header. Of course, if there is no script on the other side to consume the X-Reason header and remove it, the header will be sent to the network.

In any case, known header or content-body transparency is not supported, and SIP Transparency and Normalization is not intended for this purpose. Customers who use SIP transparency to pass through known headers or content-bodies are also responsible for the results.

In particular, SDP transparency is not supported. Cisco Unified CM cannot apply region bandwidth policies or call admission control in this case, nor can it insert media resources, which may be necessary. Many call flows result in SDP updates, which Cisco Unified CM would be unable to perform correctly. While it may be possible in some cases to manipulate declared elements in SDP successfully, manipulating negotiated elements is unlikely to succeed beyond initial call setup.

Example for 181 Transparency

Without transparency, if Cisco Unified CM receives a 181 on the outbound trunk leg, Cisco Unified CM's native behavior is to send a 180 back on the inbound trunk leg. To achieve 181 transparency, a script is required for both the inbound 181 (received on the B side) and for the would-be outbound 180 (sent on the A side).

Since the 181 is received from the PBX-B first, consider doing the following first:

- Get the Reason header value
- Pass through the Reason header value— Since the Reason header is a known header, the script will bypass the known header check by passing through the value using the header name X-Reason.

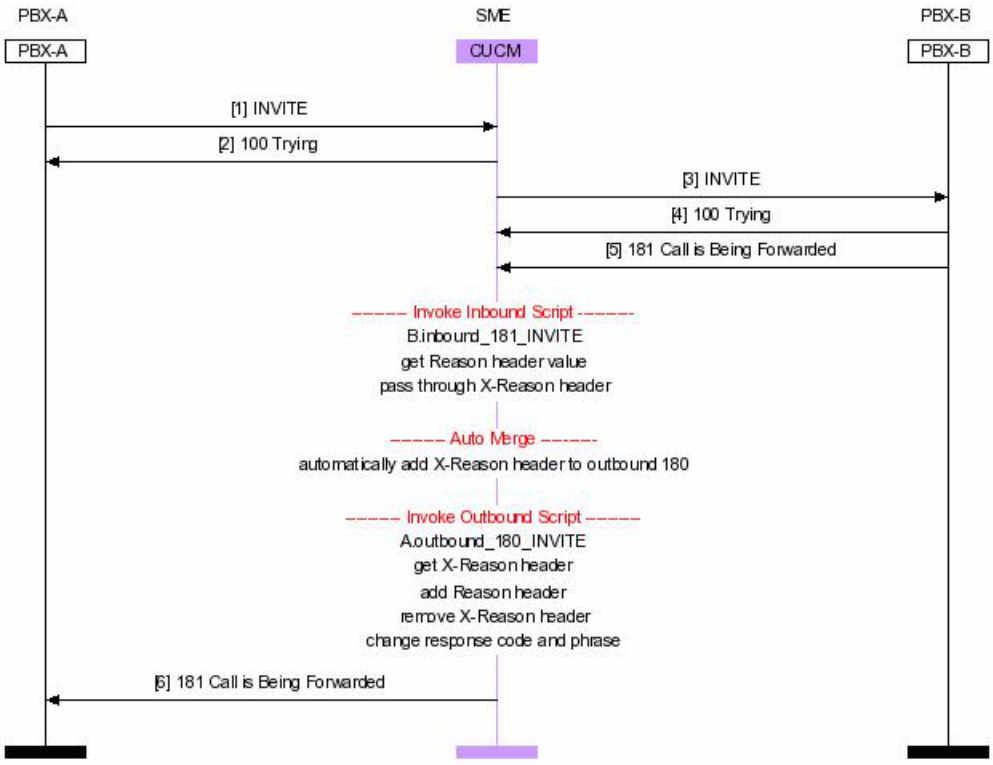
Cisco Unified CM will automatically merge the pass through data with the outbound message it would have sent. As mentioned previously, it would natively send a 180. The auto merge functionality therefore, places the X-Reason header into an outbound 180.

Next, one must consider what the A side needs to do:

- Get the X-Reason header value and see if contains something about 181
- Add a Reason header with the X-Reason header value

- Remove the X-Reason header
- Convert the response code and phrase to 181 Call is Being Forwarded.

