CCXML Reference

CCXML Reference

4/3/2026

CCXML Reference

9.x This version of the CCXML Reference applies to Genesys Voice Platform that is part of 9.0, starting with version 8.5. For version 8.1 of Genesys Voice Platform, see the Genesys Voice Platform home page.

CCXML provides a standard, xml-based language for scripting call control logic.
[+] What is Call Control Extensible Markup Language (CCXML)?

The Call Control Platform (CCP) component of Genesys Voice Platform (GVP) provides a **Call Control Extensible Markup Language (CCXML)** interpreter that integrates with existing GVP infrastructure such as the Media Control Platform (MCP) and the Resource Manager (RM). The underlying network protocol for the CCP is SIP, which means that the CCP is also interoperable with other conferencing servers or dialog servers.

Although GVP has traditionally provided extended call control capabilities through Voice Extensible Markup Language (VoiceXML), the development of CCXML provides a standard, xml-based language for scripting call control logic. Like VoiceXML, CCXML is independent of the environment in which it operates, and can run in environments ranging from Voice over IP (VoIP) based softswitch products to integrated residential gateways that manage a single telephone call. Genesys therefore recommends that new call control applications use CCXML.

The CCP currently follows the W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005. Consult the web site at this URL: http://www.w3.org/TR/2005/WD-ccxml-20050629/.

Intended Audience

This document is primarily intended for users who will be developing call control applications with Call Control Extensible Markup Language (CCXML). It has been written with the assumption that you have a basic understanding of:

- Computer-telephony integration (CTI) concepts, processes, terminology, and applications
- Network design and operation
- Your own network configurations

You should also be familiar with HTML, XML, CCXML, and VoiceXML concepts.

Features

Features

Dialing into the CCP

Dialing into the CCP

The Call Control Platform (CCP) accepts incoming SIP connections on port 5068 by default. You can change the port number by adjusting the configuration variable sip.transport.x to modify the port number used by the CCP for receiving incoming connections. *Calling to the Default CCXML Page* By default, all incoming connections will start a new CCXML session with a default URI that is specified by the GVP CCP configuration value ccpccxml.default-uri. By default it is:

```
file://$InstallationRoot$/config/default.ccxml
```

.

The installation root (directory path) can be specified differently by people who install the CCP, for Windows and for Linux.

Starting a non-Default CCXML Page

To start a different page with an incoming connection, the CCP follows the netann convention, currently:

```
http://www.ietf.org/rfc/rfc4240.txt
```

.

The request URI of the incoming request must follow this format:

```
sip:ccxml@callcontrolplatform.genesyslab.com;ccxml= http://www.genesyslab.com/page.ccxml
```

Where: The userpart of the Request URI must be ccxml and it is case-sensitive. callcontrolplatform.genesyslab.com is the host or IP address of the CCP. The Request URI must contain a ccxml URI parameter (case-sensitive) and the value is the CCXML page to be started. Other URI parameters can be included and the ordering does not matter. These additional parameters are passed to the CCP, and may be consumed by the CCP, or passed to the application server referred to by the Request URI.

Using Resource Manager to Map CCXML Applications

The GVP Resource Manager (RM), which acts as a SIP proxy, can be used to map CCXML applications by translating the SIP request URI to the netann format described in the preceding sub-section.

Here is an example:

The RM translates

```
sip:1234@10.0.0.123
```

into:

```
sip:ccxml@10.0.0.124:5068;ccxml=file:///usr/local/ccp-ccxml/config/default.ccxml
```

where 10.0.0.123 is the RM address. 10.0.0.124 is the CCP address.

For information about configuring Resource Manager, see the Genesys Voice Platform 8.1 User's Guide.

Inbound Connections

Inbound Connections

Passing URI Parameters to CCXML Applications

When fetching the initial CCXML page, the CCP appends the incoming SIP request URI parameters to the CCXML URL.

For example, a SIP Request URI looks like this:

```
sip:ccxml@ccxmlplatform.genesyslab.com:5068;ccxml=
  http://www.genesyslab.com/page.ccxml;hello=world
```

The initial URL fetch will be:

```
GET http://www.genesyslab.com/page.ccxml?hello=world
```

Call Parameters Accessible in CCXML Applications Call parameters (or SIP headers) can be made accessible in the CCXML application through the session object. Table 1 shows the connection properties available through the session object.

Table 1: Connection Properties

Connection Properties (Shown via Session Variable)	Description
session.connections[connectionid].local	To: header
session.connections[connectionid].remote	From: header
session.connections[connectionid].protocol.name	sip

session.connections[connectionid].protocol.version	2.0
session.connections[connectionid].protocol.sip.callid	Call-ID header
session.connections[connectionid].protocol.sip.requesturi	Request URI
session.connections[connectionid].protocol.sip.from	From: header
session.connections[connectionid].protocol.sip.to	To: header

183 Session Progressing Response

The CCP sends a 100 Trying response immediately upon receiving an INVITE request. The CCP sends a 200 OK response when the CCXML application executes the <accept> tag. By default, the 183 Session Progressing response message is not sent.

Setting ccpccxml.sip.send_progressing configuration parameter to 1 instructs the CCP to send 183 Session Progressing along with 200 OK when the <accept> tag is executed on an inbound connection. A CCXML application can request that the CCP send 183 Session Progressing with a <send> tag. Here is an example:

```
<send target=connectionid targettype=x-connection data=connection.progressing />
```

Rejecting Incoming Connections

Rejecting an incoming connection with the <reject> tag will cause the CCP to respond with a 480 Temporarily Unavailable response. Using the reason attribute in the <reject> tag will enable the use of the Reason header in the 400 Bad Request response. The header will contain the following:

Reason: SIP; cause=480; text= content of the reason attribute The exact SIP response code used to reject a call can be specified in the hints attribute of the <reject> tag. The responseCode property of the hints object specifies the response code that should be used, as shown below.

```
  <var name="hints" expr="new Object()"/>
  <assign name="hints.responseCode" expr="'400'"/>
. . .
<reject . . . hints="hints"/>
```

The default reject SIP code is configurable throughout the CCP and will be used if the hints attribute is not specified. The parameter mediacontroller.defaultrejectcode is the platform-wide configuration parameter for the default reject code.

Disconnecting Calls

When a connection is connected, executing the <disconnect> tag sends a BYE message on the connection to terminate the call. This applies to both inbound and outbound connections.

Using the reason attribute in the <disconnect> tag enables the use of the Reason header in the BYE message. The header will contain the following: `Reason: SIP; cause=200; text= content of the reason attribute`

Connection Signals

When a SIP INFO message is received on a connection, the CCP raises a connection.signal event. There will be two properties set in the info property:

- **event.info.contenttype** the value of Content-Type header of the SIP INFO message
- **event.info.content** the content of the SIP INFO message

Receiving DTMF Digits Through SIP INFO

When a SIP INFO message has a Content-Type of application/dtmf-relay, it implies that it is a DTMF digit. The connection.signal event will also contain the event.info.dtmf property that provides the DTMF digit(s).

Receiving Other Events Through SIP INFO

Other events are stored as text into event.info.contenttype and event.info.content. The SIP INFO message body is stored in event.info.content property, whereas the value of the SIP Content-Type header is stored in event.info.contenttype. Event content can be parsed using ECMAScript, as shown in the example below:

```
INFO sip:101@10.33.2.53;user=phone SIP/2.0
Via: SIP/2.0/UDP 10.33.2.53;branch=z9hG4bKac5906
Max-Forwards: 70
From: "anonymous" <sip:anonymous@anonymous.invalid>;tag=1c25298
To: <sip:101@10.33.2.53;user=phone>
Call-ID: 11923@10.33.2.53
CSeq: 1 INVITE
Contact: <sip:100@10.33.2.53>
X-Detect: Response=CPT,FAX
Content-Type: application/x-detect
Content-Length: xxx

Type = CPT
Subtype = reorder
```

```
    <transition event="connection.signal" name="evt">
      <if cond="evt.info.contenttype.toString() == 'application/x-detect'">
        <script>
          <![CDATA[
               var mystring = evt.info.content.split("\r\n");
               var myType1 = mystring[0].split("=");
                  var myType2 = mystring[1].split("=");
           ]]>
```

```
      </script>
        <log expr="myType1[0] + '[' + myType1[1] + ']'"/>
        <log expr="myType2[0] + '[' + myType2[1] + ']'"/>
      <else/>
        <log expr="'Unwanted Event'"/>
    </if>
    <exit/>
  </transition>
```

The log will display: `Type[CPT]`, and `Subtype[reorder]`.

