Cisco TelePresence Supervisor API guide 2.8

Product Programming Reference Guide

D14665.05

February 2011

Contents

Introduction

This document accompanies the latest version of the remote management API for the Cisco TelePresence Supervisor MSE 8050 software (respectively referred to as API and Supervisor in this document). The following Cisco TelePresence products support this API when they are running Supervisor version 2.2 and later:

- Cisco TelePresence Supervisor MSE 8050

API History

The following table shows the device's software versions and the corresponding supported API versions:

API version	Supervisor version
2.8 (this version)	2.2 and later

XML-RPC implementation

API calls and responses are implemented using the XML-RPC protocol. This simple protocol does remote procedure calling using HTTP as the transport and XML as the encoding. It is extremely simple although it does still allow for complex data structures. XML-RPC is not platform-dependent and was chosen in favor of SOAP (Simple Object Access Protocol) because of its simplicity.

The interface is stateless, which means the application must either regularly poll the device for status or continually listen to the device - if it is configured to publish feedback events.

The API implements all parameters and returned data as **`<struct>`** elements, each of which is explicitly named. For example, **`device.query`** returns (amongst other data) the current time as:

```
<member>
       <name>currentTime</name>
       <value><dateTime.iso8601>20110121T13:31:26<dateTime.iso8601></value>

</member>
```

rather than simply

```
<dateTime.iso8601>20110121T13:31:26<dateTime.iso8601>
```

Note: Unless otherwise stated, assume strings have a maximum length of 32 characters.

Refer to the XML-RPC specification[1] for more information.

Transport protocol

The device implements HTTP/1.1 as defined by RFC 2616[2]. It expects to receive HTTP communications over TCP/IP connections to port 80. The application should send HTTP POST messages to the URL **`/RPC2`** on the device's IP address.

HTTPS is provided on TCP port 443 by default, although you can configure the device to receive HTTP and HTTPS on non-standard TCP port numbers if necessary.

API overhead

Every API command that your application sends incurs a processing overhead within the target device's own application. The amount of overhead varies with the type of command and parameters. A high API processing load may impair the device's performance – in the same way as if several users simultaneously accessed the device's web interface. Bear this in mind when you design your application's architecture and software.

For this reason, we recommend that you use a single server to run the calling application and send commands to the device. If users need concurrently use the application, provide a web interface on your application server or write a client to communicate with that server. The server then manages the client requests and only the server sends API commands directly to the device.

We also recommend that you implement some control in your application to prevent the device being overloaded with API commands.

These design considerations provide more control than allowing clients to send API commands directly and will prevent the device's performance from being impaired by unmanageable API load.

API overview

Encoding

Your application can encode messages as ASCII text or as UTF-8 Unicode. If you do not specify the encoding, the API assumes ASCII encoding. You can specify the encoding in a number of ways:

Specify encoding with HTTP headers

There are two ways of specifying UTF-8 in the HTTP headers:

- Use the **`Accept-Encoding: utf-8`** header
- Modify the **`Content-Type`** header to read **`Content-Type: text/xml; charset=utf-8`**

Specify encoding with XML header

The **`<?xml>`** tag is required at the top of each XML file. The API will accept an encoding attribute for this tag; that is, **`<?xml version="1.0" encoding="UTF-8"?>`**.

Authentication

The controlling application must authenticate itself on the device as a user with administrative privileges. Also, because the interface is stateless, every call must contain authentication parameters:

`authenticationUser`

Type: **string**

Name of a user with sufficient privilege for the operation being performed. The name is case sensitive.

`authenticationPassword`

Type: **string**

The password that corresponds with the given **`authenticationUser`**. The API ignores this parameter if the user has no password. This behavior differs from the web interface, where a blank password must be blank.

Note: Authentication information is sent using plain text and should only be sent over a trusted network.

Message flow

The application initiates the communication and sends a correctly formatted XML-RPC command to the device.