These steps are depicted in the following callflow diagram:



The B-side and A-side scripts are shown below:

B-Side Script

```
B = {}
function B.inbound_181_INVITE(msg)
    local pt = msg:getPassThrough()
    local reason = msg:getHeader("Reason")
    if pt and reason
    then
        pt:addHeader("X-Reason", reason)
    end
end
return B
```

A-Side Script

```
A = {}
function A.outbound_180_INVITE(msg)
    local reason = msg:getHeader("X-Reason")
    if reason
    then
        if string.find(reason, "cause=181")
        then
            msg:setResponseCode(181,"Call is being forwarded")
            msg:addHeader("Reason", reason)
        end
```

```
        msg:removeHeader("X-Reason")
    end
end
return A
```

Example for INFO Transparrency

Without transparency, Cisco Unified CM ignores the inbound INFO message and content body. Using transparency, Cisco Unified CM extracts the proprietary content body sent by a Nortel PBX, extract the DTMF digits from that content body, create a new dtmf-relay content body and pass that through to the other call leg.

Script

```
M = {}
function M.inbound_INFO(msg)
    local body = msg:getContentBody("application/vnd.nortelnetworks.digits")
    if body
    then
        local digits = string.match(body, "d=(%d+)")
        if digits
           then
            pt = msg:getPassThrough()
            body = string.format("Signal=%d\r\nDuration=100\r\n", digits)
            pt:addContentBody("application/dtmf-relay", body)
        end
    end
end
return M
```

Inbound Message

```
INFO sip: 1000@10.10.10.1 SIP/2.0
Via: SIP/2.0/UDP 10.10.10.57:5060
From: <sip:1234@10.10.10.57>;tag=d3f423d
To: <sip:1000@10.10.10.1>;tag=8942
Call-ID: 312352@10.10.10.57
CSeq: 5 INFO
Content-Type: application/vnd.nortelnetworks.digits
Content-Length: 72

p=Digit-Collection
y=Digits
s=success
u=12345678
i=87654321
d=4
```

Outbound Message

```
INFO sip: 1000@10.10.10.58 SIP/2.0
Via: SIP/2.0/UDP 10.10.10.1:5060
From: <sip:1234@10.10.10.1>;tag=ef45ad
To: <sip:1000@10.10.10.58>;tag=1234567
Call-ID: 475623@10.10.10.1
CSeq: 5 INFO
Content-Type: application/dtmf-relay
Content-Length: 26
```

```
Signal=4
Duration=100
```

Script parameters for PCV and PAI Headers

For P-Charging Vector

P-Charging Vector header script parameter is enhanced to add transparency to pass through mobile related information.

Prior to release 8.6(1), if Cisco Unified Communications Manager received a call that had P-Charging Vector, Cisco Unified Communications Manager sent call information on the inbound trunk leg without any mobile or IP Multimedia Subsystem (IMS) values to its service providers. Now, using the modified PCV script parameter for better transparency, Cisco Unified Communications Manager extracts the charging information that is sent from a mobile or PSTN (received on the B side), and passes that information through to the other call leg without modification (sent on the A side).

Script Parameter

term-ioi—Configure this parameter if the script needs to add the term-ioi parameter to the P-Charging-Vector header in any of the outgoing 200 OK originating from Call Manager.

For P-Asserted Identity

P-Asserted Identity header script parameter is enhanced so that Cisco Unified Communications Manager can pass through any of the PAI information unmodified, present in the incoming calls to the outgoing calls, that originate from Cisco Unified Communications Manager.

Script Parameter

pai-passthru—Configure this parameter if the script needs to pass through the P-Asserted-Identity in the outgoing calls.

Diversion Counter

The diversion counter script handles the diversion counter parameter for call forward scenarios. The diversion counter script provides the following functionality:

- Acts as a pass-through diversion counter parameter for a basic tandem (trunk-to-trunk) call with *no* diversions within the Unified CM cluster
- Handles the diversion counter parameter for a tandem call with one or more diversions within the Unified CM cluster

- Handles the diversion counter parameter for a tandem call when the inbound INVITE message has more than two Diversion headers and there is also one or more diversions within the Unified CM cluster.

Exceptions

The diversion counter script has the following exceptions:

- The diversion counter script is applicable only for tandem calls.
- If there are multiple diversions within the cluster, the diversion counter parameter is increased by just one.