Outbound Connections

When making an outbound connection, provide the SIP URI in the dest attribute of the <createcall> tag. The CCP uses the given SIP URI to send an INVITE request. The SIP Request is sent directly to the destination.

Specifying Custom SIP Headers Through Hints

Custom SIP headers can be sent in an initial outgoing INVITE through the hints attribute of the <createcall> tag. The value of the hints attribute must be an ECMAScript object containing a subobject with the name headers. The headers ECMAScript object can then contain a name/value list of custom SIP header names and the corresponding values:

```
<var name="myhint" expr="new Object()"/>
<assign name="myhint.protocol" expr="new Object()"/>
<assign name="myhint.protocol.sip" expr="new Object()"/>
<assign name="myhint.protocol.sip.headers" expr="new Object()"/>
<assign name="myhint.protocol.sip.headers['X-Detect']" expr="'Request=CPT, FAX'"/
. . .
<createcall . . . hints="myhint" . . . />
```

The header name/value specified through the hints will be filtered through an allowed list of custom SIP headers defined by the configuration variable mediacontroller.sip.allowedunknownheaders, which is a space delimited list of permitted custom header names.

> **Important**
>
> If the mediacontroller.sip.allowedunknownheaders is set to "*", all unknown headers will be sent out. Also, it should be noted that mediacontroller.sip.allowedunknownheaders="* X-Detect" will not work; only "*" will work in this case.

If the header name specified in the hint has a matching header name in the configuration parameter, the first matching header name from the configuration parameter will be sent out as a custom header name with the value specified from the hint. Header names and values with no matching header name in the configuration parameter will not be sent out.

For example, if the `mediacontroller.sip.allowedunknownheaders` configuration parameter has the value of X-Detect X-other when <createcall> is called with the preceding hint example, the custom header X-Detect: Request=CPT,FAX will be added to the initial INVITE.

The table below lists some examples of the mapping between hint header names and custom header names sent out in the INVITE message:

Header name in hint	`ccpccxml.sip.allowedunknownheaders`	Header name in SIP INVITE
X-Detect	X-Detect X-Channel	X-Detect
CPA	A-CPA CPA B-CPA	CPA
CPA	CPA A-CPA B-CPA	CPA
CPA	X-Detect X-Channel	Not sent

Similarly, custom SIP headers in SIP responses that result in a connection.progressing or connection.connected event (see Mapping SIP Responses to CCXML Connection Events , below) will be available to the CCXML application if the SIP headers are configured in the `mediacontroller.sip.allowedunknownheaders` configuration parameter.

The custom SIP headers can be obtained from the CCXML application as follows:

```
session.connections[evt.connectionid].protocol.sip.headers['x-channel']
```

where x-channel is the SIP header name mentioned above.

> **Important**
>
> When the mediacontroller.sip.allowedunknownheaders parameter is changed, the CCP must be restarted to get the latest parameter changes.

Mapping SIP Responses to CCXML Connection Events

All 1xx responses except 100 received from the outgoing connection result in a connection.progressing event.

When a 2xx response is received, a connection.connected event is thrown.

When a non-2xx final response (300 699) response is received, a connection.failed event is thrown.

Disconnecting Progressing Call

When the <disconnect> tag is used on an outbound progressing call, the CCP sends a CANCEL message on the outgoing call to terminate it.

Feature Call Redirection

Redirecting an Incoming Call

Using the <redirect> tag on an incoming call (in the ALERTING state) redirects the call. The CCP sends a 302 Moved Temporarily response. The dest attribute of the <redirect> tag translates to the Contact header in the 302 response.

Redirecting a Connected Call

Using the <redirect> tag on a connected call (this applies to both inbound and outbound calls) redirects the call with a REFER message. The dest attribute of the <redirect> tag translates to the Refer-To header in the REFER message. After the CCP receives a NOTIFY message with a 200 OK message, the call is considered redirected and the connection will be released. The CCXML application will receive a connection.redirected event.

If the redirection fails for any reason, the call receives an error.connection event.

Call Merge

Two connections can merge at the network level (bridging the calls at the switch) when both of them are in a CONNECTED state. The CCP uses the REFER message with Replaces as the mechanism to initiate a call merge feature at the switch. For example: Assume the first call was connected with:

```
INVITE sip:hi@10.0.0.1 SIP/2.0
Via: SIP/2.0/UDP
From: sip:bye@10.0.0.2
To: sip:hi@10.0.0.1
Max-Forwards: 70
CSeq: 1 INVITE
Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E
Contact: sip:bye@10.0.0.2:5060
Content-Length: 147
Content-Type: application/sdp
```

Assume the second call was connected with:

```
INVITE sip:hello@10.0.0.1 SIP/2.0
Via: SIP/2.0/UDP
From: sip:world@10.0.0.3
To: sip:hello@10.0.0.1
Max-Forwards: 70
CSeq: 1 INVITE
Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123
Contact: sip:world@10.0.0.3:5060
Content-Length: 147
Content-Type: application/sdp
```

Table 3 describes the Merge SIP call flow.

Event	Direction	Message
<merge>	→	REFER sip:bye@10.0.0.2 SIP/2.0 Via: SIP/2.0/UDP 10.0.0.1:5060 From: sip:hi@10.0.0.1 To: sip:bye@10.0.0.2 Cseq: 2 REFER Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E Refer-To: world@10.0.0.3;Replaces=DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123
	←	SIP/2.0 202 Accepted Cseq: 2 REFER
connection.merged	←	NOTIFY sip:bye@10.0.0.2 SIP/2.0 Cseq: 3 NOTIFY Event: refer Content-Type: message/sipfrag;version=2.0 Content-Length: 14 SIP/2.0 200 OK
	→	SIP/2.0 200 OK Cseq: 3 NOTIFY
	→	BYE sip:bye@10.0.0.2 SIP/2.0 Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E

	→	BYE sip:world@10.0.0.3 SIP/2.0 Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123
	←	SIP/2.0 200 OK Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E
	←	SIP/2.0 200 OK Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123

> **Important**
>
> If the CCXML application has a <merge> tag followed immediately by a <disconnect> tag in the same transition, the platform will issue only one connection.merged event and one connection.disconnected event instead of two connection.merged events on both of the connections.

Dialogs

Preparing Dialogs

The <dialogprepare> tag and optional <dialogstart> tag create a new dialog; the CCP initiates a new SIP dialog to the dialog server. The CCP sends an INVITE message to the Resource Manager (configurable with the mediacontroller.sipproxy parameter) with the following netann request URI:

```
sip:dialog@sipproxy.genesyslab.com;voicexml=http%3F//www.genesyslab.com/page.vxml
```

...where `sipproxy.genesyslab.com` is the value of the configuration parameter `mediacontroller.sipproxy`.

Using <dialogprepare> to prepare a dialog will send a connectionless SDP to the dialog server to let the dialog server (probably Media Control Platform, see the note below) prepare the dialog without starting the audio. When the INVITE transaction is ACKed, the dialog is fetched and loaded on the MCP, which then waits for the INVITE response before proceding.