Example command

```
<?xml version='1.0' encoding='UTF-8'?>
  <methodCall>
    <methodName>recording.delete</methodName>
```

```
      <params>
        <param>
          <value>
            <struct>
              <member>
                <name>authenticationPassword</name>
                <value><string></string></value>
              </member>
              <member>
                <name>recordingId</name>
                <value><int>101</int></value>
              </member>
              <member>
                <name>authenticationUser</name>
                <value><string>admin</string></value>
              </member>
            </struct>
          </value>
        </param>
      </params>
  </methodCall>
```

Assuming the command was well formed, and that the device is responsive, the device will respond in one of these ways:

- With an XML **`methodResponse`** message that may or may not contain data, depending on the command.
- With an XML **`methodResponse`** that includes only a fault code message.

Example success

```
<?xml version="1.0"?>
  <methodResponse>
    <params>
      <param>
        <value>
          <struct>
            <member>
              <name>status</name>
              <value>
                <string>operation successful</string>
              </value>
            </member>
          </struct>
        </value>
      </param>
    </params>
  </methodResponse>
```

Example fault code

```
<?xml version="1.0"?>
```

```
<methodResponse>
 <fault>
   <value>
     <struct>
       <member>
         <name>faultCode</name>
           <value>
             <int>22</int>
           </value>
       </member>
       <member>
         <name>faultString</name>
           <value>
             <string>no such recording</string>
           </value>
       </member>
     </struct>
   </value>
 </fault>
</methodResponse>
```

Enumerate methods

Enumerate methods have the potential to return a large volume of data, so these calls have a control mechanism to limit the number of enumerated items per call.

Each enumerate call may take and return an **`enumerateID`** parameter which tells the API or calling application where to start the enumeration. The mechanism works as follows:

1. The application calls an enumerate method without an **`enumerateID`** parameter.
2. The device returns an array containing the enumerated items, and possibly an enumerateID. The response will always include an enumerateID if the device enumerated more items than it included in the response.
3. If there is an enumerateID, the application should call the enumerate method again, supplying the enumerateID as returned by the previous call.
4. The application should repeat this process until the response fails to include an enumerateID. This means that the enumeration is complete.

Note: Do not supply your own enumerateID values; make sure you only use the values returned by the device.

Enumerate filters

Enumerate methods will accept an optional **`enumerateFilter`** parameter, which allows you to filter the response. The parameter must contain a filter expression, which is built from criteria and operators.

The filter criteria that a call will accept vary depending on the call, but the syntax for using those criteria in expressions is the same for all methods that allow filtering. The reference information for methods that allow filtering includes acceptable filter criteria.

If the filter expression evaluates to true for the enumerated item, the item will be included in the device's response. If the expression evaluates false, the enumerated item will be filtered out of the response.

Filter expressions consist of atomic expressions combined with operators and parentheses. Whitespace is ignored. Functions are valid, and any parameters are in a comma separated list in parentheses after the function name, for example, **`function(expression1,expression2)`**.

For example, if the expression **`(inProgress && internal)`** is used to filter the response to **`recording.enumerate`**, the returned array of recordings will only include those which are both **`inProgress`** and **`internal`**.

The integer 0 evaluates to false and all other integers to true. Integers can be expressed using any string of valid digits. Prefix hex digits with **`0x`**, decimal with **`0t`** and binary with **`0z`**. The API assumes decimal if you don't supply a prefix.

Binary operators

The following binary operators are valid, in order of priority (lowest priority first):

Operator	Description
||	Boolean or
&&	Boolean and
|	Bitwise or
^	Bitwise exclusive or
&	Bitwise and
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>=	Greater than or equal
>	Greater than
<<	Bitwise left shift
>>	Bitwise right shift
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Unary operators

The following unary operators are valid. All of these bind tighter than any binary operator.

Operator	Description
-	Unary minus

+	Unary plus
!	Logical negation
~	Bitwise negation

API reference

This is a list of the API calls supported by the target device. For each API call, the following information is provided where applicable:

- Description of the call's function and status
- Accepted parameters
- Returned parameters, structure formats and data types
- Deprecated parameters

Click the call name to read a detailed description of the call.

auditlog.delete

Status: **active**

Deletes entries from the device's audit log.