Script

```
--[[
    Description:
        Diversion Counter Handling For Call Forward Scenarios. This script should be
attached to both inbound and outbound trunk. This script provides the following
functionalists:

       1. Pass through counter parameter in case Basic Tandem call with NO Diversion
within the cluster.
       2. Handling of counter parameter for Tandem call with one or more Diversion within
CUCM cluster
       3. Handling of counter parameter for Tandem call when inbound INVITE has more than
two Diversion headers and there is also one or more Diversion within CUCM cluster

    Exceptions:
       1. This script is only applicable for Tandem (trunk-to-trunk) calls.
       2. If there are multiple diversions within the cluster, the Diversion counter
parameter will be increase just by 1.
--]]

M = {}
function M.inbound_INVITE(msg)
    if msg:isReInviteRequest()
    then
        return
    end

    -- Get the Diversion header. If no Diversion header then return.
    local diversion = msg:getHeader("Diversion")
    if not diversion
    then
       return
    end
    local pt = msg:getPassThrough()
    pt:addHeader("X-Diversion", diversion)
end
function M.outbound_INVITE(msg)
    -- This script is applicable only for initial INVITE.
        if msg:isReInviteRequest()
        then
                return
        end
    -- Get Diversion header. If there is no Diversion header then return
        local diversion = msg:getHeader("Diversion")
        if not diversion
        then
                return
        end

        -- Get X-Diversion Header. If there was no X-Diversion header then return
```

```
    local xDiversion = msg:getHeader("X-Diversion")
    if not xDiversion
    then
            return
    end

-- Get URI from Diversion header
    local divString = msg:getUri("Diversion")
    if divString
    then

    -- Parse URI and get DN and Host
            local divuri = sipUtils.parseUri(divString)
            if divuri
            then
                    lrn_user = divuri:getUser()
                    lrn_host = divuri:getHost()

           end
    end

-- Get URI from X-Diversion header
    local xdivString = msg:getUri("X-Diversion")
    if xdivString
    then
       -- Parse URI and get DN and Host
            local xdivuri = sipUtils.parseUri(xdivString)
            if xdivuri
            then
                    xlrn_user = xdivuri:getUser()
                    xlrn_host = xdivuri:getHost()
            end
    end

-- Get counter parameter value from inbound diversion (X-Diversion).
local counter = msg:getHeaderValueParameter("X-Diversion", "counter")

-- If new LRN is different from incoming LRN, that means there was a local
    -- call forward on UCM, so increase the counter.
    if not ( string.match(lrn_user, xlrn_user) and string.match(lrn_host, xlrn_host) )
    then

    -- If there is no counter parameter, then find out how many Diversion
    -- headers are there. Set the counter to no. of Diversion Header.
    -- If there is a counter value, just increase it by 1.
            if not counter
            then

                    local xdiv = msg:getHeaderValues("X-Diversion")
                    if #xdiv > 1
                     then
                             counter = #xdiv
                     end
           else
                    counter = counter + 1
            end
            msg:addHeaderValueParameter("Diversion","counter", counter)
    else
            -- As there is no local call forwarding, so pass through the counter
            -- parameter.
            if counter
            then
       msg:addHeaderValueParameter("Diversion","counter", counter)
       end
```

```
        end
        msg:removeHeader("X-Diversion")
end
return M
```

Message Transformation

If there is call forwarding within the cluster and an incoming INVITE message has a Diversion header with a counter parameter, the diversion counter script increases the counter parameter by one in the outbound INVITE message.

Original Inbound Message

```
INVITE sip:3002@10.10.10.53:5060 SIP/2.0^M
Via: SIP/2.0/TCP 10.10.10.51:5060;branch=z9hG4bK4b8a7eea6b41^M
From: <sip:1003@10.10.10.51>;tag=130169~d434b1d7-c1e3-44e7-a77e-abf211d2682e-26620367^M
To: <sip:3002@10.10.10.53>^M
Date: Fri, 04 Nov 2011 14:51:39 GMT^M
Call-ID: 7ef45500-eb31fbfb-3ec2-330a0a0a@10.10.10.51^M
Supported: timer,resource-priority,replaces^M
Min-SE:  1800^M
User-Agent: Cisco-CUCM8.6^M
Allow: INVITE, OPTIONS, INFO, BYE, CANCEL, ACK, PRACK, UPDATE, REFER, SUBSCRIBE, NOTIFY^M
CSeq: 101 INVITE^M
Expires: 180^M
Allow-Events: presence, kpml^M
Supported: X-cisco-srtp-fallback^M
Supported: Geolocation^M
Call-Info: <sip:10.10.10.51:5060>;method="NOTIFY;Event=telephone-event;Duration=500"^M
Cisco-Guid: 2129941760-0000065536-0000000148-0856295946^M
Session-Expires:  1800^M
Diversion: <sip:1001@10.10.10.51>;reason=no-
answer;privacy=off;screen=yes;counter=2,<sip:1006@10.10.10.51>;reason=no-
answer;privacy=off;screen=yes^M
P-Asserted-Identity: <sip:1003@10.10.10.51>^M
Remote-Party-ID: <sip:1003@10.10.10.51>;party=calling;screen=yes;privacy=off^M
Contact: <sip:1003@10.10.10.51:5060;transport=tcp>^M
Max-Forwards: 69^M
Content-Length: 0^
```