> **Important**

> Older Genesys installations may be using the Stream Manager as a dialog server, but this is infrequent.

A connectionless SDP represents SDP content that would put the MCP on hold. The SDP content will depend on the device profile configuration of the dialog server.

Passing Dialog Results Back to CCXML

VoiceXML pages can return results to the CCP by adding content to the BYE message. The VoiceXML page can use the namelist attribute in the <exit> tag to return dialog results to the CCXML application.

Here is an example in which the VoiceXML application ends the call with `<exit namelist="hello a"/>`:

The MCP sends BYE to the CCP:

```
BYE sip:10.0.0.1:5060 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.3
Via: SIP/2.0/UDP 10.0.0.2:5060
From: sip:genesyslab@10.0.0.2
To: sip:10.0.0.1:5060
Max-Forwards: 69
CSeq: 1 BYE
Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E
Content-Length: 16
Content-Type: application/text

hello=world
a=b
```

The dialog.exit event contains:

```
values.hello = 'world'
values.a = 'b'
namelist='hello a'
```

The dialog.disconnect event is not currently supported by the CCP. When a VoiceXML application exits, dialog.exit will be raised.

Dialog User Event

The VoiceXML dialog may send a user event to the CCXML application by using the <send namelist="name type uri"/> tag. Here is an example of the VoiceXML <send> block:

```
<var name="name" expr="'transfer'"/>
<var name="type" expr="'bridge'"/>
<var name="uri" expr="'1111@205.150.90.19'"/>
<gvp:send namelist="name type uri"/>
```

The CCXML session receives the following:

```
15:02:04.416 Int 51030 F9187A00-E558-44C6-61AE-FFA9A066180C-FF326086-ECB5 dlg_event 7|
```

```
dialog.user.transfer|DD92E8B2-51AD-4F3F-8C8D-40AFA169EA9B|
values.name="transfer";values.type="bridge";values.uri="1111@205.150.90.19
```

This raises a dialog.user.transfer event to the CCXML application that owns the dialog. The event itself contains the following properties:

- **values.name=transfer** (available as `dialog.user.transfer.values.name` in the example above)
- **values.type=bridge** (available as `dialog.user.transfer.values.type` in the example above)
- **values.uri=1111@ 205.150.90.19** (available as `dialog.user.transfer.values.uri` in the example above)

> **Important**
>
> The `contenttype` attribute is not supported by the <send> tag if the namelist is used.

Dialog-Initiated Blind Transfer

To begin a dialog-initiated blind transfer, the VoiceXML application must call `<transfer destexpr="number_to_call" bridge="false" type="unsupervised">`.

The following sequence of events occurs:

- The MCP sends a REFER message on the SIP dialog.
- The CCXML application receives a dialog.transfer event. The type attribute is blind and the uri attribute is the destexpr in the <transfer> tag.
- The CCXML application executes <redirect> to move the call specified in the dialog.transfer event.
- If redirection is successful, the CCXML application sends telephone.disconnect.transfer event to the dialog.
- The CCP sends NOTIFY (200 OK) to report the result of the transfer.
- If redirection fails, the CCXML application sends error.transfer.noroute event to the dialog.
- The CCP sends NOTIFY (500 Server Internal Error) to report a transfer failure.
- The VoiceXML application receives a telephone.disconnect.transfer event to end the transfer and the VoiceXML page. The result is recorded in the metrics file of the MCP.

> **Important**
>
> When the inbound call is made through SIP Server, a dialog-initiated blind transfer will only work if the SIP Server has the appropriate DN trunk group set up and enabled to do refer transfer. Configure this by setting the 'refer-enabled<equals>true' on the SIP Server.

Dialog-Initiated Supervised Transfer

A dialog-initiated supervised transfer is application driven in both the MCP and CCP. The MCP sends a SIP REFER message to the CCP when the VoiceXML <transfer> is invoked for a supervised transfer. The CCP will throw a dialog.transfer event to the application with the type attribute of the event set to blind.

> **Important**
>
> The type attribute is always populated to blind whether the request from the VoiceXML is for blind transfer or supervised transfer. The CCP application developer should write the application according to either blind or supervised transfer.

Dialog-Initiated Bridge Transfer

A dialog-initiated bridge transfer is application-driven from both the MCP and CCP perspective. Within the VoiceXML application, you can use the <send> tag (translating to SIP INFO) to inform the CCP of a bridge transfer request.

The CCP application can be written according to the W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005, Appendix D for bridge transfer.

MSML Dialogs

Media Server Markup Language (MSML) allows CCXML applications to have additional control over the media operations beyond what VoiceXML can offer. For example, the customer can create a MSML script that performs Call Progress Analysis (CPA), and then, based on the CPA result, it can choose to either start the VoiceXML dialog or play a prerecorded prompt.

Genesys CCXML supports MSML dialogs (the dialog packages include Dialog Core, Dialog Base, and Dialog CPA) by extending the CCXML specification. CCMXL does not support the MSML Conference Core package. For the list of supported tags, see Appendix C on page 51.

When `<dialogstart>` is used without `<dialogprepare>`, you must set the type attribute to `application/vnd.radisys.msml+xml`. In this case, the src attribute content is ignored. You can specify the MSML body inline as follows:

> **Important**
>
> For <dialogprepare>, the same rules for <dialogstart> applies on type, src, and the inline MSML body. The inline MSML body should not contain the <?xml> tag.

The developer should include the attribute xmlns as shown below so that you can inline MSML

markups in the CCXML document.

```
<dialogstart src="." type="application/vnd.radisys.msml+xml" connectionid="..."
xmlns="urn:ietf:params:xml:ns:msml">
<msml>
<dialogstart type="'application/moml+xml'">
<play>
<audio uri="'http://example.com/dictionary.vox'"/>
</play>
</dialogstart>
</msml>
</dialogstart>
```

Alternatively, you can use <dialogprepare> before a <dialogstart>. If you use <dialogprepare>, the type, src attributes, and the inline MSML body are required by the <dialogprepare> tag and must not be repeated in the <dialogstart> tag. The MSML information will be sent with the initial INVITE generated by the <dialogprepare> with an on-hold SDP.

```
<dialogprepare src="." type="application/vnd.radisys.msml+xml" connectionid="connectionid"
xmlns="urn:ietf:params:xml:ns:msml">
<msml>
<dialogstart type="'application/moml+xml'">
<play>
<audio uri="'http://example.com/dictionary.vox'"/>
</play>
</dialogstart>
</msml>

</dialogprepare>
.
.
.
<dialogstart prepareddialogid="dialogid"/>
```

Where `connectionid` and `dialogid` are generated by CCXML.

Example of the initial INVITE for an MSML dialog

```
INVITE sip:dialog@genesyslab.example.com; moml=cid:14864099865376@genesyslab.example.com SIP/
2.0
...
Content-Type: multipart/mixed; boundary=boundary
--boundary
Content-Type: application/sdp
SDP BODY
--boundary
Content-Id: <14864099865376@genesyslab.example.com>
Content-Type: application/vnd.radisys.msml+xml
```

The resulting MSML body from the preceding example should look like this:

```
<?xml version="1.0"?>
<msml>
<dialogstart type="'application/moml+xml'">
<play>
<audio uri="'http://example.com/dictionary.vox'"/>
</play>
</dialogstart>
</msml>
```

Note that CCP does not modify the MSML markups in any way other than placing the XML header at the top. For example:

```
--boundary--
```

Once the dialog is started, Media Server can send MSML information using SIP INFO messages.

```
INFO sip:dialog@genesyslab.example.com SIP/2.0
...
Content-Type: application/vnd.radisys.msml+xml
<?xml version="1.0"?>
<msml>
<event name="msml.dialog.exit" id="conn:1234/dialog:1234"/>
</msml>
```