Accepts:

`deleteIndex`

Type: **integer**

You can delete logs in segments of 400 entries.To delete logs, enter the value returned by the deleteableIndex response (see auditlog.query below). This will delete all complete 400 log segments below this amount, leaving the residuals. Alternatively, you can delete less than this amount by picking a number below the value of deleteableIndex. This will delete all complete 400 logs segments below that number, leaving any residuals. Stored audit events up to and including the indicated deleteIndex will be permanently deleted.

auditlog.query

Status: **active**

Queries the device for statistics about the audit log.

Returns:

`firstIndex`

Type: **integer**

The index of the oldest stored audit event.

`deletableIndex`

Type: **integer**

The index of the most recent deletable audit event.

`numEvents`

Type: **integer**

The total number of events stored.

`percentageCapacity`

Type: **integer**

The percentage of the total available capacity being used by the audit log.

chassis.alarms.clear

Status: **active**

Clears historic alarms. Does not accept parameters (other than authentication) and does not return any data.

chassis.alarms.query

Status: **active**

Queries the device for data on chassis alarms.

Returns:

`alarms`

Type: **array**

An array whose members represent the alarms from the chassis.

`alarms` array members

`alarmDescription`

Type: **string**

A verbose description of the alarm, for example, **`No CompactFlash card is inserted in the Supervisor blade`**.

`alarmLevel`

Type: **string**

The relative importance of the alarm.

Value	Description
None	
Minor	
Major	
Critical	

alarmName

Type: **string**

Identifies the alarm, for example, **noCompactFlashCardInstalled**.

alarmState

Type: **string**

The current alarm state.

Value	Description
OK	There is no alarm condition.
Alarm	The device is experiencing an alarm condition and the alarm is active.
Muted	The alarm condition is present but the alarm is suppressed.
HistoricAlarm	A previous alarm condition is no longer present.

alarmTitle

Type: **string**

A verbose string that identifies alarms that have titles. For example, the alarm named **noCompactFlashCardInstalled** has the title **No compact flash card inserted**.

chassis.blades.query

Status: **active**

Queries the Supervisor for data about the blades installed in the chassis. If the blade has more than one active Ethernet port, the query returns address information about each port; this topic shows only the responses for portA.

Returns:

blades

Type: **array**

An array whose members represent the blades installed in the chassis.

`blades` array members

`slot`

Type: **integer**

The number of the chassis slot in which the blade is installed.

Value	Description
1	The Cisco TelePresence Supervisor MSE 8050 is always in slot 1.
2	
10	The highest slot number in the Cisco TelePresence MSE 8000 chassis.

`type`

Type: **string**

The type of blade. If there is no blade in the queried slot, `type` is an empty string.

`softwareVersion`

Type: **string**

The version number of the blade's software.

`status`

Type: **string**

The current status of the blade, which consists of a primary status string that may be concatenated with additional status string(s). The blade will have one of the following primary status strings:

Value	Description
Blade absent	
Blade OK	The device only reports this status if none of the other status conditions apply.
Waiting for communications	
Lost communication	
Blade software too old	
Please upgrade	
Blade shut down	
Blade shutting down	
Blade attempting restart	
Blade restarting	

If the chassis has additional status information for the blade, one or more of the following additional strings are concatenated with the primary string, separated by semi-colons. For example, if the blade is shut down at the time of the query, the reported status might be **`Blade shut down; Requires restart; Temperature critical`**.

Secondary status strings:

Value	Description
Temperature critical	
Voltages critical	
Requires restart	
Blade badly inserted	
Invalid blade ID	
RTC Battery critical	

`portA`

Type: **string**

The IPv4 address of the blade's Ethernet port A, in dotted quad format.

`portAIpv6Address`

Type: **string(63)**

The IPv6 address of this Ethernet port, in CIDR format.

`portALinkLocalIpv6Address`

Type: **string(63)**

The link local IPv6 address of this Ethernet port, in CIDR format.

chassis.fantrays.query

Status: **active**

Queries the Supervisor for information on the chassis's fan tray.