Outbound Message

Changes to the outbound message after the diversion counter script runs are in bold.

```
IINVITE sip:1005@10.10.10.51:5060 SIP/2.0^M
Via: SIP/2.0/TCP 10.10.10.53:5060;branch=z9hG4bK1d64062fa6f^M
From: <sip:1003@10.10.10.53>;tag=18448~94147210-61b5-4d32-a08d-5daf91ec321b-27003595^M
To: <sip:1005@10.10.10.51>^M
Date: Fri, 04 Nov 2011 14:51:44 GMT^M
Call-ID: 81ef4580-eb31fc00-e2-350a0a0a@10.10.10.53^M
Supported: timer,resource-priority,replaces^M
Min-SE:  1800^M
User-Agent: Cisco-CUCM8.6^M
Allow: INVITE, OPTIONS, INFO, BYE, CANCEL, ACK, PRACK, UPDATE, REFER, SUBSCRIBE, NOTIFY^M
CSeq: 101 INVITE^M
Expires: 180^M
Allow-Events: presence, kpml^M
Supported: X-cisco-srtp-fallback^M
Supported: Geolocation^M
Call-Info: <sip:10.10.10.53:5060>;method="NOTIFY;Event=telephone-event;Duration=500"^M
Cisco-Guid: 2179941760-0000065536-0000000121-0889850378^M
Session-Expires:  1800^M
Diversion: <sip:3002@10.10.10.53>;reason=no-
```

```
answer;privacy=off;screen=yes;counter=3,<sip:1006@10.10.10.51>;reason=no-
answer;privacy=off;screen=yes^M
P-Asserted-Identity: <sip:1003@10.10.10.53>^M
Remote-Party-ID: <sip:1003@10.10.10.53>;party=calling;screen=yes;privacy=off^M
Contact: <sip:1003@10.10.10.53:5060;transport=tcp>^M
Max-Forwards: 68^M
Content-Length: 0^M
```

CHAPTER 11

Preloaded Scripts

Preloaded scripts are Lua scripts that are provided as part of a Cisco Unified Communications Manager (Unified CM) basic installation. The preloaded scripts are automatically available when you upgrade to (or install) Unified CM Release 8.6 or a later release. These scripts normally have transparency- and normalization-related Lua code that you can use for a particular feature and are named with an easy-to-follow naming convention.

The administrator can select the preloaded scripts on the SIP Trunk Normalization Script drop-down menu. To add additional transparency- or normalization-related changes to a preloaded script, copy the content of the preloaded script and create a new script with the copied content plus additional desired changes.

The following preloaded scripts are provided with Unified CM Release 9.0 and later releases:

- Refer-passthrough—Used to pass through an inbound in-dialog REFER (without Replaces) message to the other side of a call arc, if it is a SIP trunk.The Lua script will copy the Refer-To and Referred-By headers from the inbound REFER message and save them in the transparency object.
- HCS-PCV-PAI-passthrough—This script is used to:
  - Pass through a P-Charging-Vector header for INVITE, UPDATE, and 200 OK
  - Use the configured term-ioi while adding the P-Charging-Vector to 200 OK
  - Pass through a P-Asserted-Identity header for INVITE
- vcs-interop—Used to allow proper interoperation between Unified CM and VCS. This script specifically handles the differences in how the two nodes support SRTP and will change the right side of URIs in the From, Remote-Party-Id and P-Asserted-Id headers to use the configured top-level domain instead of the IP address of Unified CM.
- diversion counter—Used to handle the diversion counter parameter for call forward and diversion scenarios. This script should be attached to both the incoming and outgoing trunks. If there is a call diversion or call forward within the cluster, then this script will adjust the counter parameter in the outgoing diversion header. If there is no call forwarding within the cluster, then this script will simply pass through the counter parameter from the inbound to the outbound side. For more information about the diversion counter script, see Diversion Counter, page 10-7.