The MSML body is available through the `dialog.user.msml events info.content` attribute, with `info.contenttype = "application/vnd.radisys.msml+xml"`

The info.content will look like the following:

```
<?xml version="1.0"?>
<msml>
<event name="msml.dialog.exit" id="conn:1234/dialog:1234"/>
</msml>
```

VoiceXML Session Variables

The MCP does not support the session.connection.ccxml VoiceXML session variable described in Appendix D of the CCXML specification, here:

```
http://www.w3.org/TR/2005/WD-ccxml-20050629/#Appendix-VoiceXML-SessionVars
```

Conferences

When a CCXML application joins to a conference, the CCP sends an INVITE message to the RM with a specially formatted netann Request URI:
`sip:conf=ABCD1234@10.0.0.1;confinstid=ABCD1234;confreserve=3;confmaxsize=3` where

- `conf`, `confinstid` are cluster-wide unique conference identifiers
- `confreserve` is the number of conference participants to reserve for this conference
- `confmaxsize` is the maximum size of this conference

The sum of the reservedtalkers and reservedlisteners attributes in the <createconference> tag represents the number of conference participants to reserve for this conference. The default value can be set using the configuration parameter mediacontroller.conference.defaultreserve. The maximum size of the conference is equal to confreserve by default. This value can be set in the hints attribute of the <createconference> tag; it is the maxsize property of the hints object. In a clustered environment where multiple conference servers are available, the CCP relies on GVP RM to forward the requests for the same

instance of a conference to the same conference server. This is a feature of the RM.

> **Important**
>
> The <createconference> tag proceeds successfully even if the cluster has no conferences available or no conference servers can serve the requested conference size. A <join> operation may fail due to the preceding reasons and returns error.conference.join event.

Implicit Transcoding and Conferencing

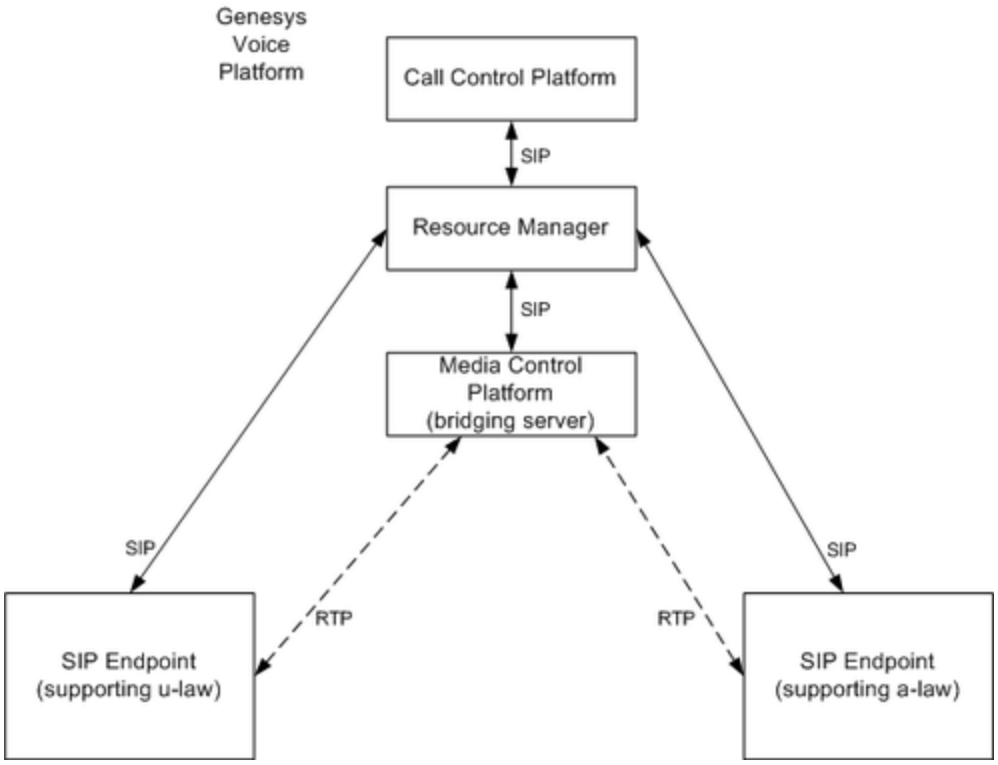
The Resource Manager can be configured with bridging server information, to handle CCXML operations that require implicitly connecting endpoints to a media server. The MCP can be used as a bridging server. There are two cases in which a bridging server may be used internally by the CCP:

- Audio-transcoding between endpoints which do not share common codecs
- Multiple sessions listening to a single media stream through the use of an RTP splitter/proxy or an implicit conference

The CCP determines whether implicit transcoding or conferencing is required upon evaluation of the CCXML application. The connection of the endpoints to the bridging server will be transparent to the CCXML application.

Implicit Transcoding

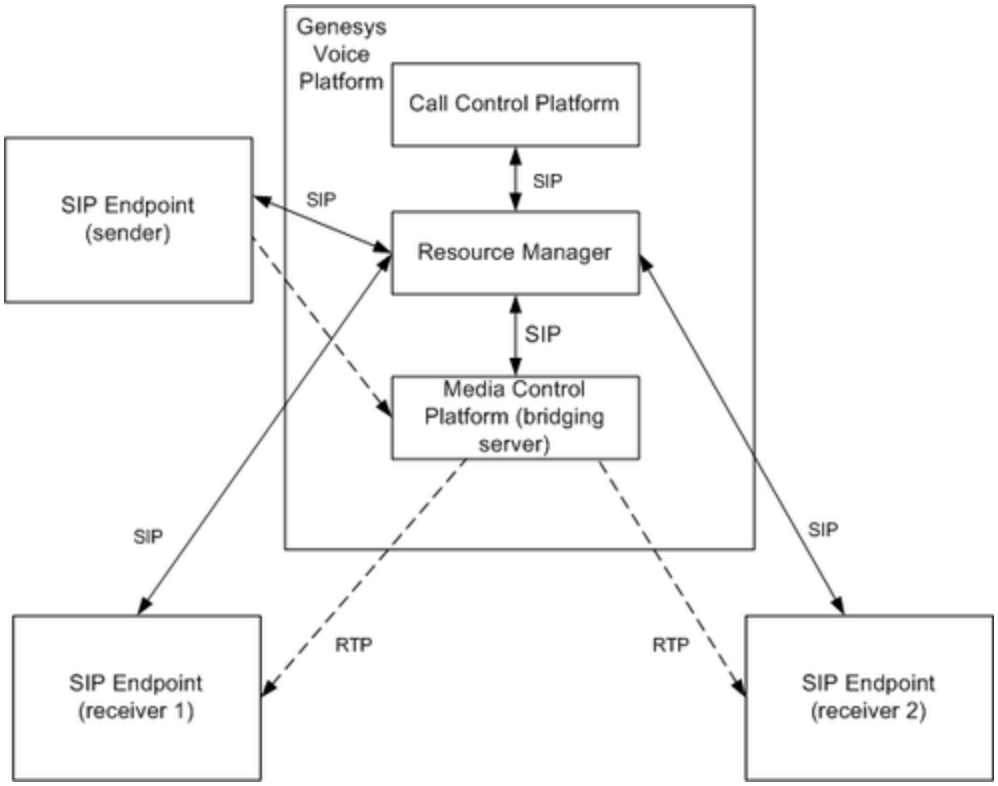
When a CCXML session specifies a join between two endpoints that do not share any common audio codecs, the CCP uses the bridging server internally to transcode the media between the endpoints (see Figure 1 below).



> **Important**
>
> If call legs are being joined implicitly with the <createcall> tag, and the call legs do not provide their codec capabilities either in the initial INVITE to CCP or in the 200 OK response to the initial INVITE sent by CCP, the CCP will not use the transcoding feature even if a transcoding server has been defined in the mediacontroller.bridge_server option. This is because the CCP requires knowledge of the codec capabilities of each call leg before creating bridges so that it can determine whether a bridging server is needed. The workaround is to connect each call leg separately and join them together after they are in the connected state.