Returns:

`fanTrays`

Type: **array**

An array whose members represent the fan trays in the chassis.

`fanTrays` array members

`tray`

Type: **string**

Identifies the fan tray.

Value	Description
Upper	The adjacent information is about the chassis's upper fan tray.
Lower	The adjacent information is about the chassis's lower fan tray.

status

Type: **string**

The status of the fan tray:

Value	Description
OK	
Problem	
Initializing	
Fan tray absent	

type

Type: **string**

The model name of the fan tray. The query only returns this data if the status is either `OK` or `Problem`.

serial

Type: **string**

The serial number of the fan tray. The query only returns this data if the status is either `OK` or `Problem`.

averageFanSpeed

Type: **integer**

The average speed of the fan in RPM. The query only returns this data if status is either `OK` or `Problem`.

chassis.powershelf.query

Status: **active**

Queries the Supervisor for information about the chassis's powershelves.

Returns:

powerShelves

Type: **array**

An array whose members represent the powershelves in the chassis.

`powerShelves` array members

`shelf`

Type: **string**

This value, either `A` or `B`, identifies the power shelf to which the adjacent information applies.

`status`

Type: **string**

The status of the power shelf:

Value	Description
OK	
Power shelf reporting fault	
Power shelf not monitored	
Lost contact with power shelf	
Insufficient current capacity	
Authentication failed	

`type`

Type: **string**

The model of the power shelf. The query only returns this data when Valere monitoring is enabled (via the web interface) and the shelf status is not `Authentication failed`.

`reportedInfo`

Type: **array**

An array whose members represent each rectifier in the powershelf.

`reportedInfo` array members

`rectifier`

Type: **integer**

This value between `1` and `4` identifies the rectifier to which the adjacent information applies.

`status`

Type: **string**

The status of this rectifier, as reported directly by the rectifier.For example, `0807-OK\r`.

`voltage`

Type: **integer**

The output voltage of this rectifier in mV.

`current`

Type: **integer**

The current being supplied by this rectifier at the time of the query, in mA.

`capacity`

Type: **integer**

The current capacity of this rectifier in mA.

chassis.supplyvoltage.query

Status: **active**

Queries the Supervisor for information on the chassis's supply voltage.

Returns:

`powerSupplies`

Type: **array**

An array whose members represent the chassis's power supplies.

`powerSupplies` array members

`supply`

Type: **string**

This value, either `A` or `B`, identifies the power supply to which the adjacent information applies.

`status`

Type: **string**

The status of the power supply voltage:

Value	Description
OK	
Out of range high	
Out of range low	
Too high	
Too low	
Supply not monitored	

`voltage`

Type: **integer**

The supply's output voltage in mV. The query does not return this data if status is **`Out of range low`** or **`Supply not monitored`**.

device.health.query

Status: **active**

Returns the current status of the device, such as health monitors and CPU load.

Returns:

`cpuLoad`

Type: **integer**

The CPU load as a percentage of the maximum.

`temperatureStatus`

Type: **string**

Value	Description
ok	The temperature is currently within the normal operating range.
outOfSpec	The temperature is currently outside the normal operating range.
critical	The temperature is too high and the device will shutdown if this condition persists.

`temperatureStatusWorst`

Type: **string**

The worst temperature status recorded on this device since it booted.

Value	Description
ok	The temperature has been within the normal operating range since the device was booted.
outOfSpec	The temperature has been outside the normal operating range at least once since the device was booted.
critical	At some point since the last boot the temperature was too high. The device will shutdown if this condition persists.

`rtcBatteryStatus`

Type: **string**

The current status of the RTC battery (Real Time Clock).

Value	Description
ok	The battery is operating within the normal range.
outOfSpec	The battery is operating outside of the normal range, and may require service.

rtcBatteryStatusWorst

Type: **string**

The worst recorded status of the RTC battery.