Implicit Conferencing

A CCXML session can specify multiple joins to a single endpoint so that it is required to split its output and send to multiple destinations. If the sender does not support splitting its output to send to multiple destinations concurrently, the CCP internally uses the Bridging Server to do the RTP media splitting as shown in Figure 2.



The bridging server can be accessed directly from the CCP or via the RM. The mediacontroller.bridge_server configuration parameter is used to specify the location of the bridging server. If the bridging server is to be accessed via the RM, the location of the RM should be specified in the configuration parameter mediacontroller.bridge_server.

The MCP can be configured to act as a bridging server.

Profile Configuration

The CCP provides a set of device profiles that reflect Genesys current knowledge regarding the behavior of various devices that interact with the platform. For additional information about the configuring device profiles, see the Genesys Voice Platform 8.1 User's Guide.

Inbound Connections

The CCP provides regular expression matching for incoming connections in order to select the most appropriate Device Profile for these connections.

The CCP will try to match the SIP User-Agent header from the incoming SIP INVITE to the SIP Header Name property value that is defined in the Device Profile with the highest precedence first. If there is a match, the matching Device Profile will be assigned to the connection and the CCP will use the Device Profile parameters to determine the correct behavior.

If there is no match with the SIP Header Name property value, the CCP will try to match the SIP User-

Agent header to the Device Profile with the next precedence. If there are no matches with any preset Device Profile, the Default Inbound Device Profile will be used for the connection.

Outbound Connections

The device profiles of outbound connections, dialogs, and conferences can be specified in CCXML hints. The value of the device profile hint should be the same value as the Device Profile Name from the Device Profile configuration. The example below illustrates the use of hints for an outbound connection.

Similarly, the same hint can be used for <dialogprepare>, <dialogstart>, and <createconference>. If the hint is already passed in <dialogprepare>, the subsequent <dialogstart> should not reconfigure the device profile hint.

> **Important**
>
> The timeout value for <createcall> may not work correctly if the value is less than 32 seconds. If the destination does not respond at all (for example, a device is down), the timeout will not occur until after the 32 second period of INVITE re-transmission (as dictated by RFC3261) has passed.

```
<var name="myhint" expr="new Object()">
<assign name="myhint.deviceprofile" expr="'GVP MCP'">
. . .
<createcall . . . hints="myhint">
```

Limitations

- The CCP supports up to 100 conference participants in a conference.
- Forward join to a conference that is running on an MCP will not join properly. Avoid this limitation by always using duplex join to join a conference.
- In the Offer-Answer case, an incoming connection cannot be joined to two outbound connections if the alerting connection is accepted in a later transition than the joins. Avoid this limitation by accepting the alerting connection in the same transition as the joins or in an earlier transition.
- If an existing media loop is completely reversed in the same transition, some media might be missed at some endpoints.

For example, initially `A->B->C->A`

- The CCXML application contains multiple joins to reverse the initial bridges to `A<-B<-C<-A`.

Avoid this by unjoining the bridges in separate transitions.

- The options-support Device Profile parameter should be set to false when the CCP is used in conjunction with Resource Manager (RM) or a SIP Proxy. If the CCP is directly connected to SIP User Agents, the options-support feature can be used to query the SIP capabilities of the devices.

- If the Default Conference device profile is not configured or <createconference> refers to the device profile that is invalid, `error.conference.create` will be thrown. The error.conference.create event in this case will not contain the `conferenceid` since the <createconference> operation was aborted prematurely.

Event Processors

Event Processors

Session Variable

The session variable session.ioprocessors is an associative array of external event I/O access URIs available to the current session, keyed by the type of the event I/O processor. Currently, this array has only two items:

- `session.ioprocessors["basichttp"]`
- `session.ioprocessors["createsession"]`

Receiving Events

The URI that accepts the request is available through the session.ioprocessors["basichttp"] session variable described in the previous section. If the event I/O processor can satisfy the HTTP request, an event is ejected into an active CCXML session; otherwise, if the I/O processor is not able to satisfy the request (because the request was malformed, or an error occurred, or the requested session does not exist, or the operation was not permitted, for example), then no action is taken. The event I/O processor then reports the result of the request to the external client in the response to the request. HTTP POST request parameters (within an application/x-www-form-urlencoded message body) are used to specify the information that is to be injected into the session. In particular, the parameters shown in Table 4 have special meanings:

Table 4: HTTP Request Parameters

HTTP Parameter Name	Meaning
sessionid	This is the ID of the CCXML session

	that will receive the injected event. This parameter is required.
name	This is the name of the event that will be injected. This parameter is required. Valid event names consist of alphanumeric characters and periods only. The first character of an event name must be a letter.
eventsource	This value specifies a URI to which events may be sent (that is, it may be used as the value of the target attribute in a <send> element). This parameter is optional and can be in any form.

When an event is successfully injected into the target session, the evensourcetype property of the event object is set to basichttp and the eventid property contains a unique event id (generated by the basichttp event I/O processor) for the event. When provided in the HTTP request as parameters; eventsourcetype or eventid parameters are ignored.

Other parameters provided in the HTTP request are treated as the event payload. Payload parameter names must be valid ECMAScript variable names. Qualified parameter names (for example, x.y.z) are nested inside parent parameters (for example, y and its parent x). Reserved and payload parameter values must be valid ECMAScript expressions.

The CCP replies to the HTTP request with one of the HTTP response codes shown in Table 5 on page 35.

Table 5: HTTP Response Codes

Response Code	Condition
204"No Content"	The sessionid parameter matches an existing CCXML session ID, and the

	event name and payload parameters are valid.
400"Bad Request"	One or more parameters has an invalid name or value or there are conflicts (for example, both x and x.y defined).
403"Forbidden"	Failure occurs due to other reasons (for example, the session ID does not match an existing CCXML session ID, or the matched session is terminating).

Table 6 describes the event attributes of an event that was successfully received via HTTP request:

Table 6: Event Attributes

Attribute Name	Description
name	The value of the name attribute.
eventid	A unique string identifier for the event generated by the CCP.
eventsource	The value of the eventsource parameter if provided; otherwise this event attribute is undefined.
eventsourcetype	Always has the value basichttp.

<param-name>	For each param-name=value appearing in parameter name in the HTTP request, the parameter param-name appears as a property (or a nested property) in the event object. Its value is set to value.

HTTPS Support

To use secure HTTP (HTTPS), set the following GVP configuration parameters:

- `ccxmli.ssl` must be `true`.
- `ccxmli.ssl.recv.cert_file` sets the path and the filename of the SSL certificate to be used for createsession and BasicHTTP.
- `ccxmli.ssl.recv.private_key_file` sets the path and the filename of the SSL key to be used for createsession and BasicHTTP.
- `ccxmli.ssl.recv.password` sets the password associated with the certificate and key pair. Required only if the key file is password protected.
- `ccxmli.ssl.recv.protocol_type` must be set to the following values for SSL or TTS, as appropriate:
  - SSLv3, SSLV2 or SSLv23
  - TLSv1 or an empty string that defaults to TLS (Note that the HTTPS protocols SSLv2, SSLv3, and TLSv1 will also work when the protocol type is set to SSLv23.)
- For SSLv2 andSSLv3, `ccpccxml.fips_enabled` must be `false`.

Sending Events

The CCXML session may send an event to an external entity by using the <send> tag, as described in Section 9.2.3 of the CCXML specification. The <send> tag may only specify message content using the namelist attribute; use of inline content is not supported. For an example of how a web application can send an event to a specific CCXML session, see the W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005, Appendix K - Basic HTTP Event I/O Processor, specifically, K.2 and K.3, which contains the information and the example. Table 7 describes how the various attributes of <send> map to the HTTP request:

Table 7: <send> Attributes

Attribute Name	Meaning

targettype	If this attribute is set to basichttp, the message will be routed to the HTTP I/O Processor. **Note:** When this attribute is present, the target CCXML session must belong to the same tenant (i.e. the sender and receiver tenant IDs must match). Otherwise, a error.send.failed event is thrown with the reason property set to Tenant ID mismatch.
target	This is the HTTP URL to which a POST request will be made.
name (or data)	This attribute is required when sending an event to the basichttp event I/O processor. It must contain an ECMAScript expression evaluating to the event name. This will become the value of the name parameter of the HTTP request.
xmlns	This attribute is not supported, because the current platform does not support the sending of inline content.
namelist	This is an optional parameter. If it is present, the variable names and values it contains are mapped to HTTP parameters.
hints	timeout

The basichttp event I/O processor interprets the HTTP response codes as shown in Table 8:

Table 8: Response Codes	
Response Code	Interpretation
2xx	The <send> was successfully accepted by the HTTP server and a `send.successful` event is posted to the session issuing the <send>.
Any other HTTP response code	The <send> was not accepted by the HTTP server and a `error.send.failed` event is posted to the session issuing the <send>.

Creating Sessions

- An external entity can initiate a new CCXML session issuing an HTTP POST to the createsession event I/O processor (as per the W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005).
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP hostname is server.example.com (and the default HTTP port 80 is used), the URI for the event I/O processor is
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP `http:// server.example.com/ccxml/createsession`.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP The uri parameter specifies the URI of the initial CCXML page for the new session.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP The form url-encoded parameters in the body of the HTTP POST request determine the parameters for session creation. The uri parameter determines the initial URI of the initial CCXML page for the new session. The optional eventsource parameter indicates a URI to which events can be returned using the basichttp event I/O processor.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP The eventsource value is exposed to the session as a property of the session.values session variable (that is, session.values.eventsource). The remaining

parameters are used to create additional properties of the session.values object. All parameter values are treated as strings. The property names may be qualified to specify subobjects.

- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP Multiple parameter values (for example, var=val1&var=val2) are not supported and result in a 400 HTTP response.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP The type property of the session.values must be set to createsession. If this property appears in the request body parameters, the specified value is ignored.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP If method, postbody, timeout, maxage and/or maxstale parameters are specified, they have the same effect as the equivalent request URI parameters in SIP-initiated session creation.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP If the create session request can be completed successfully, then a 200 HTTP response code replies to the request. The response body is an application/x-www-form-urlencoded name-value pair list in which the session.id parameter specifies the id of the newly created session.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP specifies the IVR Profile named Profile1 under the tenant named Customer1. Profiles that are created under the Environment tenant (which is the root tenant), use the tenant named Environment, for example, [Environment]. Profile1.
- The access path of the URI used by the createsession I/O event processor is configurable and defaults to /ccxml/createsession. For example, if the CCP

Creating a new CCXML session or sending the event to an existing session

The following configuration parameters specify the address of the existing session:

- `Ccxmli.createsession.recv.path = /ccxml/createsession`
- `Ccxmli.createsession.recv.port = 4892`
- `Ccxmli.basichttp.recv.path = /ccxml/basichttp`
- `Ccxmli.basichttp.recv.port = 4892`

The following example shows how a web page can create a CCXML session in the GVP platform. If you want to send an event to an existing session, use `/ccxml/basichttp` instead, with the `sessionid` parameter containing the real ID of that session.

```
<link rel="stylesheet" type="text/css" href="test.css"/><br>
<form action="http://138.120.84.95:4892/ccxml/createsession" method="post"><br>
  <div><br>
    <label for="uri">uri</label><br>
    <input type="text" name="uri" id="sessionid" value="file:///c:/testpages/
external_simpledialog.ccxml"/><br>
  </div><br>
```

```
  <div><br>
    <label for="eventsource">eventsource</label><br>
    <input type="text" name="eventsource" id="eventsource" value="SOURCE"/><br>
  </div><br>
  <div><br>
    <label for="eventsourcetype">eventsourcetype</label><br>
    <input type="text" name="eventsourcetype" id="eventsourcetype" value="createsession"/><br>
  </div><br>
  <div><br>
    <label for="method">method</label><br>
    <input type="text" name="method" id="method" value="get"/><br>
  </div><br>
  <div><br>
    <label for="timeout">timeout</label><br>
    <input type="text" name="timeout" id="timeout" value="30s"/><br>
  </div><br>
  <div><br>
    <label for="maxage">maxage</label><br>
    <input type="text" name="maxage" id="maxage" value="60"/><br>
  </div><br>
  <div><br>
    <label for="maxstale">maxstale</label><br>
    <input type="text" name="maxstale" id="maxstale" value="30"/><br>
  </div><br>
  <div><br>
    <label for="postbody">postbody</label><br>
    <input type="text" name="postbody" id="postbody" value="n1=v1&n2=v2"/><br>
  </div><br>
  <div><br>
    <label for="name1">name1</label><br>
    <input type="text" name="name1" id="name1" value="value1"/><br>
  </div><br>
  <div><br>
    <label for="name2">name2</label><br>
    <input type="text" name="name2" id="name2" value="value2"/><br>
  </div><br>
  <div><br>
    <label for="complex.name3">complex.name3</label><br>
    <input type="text" name="complex.name3" id="complex.name3" value="value3"/><br>
  </div><br>
  <div><br>
    <label for="gvp-tenant-id ">gvp-tenant-id</label><br>
    <input type="text" name="gvp-tenant-id" id="gvp-tenant-id"
value="CCXMLSimpleDialogLoad"/><br>
  </div><br>
  <div id="submit"><br>
    <input type="submit"/><br>
  </div><br>
</form>
```

Error Handling

If the event properties are not valid, a 400 response is given to the request. Event properties can be considered invalid if, for example, they are not valid ECMAScript variable names, or if multiple values are specified or are conflicting (for example, obj1 and obj1.x are both given a value).

If a fetch timeout is specified in the createsession POST parameters and the timeout expires before the initial CCXML fetch completes, a 408 response is returned to the createsession request.

If fetch, compilation, or initialization of the initial CCXML page URI specified by the uri parameter of a createsession POST request fails for any reason, a 403 response is returned.

If any of the scripts statically referenced by the CCXML page specified by the uri parameter of a createsession POST request cannot be fetched or compiled for any reason, a 403 response is returned.

If any of the scripts statically referenced by the page that is specified by the uri parameter of a createsession POST cannot be fetched within the expected retrieval time, a 408 response is returned.

For this example, assume that the value of the session variable session.ioprocessors["basichttp"] is `http:// ccxml.genesyslab.com/ccxml/ basichttp`. When the following HTTP request is made to this platform:

```
POST http://ccxml.genesyslab.com/ccxml/basichttp?sessionid=ccxmlsession1&<br>
name=basichttp.myevent&eventsource=http://www.example.org/<br>
ccxmlext&<br>
agent=agent12&site=Orlando HTTP/1.0<br>
. . .[other HTTP headers]. . .<br>
. . .[other HTTP headers]. . .</tt>
```

If ccxmlsession1 (the value of the sessionid parameter in the preceding HTTP request) matches the session ID of an existing CCXML session, an event with the name basichttp.myevent is triggered in the session ccxmlsession1. It may be handled as follows:

```
<transition state="'dialogActive'" event="basichttp.*" name="evt"><br>
  <log expr="'Received event'" /><br>
  <log expr="'name=' + evt.name" /><br>
  <log expr="'sourcetype=' + evt.eventsourcetype" /><br>
  <log expr="'source=' + evt.eventsource" /><br>
  <log expr="'agent=' + evt.agent" /><br>
  <log expr="'site=' + evt.site" /><br>
</transition>
```

where:

- **evt.name** would have the value basichttp.myevent
- **evt.eventsourcetype** would have the value basichttp
- **evt.eventsource** would have the value http:// www .example.org/ccxmlext
- **evt.agent** would have the value agent12
- **evt.site** would have the value orlando

The CCP responds with a 204 HTTP response code: `HTTP/1.0 204 No Data`

Example of Sending Events via HTTP

Consider the following CCXML code snippet in the CCXML session with session ID `ccxmlsession2`:

```
<script>
  var agent='agent21';
  var site='miami';
</script>

<send target="'http://travel.genesyslab.com/travelagent'" data="'myevent'"
targettype="'basichttp'" namelist="agent site"/>
```

With this CCXML snippet, the following HTTP GET request is made:

<source lang="xml">GET http://travel.genesyslab.com/travalagent?sessionid=ccxmlsession2&
name=myevent&agent=agent21&site=miami HTTP/1.0 CRLF <source lang="xml">

CCXML Support

CCXML Support

This section lists the extent, and limits, of GVP support of CCXML features.