Value	Description
ok	The battery has been operating inside the normal range since the device was booted.
outOfSpec	The battery has operated outside of the normal range at some time since the device was booted.

voltagesStatus

Type: **string**

Value	Description
ok	The voltage is currently within the normal range
outOfSpec	The voltage is currently outside the normal range

voltagesStatusWorst

Type: **string**

Value	Description
ok	The voltage has been within the normal range since the device last booted.
outOfSpec	The voltage has been outside the normal range at some time since the device last booted

operationalStatus

Type: **string**

Value	Description
active	
shuttingDown	
shutDown	

device.network.query

Status: **active**

Queries the device for its network information. Some of the data listed below will be omitted if the interface is not enabled or configured. The query returns empty strings for addresses that are not configured.

Returns:

dns

Type: **array**

An array whose members represent the device's DNS parameters.

portA

Type: **array**

A structure that contains configuration and status information for Ethernet port A on the device.

portB

Type: **array**

A structure that contains configuration and status information for Ethernet port B on the device.

dns array

hostName

Type: **string**

The host name of queried device.

Deprecated in API version 2.8.

nameServer

Type: **string(63)**

The IP address of the name server, in dotted quad format (IPv4) or CIDR format (IPv6).

nameServerSecondary

Type: **string(63)**

The IP address of the secondary name server, in dotted quad format (IPv4) or CIDR format (IPv6).

domainName

Type: **string**

The domain name (DNS suffix).

Port arrays structure

enabled

Type: **boolean**

Value	Description
true	The port is enabled.
false	The port is not enabled.

ipv4Enabled

Type: **boolean**

true if IPv4 interface is enabled.

ipv6Enabled

Type: **boolean**

true if IPv6 interface is enabled.

linkStatus

Type: **boolean**

Value	Description
true	The ethernet connection to this port is active.
false	The ethernet connection to this port is not active.

speed

Type: **integer**

Speed of the connection on this Ethernet interface. One of 10, 100 or 1000, in Mbps.

fullDuplex

Type: **boolean**

Value	Description
true	The port can support a full-duplex connection.
false	The port can support a half-duplex connection.

macAddress

Type: **string**

The MAC address of this interface. A 12 character string of hex digits with no separators.

packetsSent

Type: **integer**

The number of packets sent from this Ethernet port.

packetsReceived

Type: **integer**

The number of packets received on this Ethernet port.

multicastPacketsSent

Type: **integer**

Number of multicast packets sent from this Ethernet interface.

multicastPacketsReceived

Type: **integer**

Number of multicast packets received on this Ethernet interface.

bytesSent

Type: **integer**

The number of bytes sent by the device.

bytesReceived

Type: **integer**

The number of bytes received by the device.

queueDrops

Type: **integer**

Number of packets dropped from the queue on this network interface.

collisions

Type: **integer**

Count of the network collisions recorded by the device.

transmitErrors

Type: **integer**

The count of transmission errors on this Ethernet interface.

receiveErrors

Type: **integer**

The count of receive errors on this interface.

bytesSent64

Type: **string**

64 bit versions of the bytesSent statistic, using a string rather than an integer.

bytesReceived64

Type: **string**

64 bit versions of the bytesReceived statistic, using a string rather than an integer.

The following parameters are returned only if the interface is enabled and configured.

dhcp

Type: **boolean**

Value	Description
true	The device's IP address is allocated by DHCP
false	The device's IP address is manually configured

ipAddress

Type: **string**

IPv4 IP address in dotted-quad format.

subnetMask

Type: **string**

The IPv4 subnet mask in dotted quad format.

defaultGateway

Type: **string**

The device's IPv4 default gateway in dotted quad format.

ipv6Address

Type: **string(63)**

The IPv6 address in CIDR format.

ipv6PrefixLength

Type: **integer**

The length of the IPv6 address prefix.

defaultIpv6Gateway

Type: **string(63)**

The address of the IPv6 default gateway in CIDR format.

linkLocalIpv6Address

Type: **string(63)**

The link local IPv6 address in CIDR format.

linkLocalIpv6PrefixLength

Type: **integer**

Length of the link local IPv6 address prefix.

device.query

Status: **active**

Returns high level status information about the device.