Events

The **user event** from a VoiceXML dialog must be a simple name-value pair; multi-level objects are not supported.

Tags

<fetch>

If the CCP fails to fetch the URI specified by the next attribute of a <fetch> element for any reason, the error.fetch event is thrown. If the URI scheme is http:, the reason property of the event will be set to Fetch failed: <error code> <reason phrase> where <error code> is the HTTP error code and <reason phrase> is the HTTP response's reason phrase.

<move>

The <move> element cannot move an event source, connection, or dialog to a session executing on a different physical machine.

<join> and <unjoin>

CCXML applications may use the <join> and <unjoin> elements at any time to create a bridge between two connections, conferences, or started dialogs.

Note that:

- Neither the <join> nor the <unjoin> element may be used on dialogs which have not been started with the <dialogstart> element.
- The CCP does not support an early join for an outbound call that is being joined to a conference.}}

Attributes

prepareddialogid

As described in the W3 specification, if the `<dialogstart>` element follows a `<dialogprepare>` element which contains either the `connectionid` or the `conferenceid` attribute, then the value of the `connectionid` or `conferenceid` attribute in the `<dialogstart>` element must match the one appearing in the `<dialogprepare>` element or an `error.dialog.notstarted` event will be thrown.

CCXML does not support emitting AAI in the CDR(See AAI under "Not Supported" below).

<createccxml>

The parameters attribute of the <createccxml> element is not supported. The attribute contains a namelist of CCXML parameters that will be created as properties of the session.values session variable in the new session. For example, if the parameters attribute has a value of foo.bar test, the values of those variables will be assigned to the session.values.foo.bar and session.values.test variables in the new session.

If a CCXML session that was created by another session using <createccxml> exits for any reason other than executing <exit> (for example, it does not catch an error.* event), queuing ccxml.exit to the parent session is not supported.

<createcall>

error.conference.join event
Referencing a dialog that has been prepared but not started in the joinid attribute of <createcall> always results in an error, and thus throwing an error.conference.join event is not supported.

Joining and Codecs
If call legs are being joined implicitly with the <createcall> tag, and the call legs do not provide their codec capabilities either in the initial INVITE to CCP or in the 200 OK response to the initial INVITE sent by CCP, the CCP will not perform transcoding even if a transcoding server has been defined in the mediacontroller.bridge_server option. This is because the CCP requires knowledge of the codec capabilities of each call leg before creating bridges so that it can determine whether a bridging server is needed.

The workaround is to connect each call leg separately using <createcall> and join them together

after they are in the connected state.

Not Supported

- The **dialog.disconnect** event is not supported by the CCP.
- **AAI** (Application-to-Application Information) data received by the CCP from an incoming request URI is not available for use at the application level.
- **HTTPS Cookies** and **Session Cookies** are not supported by the HTTP server interface of the CCP.
- The **<metadata>** tag element is ignored.
- Namespaces specified in the ccxml tag using the **xmlns attribute** are ignored.
  The send element does not support the xmlns attribute to define inline content;
  the namelist attribute is the only mechanism currently supported for providing the message content.
- The **http-equiv Attribute**is not supported by the <meta> element.
- The CCP does not allow configuration of a default set of initial **URI parameters**.
- **UTF Character Sets**: The CCP does not support ECMAScript scripts or CCXML pages that are authored using UTF-16 encoding. The CCP does not support compiling and processing ECMAScript pages using UTF-8 encoding.

Properties

dialogid

While a dialog is sending media to one or more other dialogs over a bridged connection, the dialogid property contains the ID of the sender. When no dialog is sending, the dialogid property contains a null value.

Repeated Parameter Names

- The W3C specification states that parameter names may not be repeated within a request.
- A request with repeated parameter names is considered to be invalid, and should be rejected by the basichttp event I/O processor.
- An HTTP request to I/O processors which contains repeated parameter names is currently not rejected, and does not generate a 400 (Bad Request) response.

Early Media Support

Early Media Support

Early Media refers to the delivery of a media stream to a connection prior to a call being answered, for example, the ringback heard by the calling party on a traditional telephone call before the called party picks up the call.

For a SIP call, early media is transmitted during call setup, i.e., between the time that the INVITE message is issued and the time that the 200 OK response is received.

Early Media has many uses, including:

- Delivery of inband call progress messages, such as announcements.
- Customized ringback tones.
- Prevention of media clipping, which occurs when the user making the call believes that the media session has been established and begins speaking when the call is still being set up, leading to the loss of the first few syllables or words spoken. Use of early media helps to avoid this problem by establishing the media path early.

Announcement Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml xmlns="http://www.w3.org/2002/09/ccxml" version="1.0">
  <!-- Create our ccxml level vars -->
  <var name="in_connectionid" expr="''" />
  <var name="dialogid" expr="''" />
  <var name="timer" expr="''"/>
  <!-- Set our initial state -->
  <var name="currentstate" expr="'state1'" />
  <eventprocessor statevariable="currentstate">
    <!-- Deal with the incoming call -->
    <transition state="state1" event="connection.alerting" name="evt">
      <assign name="in_connectionid" expr="evt.connectionid" />
   <dialogprepare
        src="'file:///usr/local/phoneweb/samples/helloaudio.vxml'"
        dialogid="dialogid"
        connectionid="in_connectionid"/>
      <assign name="currentstate" expr="'state2'"/>
    </transition>
    <transition state="state2" event="dialog.prepared" name="evt">
      <log expr="'Dialog has been prepared'"/>
      <dialogstart prepareddialogid="dialogid"/>
    </transition>
    <transition state="state2" event="send.successful" name="evt">
       <log expr="'send successful'"/>
    </transition>
    <transition state="state2" event="dialog.started" name="evt">
      <log expr="'Dialog has started'"/>
      <send target="in_connectionid" targettype="'x-connection'"
data="'connection.progressing'"/>
    </transition>
    <transition state="state2" event="dialog.exit" name="evt">
      <log expr="'Dialog has terminated; accepting connection'"/>
      <accept connectionid="in_connectionid" />
      <assign name="currentstate" expr="'state3'"/>
```