Returns:

currentTime

Type: **dateTime.iso8601**

The system's current time (UTC).

restartTime

Type: **dateTime.iso8601**

The date and time when the system was last restarted.

serial

Type: **string**

The serial number of the device.

chassisSerial

Type: **string**

The serial number of the chassis.

softwareVersion

Type: **string**

The version number of the software running on the device.

buildVersion

Type: **string**

The build version of the software running on the device.

model

Type: **string**

The model number.

apiVersion

Type: **string**

The version number of the API implemented by this device.

activatedFeatures

Type: **array**

Each member contains a string named **feature** containing a short description of that feature, for example, **Encryption**.

device.restartlog.query

Status: **active**

Returns the restart log - also known as the system log on the web interface.

Returns:

log

Type: **array**

Each member of the array contains log information (called system log in the user interface).

log array members

time

Type: **dateTime.iso8601**

The time when the device restarted.

Value	Description
20110119T13:52:42	yyyymmddThh:mm:ss

reason

Type: **string**

Value	Description
unknown	The software is unaware why the device restarted.
User requested shutdown	The device restarted normally after a user initiated a shutdown.
User requested upgrade	The device restarted itself because a user initiated an upgrade.

Feedback receivers

The interface between the calling application and the Cisco TelePresence device is stateless, so the API allows you to register your application as a feedback receiver. This means that the application doesn't have to constantly poll the device if it wants to monitor activity.

The device publishes events when they occur. If the device knows that your application is listening for these events, it will send XML-RPC messages to your application's interface when the events occur.

Use `feedbackReceiver.configure` to register a receiver to listen for one or more feedback events.

Use `feedbackReceiver.query` to return a list of receivers that are configured on the device.

After registering as a feedback receiver, the application will receive feedback messages on the specified interface.

feedbackReceiver.configure

Status: **active**

This call configures the device to send feedback about the specified **`events`** to the specified **`receiverURI`**. See the list of feedback events when you define the **`events`** array.

Accepts:

`receiverURI`

Type: **string**

Fully-qualified URI that identifies the listening application's XML-RPC interface (protocol, address, and port), for example, **`http://tms1:8080/RPC2`**. If this parameter is absent or empty, then the receiver will not receive notifications. You can use **`http`** or **`https`** and, if no port number is specified, the device will use the protocol defaults (80 and 443 respectively).

`sourceIdentifier`

Type: **string**

The device will use this optional parameter in feedback messages to this receiver (event notifications and **`feedbackReceiver.query`** responses). If this parameter is absent, the device uses the MAC address of its Ethernet port A interface.

`receiverIndex`

Type: **integer**

Sets the position of this feedback receiver in the device's table of feedback receivers.

Value	Description
-1	The feedback receiver will use any available position.
1	The first position is assumed if you don't supply this parameter (will overwrite existing entry in position 1)
20	The 20th (maximum allowed) position

events

Type: **struct**

An associative array mapping strings to booleans, where the string is the name of the feedback event and the boolean indicates if the feedback receiver wants to receive a notification of that event.

If this parameter is absent, then the receiver will be configured to receive the default notification messages (all notifications).

feedbackReceiver.query

Status: **active**

This call asks the device for a list of all the feedback receivers that have previously been configured. It does not accept parameters other than the authentication strings.

Returns:

receivers

Type: **array**

An array of feedback receivers, with members corresponding to the entries in the receivers table on the device's web interface.

Receiver members

Each receiver in the response contains the following parameters:

receiverURI

Type: **string**

Fully-qualified URI that identifies the listening application's XML-RPC interface (protocol, address, and port), for example, **`http://tms1:8080/RPC2`**. If this parameter is absent or empty, then the receiver will not receive notifications. You can use **`http`** or **`https`** and, if no port number is specified, the device will use the protocol defaults (80 and 443 respectively).

sourceIdentifier

Type: **string**

The device will use this optional parameter in feedback messages to this receiver (event notifications and **`feedbackReceiver.query`** responses). If this parameter is absent, the device uses the MAC address of its Ethernet port A interface.

index

Type: **integer**

The position in the device's table of feedback receivers that this receiver uses. A number between 1 and 20 (inclusive).