```
    </transition>
    <transition event="connection.disconnected" name="evt">
      <exit/>
    </transition>
  </eventprocessor>
</ccxml>
```

Notes

In the preceding CCXML example:

- a dialog (`helloaudio.vxml`, a simple VoiceXML page that plays only an audio clip) is established for an incoming call.
- The call is not accepted until the dialog finishes (`helloaudio.vxml` finishes playing the audio file and terminates).
- The call accepting action is handled by the dialog.exit event.

The key logic in this application is the line
`<send target="in_connectionid" targettype="'x-connection'" data="'connection.progressing'"/>`

Sending this instruction to a connection that is in the alerting state triggers the CCP to send a 183 `Session Progress` message to the call originating side, with a valid SDP component so that a media path can be successfully established. The dialog is prepared with connectionid set to in_connectionid.

This simple example illustrates how to write an application that makes use of the Early Media capability. Advanced CCXML users can modify the example to simulate a ringback tone application by doing the following:

- Use the <createcall> tag to create an outbound call.
- Replace the simple helloaudio.vxml with a more sophisticated VoiceXML application, such as one that repeats an audio clip until it is interrupted.
- Terminate the dialog when the outbound call is connected.
- Connect the inbound call and then join the two calls together using <join> or <merge> (as appropriate).

Explaining the use of <createcall>, <join>, <merge> and other CCXML elements is outside the scope of this document.

MSML Spec

MSML Specification Support Notes

- CCXML supports a subset of the MSML specification in `<dialogstart/dialogprepare>`: the Dialog Core package, the Dialog Base package, and the Dialog CPA package.

- Appendix B in the Media Server Deployment Guide contains the Core, Dialog Core, Dialog Base, Dialog Call Progress Analysis MSML specification packages, which also apply to CCXML.
- The Genesys Media Server Deployment Guide Appendix contains the section "MSML Conference Core Package," which does *not* apply to CCXML.

Example

The following example code initiates the CPD detection at the preconnect state, with a five second timeout period:

The following example initiates the CPD detection at the preconnect state, with a five second timeout period.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
        <dialogstart target="conn:xxxx" name="cpd" type="application/moml+xml">
                <cpd initial="preconnect" preconnecttimeout="5s">
                        <cpdtimeout>
                                <send target="source" event="done" namelist="cpd.recfile
cpd.end cpd.result"/>
                        </cpdtimeout>
        </cpd>
        </dialogstart>
</msml>
```

If no media activity was detected during the preconnect state, after five seconds, the CPD completes and sends an event with shadow variables. The actual result should appear as the following, in info.content for the dialog.user.msml event:

```
<?xml version="1.0"?>
<msml version="1.1">
  <event name="msml.dialog.exit" id="conn:xxx/dialog:yyy">
    <name>cpd.recfile</name>
    <value>undefined</value>
    <name>cpd.end</name>
    <value>cpd.completed</value>
    <name>cpd.result</name>
    </value>cpd.preconnect_timeout</value>
  </event>
</msml>
```

CCXML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml xmlns="http://www.w3.org/2002/09/ccxml" version="1.0">

  <!-- Test: SIM_7b: testing dialog.user.msml -->

  <var name="in_connectionid" expr="''" />
  <var name="dialogid" expr="''" />
  <var name="currentstate" expr="'state1'" />
  <var name="mediatypes" expr="'audio'" />
  <script src ="cpd.js"/>
  <eventprocessor statevariable="currentstate">
    <!-- Deal with the incoming call -->
    <transition state="state1" event="connection.alerting" name="evt">
      <assign name="in_connectionid" expr="evt.connectionid" />
      <accept connectionid="in_connectionid" />
```

```
      <assign name="currentstate" expr="'state2'"/>
    </transition>
    <transition state="state2" event="connection.connected" name="evt">
      <assign name="currentstate" expr="'state3'"/>
          <dialogprepare
                        src="'.'"
                        type="'application/vnd.radisys.msml+xml'"
                        dialogid="dialogid" connectionid="in_connectionid"
                        xmlns="urn:ietf:params:xml:ns:msml">
                        <msml version="1.1">
                        <dialogstart name="cpd" type="application/moml+xml">
                        <cpd initial="preconnect" preconnecttimeout="5s">
                        <cpdtimeout>
                        <send target="source" event="done" namelist="cpd.recfile cpd.end
                        cpd.result"/>
                        </cpdtimeout>
                        </cpd>
                        </dialogstart>
                        </msml>
          </dialogprepare>
    </transition>
    <transition state="state3" event="dialog.prepared">
      <assign name="currentstate" expr="'state4'"/>
       <dialogstart prepareddialogid="dialogid" connectionid="in_connectionid"/>
    </transition>
    <transition state="state4" event="dialog.started">
      <assign name="currentstate" expr="'state5'"/>
    </transition>
        <transition state="state5" event="dialog.user.msml">
                <log expr="' content:' + event$.info.content"/>
                  <if cond="event$.info.contenttype == 'application/
vnd.radisys.msml+xml'">
                        <script>
                        var cpdresult = parseCPD(event$.info.content);
                        </script>
                        <log expr="'#PASSED#'"/>
                        <log expr="'result : cpd.recfile:'"/> <log expr="cpdresult.recfile"/>
                        <log expr ="'cpd.end:'"/> <log expr="cpdresult.end"/>
                        <log expr ="'cpd.result:'"/> <log expr="cpdresult.result"/>
                        <exit/>
                  <else/>
                    <log expr="'#FAIL# Incorrent contenttype:' + event$.info.contentype"/>
                  </if>
        </transition>
    <transition state="state5" event="dialog.exit" name="evt">
      <assign name="currentstate" expr="'state6'"/>
      <exit/>
    </transition>
    <transition event="fetch.done">
    </transition>
    <transition event="ccxml.loaded">
    </transition>
    <transition event="*">
        <log expr="'#FAILED# SIM_7b'"/>
        <exit/>
    </transition>
  </eventprocessor>
</ccxml>
```

A line of code early in the above example (<script src ="cpd.js"/>) refers to the Javascript file cpd.js (see below), which performs a single function: parsing the cpd file.

cpd.js

```
function parseCPD(cpdstring)
{
        var result = new Object();
        result.error = 0;
        var nameOpen = 6;
        var valueOpen = 7;

        if(cpdstring.indexOf("<?xml")!=0)
        {
           result.error = 1;
           return result;
        }
        var end, begin;
        end = cpdstring.indexOf("?>");
        if(end <= 0)
        {
                result.error = 2;
                return result;
        }
        begin = cpdstring.indexOf("<msml");
        if(begin < 0 || begin < end)
        {
                result.error = 3;
                return result;
        }
        end = cpdstring.substring(begin).indexOf("</msml>");
        if(end < 0 || end < begin)
        {
                result.error = 4;
                return result;
        }
        begin = cpdstring.substring(begin).indexOf("<event");
        if(begin < 0 || begin > end)
        {
                result.error = 5;
                return result;
        }
        end = cpdstring.substring(begin).indexOf("</event>");
        if(end < 0 )
        {
                result.error = 6;
                return result;
        }
        var beginpair = cpdstring.substring(begin).indexOf("<name");
        if(beginpair < 0 || beginpair > end)
        {
        result.error = 7;
                return result;
        }
        var namevalue=cpdstring.substring(begin+beginpair,end+begin);
        result.name = namevalue;

        var pairs = namevalue.split("</value>");
        if (pairs.length <= 0)
        {
                result.error = 8;
                return result;
        }
        for (var i=0; i<pairs.length; i++)
    {
                begin = pairs[i].indexOf("<name>");
```

```
                end = pairs[i].indexOf("</name>");
                if(end <0 )
                {
                        result.error = pairs[i];
                        return result;
                }
                if( begin <0 )
                {
                        result.error = i+11;
                        return result;
                }
                if( end < begin)
                {
                        result.error = i+12;
                        return result;
                }
                var name = pairs[i].substring(begin+nameOpen,end);
                begin = pairs[i].indexOf("<value>");
                if (begin < 0 || begin < end)
                {
                        result.error = i+13;
                        return result;
                }
                var value = pairs[i].substring(begin+valueOpen);

                if(name == "cpd.recfile")
                        result.recfile = value.toString();
                else if (name=="cpd.end")
                        result.end = value.toString();
                else if (name=="cpd.result")
                        result.result = value.toString();
                else
                        result.error = "wrong cpd Element";
        }
        result.error = "0";
    return result;
}
```