Feedback messages

The feedback messages follow the format used by the device for XML-RPC responses.

The messages contain two parameters:

- **`sourceIdentifier`** is a string that identifies the device, which may have been set by **`feedbackReceiver.configure`** or otherwise will be the device's MAC address.
- **`events`** is an array of strings that contain the names of the feedback events that have occurred.

Example feedback message

```
<params>
  <param>
    <value>
      <struct>
        <member>
          <name>sourceIdentifier</name>
          <value><string>000D7C000C66</string></value>
        </member>
        <member>
          <name>events</name>
          <value>
            <array>
              <data>
                <value><string>restart</string></value>
              </data>
            </array>
          </value>
        </member>
      </struct>
    </value>
  </param>
</params>
```

Feedback events

The following table lists the feedback events that this device can publish.

Event	Description
restart	The source publishes this event when it starts up.
configureAck	The source publishes this event to acknowledge that an application has successfully configured a feedback receiver.
alarmChanged	The Supervisor publishes this event when an alarm status changes.

Related information

Fault codes

Many of the software applications that run on Cisco TelePresence hardware will return a fault code when they encounter a problem while processing an XML-RPC request.

The following table describes all the fault codes used and their most common interpretations. Not all codes are used by all products.

Fault Code	Description
1	**`Method not supported`**. This method is not supported on this device.
2	**`Duplicate conference name`**. A conference name was specified, but is already in use.
3	**`Duplicate participant name`**. A participant name was specified, but is already in use.
4	**`No such conference or auto attendant`**. The conference or auto attendant identification given does not match any conference or auto attendant.
5	**`No such participant`**. The participant identification given does not match any participants.
6	**`Too many conferences`**. The device has reached the limit of the number of conferences that can be configured.
7	**`Too many participants`**. There are already too many participants configured and no more can be created.
8	**`No conference name or auto attendant id supplied`**. A conference name or auto attendant identifier was required, but was not present.
9	**`No participant name supplied`**. A participant name is required but was not present.
10	**`No participant address supplied`**. A participant address is required but was not present.
11	**`Invalid start time specified`**. A conference start time is not valid.
12	**`Invalid end time specified`**. A conference end time is not valid.
13	**`Invalid PIN specified`**. A PIN specified is not a valid series of digits.
14	**`Unauthorised`**. The requested operation is not permitted on this device.
15	**`Insufficient privileges`**. The specified user id and password combination is not valid for the attempted operation.
16	**`Invalid enumerateID value`**. An enumerate ID passed to an enumerate method invocation was invalid. Only values returned by the device should be used in enumerate methods.
17	**`Port reservation failure`**. This is in the case that reservedAudioPorts or reservedVideoPorts value is set too high, and the device cannot support this.
18	**`Duplicate numeric ID`**. A numeric ID was given, but this ID is already in use.
19	**`Unsupported protocol`**. A protocol was used which does not correspond to any valid protocol for this method. In particular, this is used for participant identification where an invalid protocol is specified.
20	**`Unsupported participant type`**. A participant type was used which does not correspond to any participant type known to the device.

21	**`No such folder`**. A folder identifier was present, but does not refer to a valid folder.
22	**`No such recording`**. A recording identifier was present, but does not refer to a valid recording.
23	**`No changes requested`**. This is given when a method for changing something correctly identifies an object, but no changes to that object are specified.
24	**`No such port`**. This is returned when an ISDN port is given as a parameter which does not exist on an ISDN gateway.
25	**`New port limit lower than currently active`**
26	**`Floor control not enabled for this conference`**
27	**`No such template`**. The specified template wasn't found.
28	**`No such connection`**. The specified connection was not found.
30	**`Unsupported bit rate`**. A call tried to set a bit rate that the device does not support.
31	**`Template name in use`**. This occurs when trying to create or rename a template to have the same name as an existing template.
32	**`Too many templates`**. This occurs when trying to create a new template after the limit of 100 templates has been reached.
33	**`Parameter value is not inside the expected range`**. A call supplied a value that is outside of the allowed range for this parameter.
34	**`Internal error`**
35	**`String is too long`**. The call supplied a string parameter that was longer than allowed. Strings are usually limited to 32 characters but may be 63 in some cases.
101	**`Missing parameter`**. This is given when a required parameter is absent. The parameter in question is given in the fault string in the format "missing parameter - parameter_name".
102	**`Invalid parameter`**. This is given when a parameter was successfully parsed, is of the correct type, but falls outside the valid values; for example an integer is too high or a string value for a protocol contains an invalid protocol. The parameter in question is given in the fault string in the format "invalid parameter - parameter_name".
103	**`Malformed parameter`**. This is given when a parameter of the correct name is present, but cannot be read for some reason; for example the parameter is supposed to be an integer, but is given as a string. The parameter in question is given in the fault string in the format "malformed parameter - parameter_name".
201	**`Operation failed`**. This is a generic fault for when an operation does not succeed as required.

HTTP keep-alives

Note: This feature is available from API version 2.4 onwards.

Your application can use use HTTP keep-alives to reduce the amount of TCP traffic that results from constantly polling the device. Any client which supports HTTP keep-alives may include the following line in the HTTP header of an API request:

```
Connection: Keep-Alive
```

This indicates to the device that the client supports HTTP keep-alives. The device may then choose to maintain the TCP connection after it has responded. If the device will close the connection it returns the following HTTP header in its response:

```
Connection: close
```

If this line is not in the HTTP header of the response, the client may use the same connection for a subsequent request.

The device will not keep a connection alive if:

- the current connection has already serviced the allowed number of requests
- the current connection has already been open for the allowed amount of time
- the number of open connections exceeds the allowed number if this connection is maintained

These restrictions are in place to limit the resources associated with open connections. If a connection is terminated for either of the first two reasons, the client will probably find that the connection is maintained after the next request.

Note: The client should never assume a connection will be maintained. Also, the device will close an open connection if the client does not make any further requests within a minute. There is little benefit to keeping unused connections open for such long periods.

system.xml file

You can derive some information about the device from its **system.xml** file. You can download this file via HTTP from the device's root.

Example **system.xml**

```
<?xml version="1.0"?>
  <system>
    <manufacturer>Codianlt;/manufacturer>
    <model>MSE Supervisor 8050</model>
    <serial>SM0100A7</serial>
    <chassisSerial>XX7000E9</chassisSerial>
    <softwareVersion>2.2(1.12)</softwareVersion>
    <buildVersion>8.6(1.12)</buildVersion>
    <hostName></hostName>
    <uptimeSeconds>8174</uptimeSeconds>
  </system>
```

Checking for updates and getting help

We recommend registering your product at http://www.tandberg.com/services/video-conferencing-product-registration.jsp in order to receive notifications about the latest software and security updates. New feature and maintenance releases are published regularly, and we recommend that your software is always kept up to date.

If you experience any problems when configuring or using the product, consult the documentation at http://www.tandberg.com/support/video-conferencing-documentation.jsp for an explanation of how its individual features and settings work. You can also check the support site at http://www.tandberg.com/support/ to make sure you are running the latest software version.

You or your reseller can also get help from our support team by raising a case at http://www.tandberg.com/support/. Make sure you have the following information ready:

- The software build number which can be found in the product user interface (if applicable).
- Your contact email address or telephone number.
- The serial number of the hardware unit (if applicable).

References

1. XML-RPC specification (Dave Winer, June 1999); http://www.xmlrpc.com/spec, accessed 24/01/2011.
2. HTTP/1.1 specification (RFC 2616, Fielding et al., June 1999); http://www.ietf.org/rfc/rfc2616.txt, accessed 24/01/2011.

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Cisco and the Cisco Logo are trademarks of Cisco Systems, Inc. and/or its affiliates in the U.S. and other countries. A listing of Cisco's trademarks can be found at www.cisco.com/go/trademarks. Third party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1005R)

Any Internet Protocol (IP) addresses and phone numbers used in this document are not intended to be actual addresses and phone numbers. Any examples, command display output, network topology diagrams, and other figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses or phone numbers in illustrative content is unintentional and coincidental.