Start developing CCXML applications

Develop call control functionality on any base proprietary telephony system

Skill Level: Introductory

Susan Jackson (sjack@uk.ibm.com)
Developer
IBM

Hannah Parker (hannah.parker@uk.ibm.com)
Software Engineer
IBM

24 Jun 2004

In this tutorial, gain an insight into the new Call Control XML (CCXML) language. This language enhances the telephony industry by allowing development of call control functionality, irrespective of the base proprietary telephony system.

Section 1. About this tutorial

Purpose and audience

This tutorial introduces the Call Control Extensible Markup Language (CCXML), providing information and sample code that will enable users to get started with CCXML, and develop CCXML applications.

This tutorial is aimed at developers who have no previous experience with the CCXML language. It will also be of interest to people with telephony skills who would like to learn how CCXML relates to telephony systems.
The material presented in the tutorial complements the alphaWorks release of the IBM(R) CCXML Interpreter for WebSphere(R) Voice Response for AIX. To complete and make practical use of the examples in this tutorial, download and install the Call Control XML Interpreter (http://www.alphaworks.ibm.com/tech/ccxml) from the IBM alphaWorks site (http://www.alphaworks.ibm.com/).

The authors would like to thank Nathan Brice for his assistance in reviewing this tutorial, and Matthew Whitbourne for his help with testing the example code.

Version information

This tutorial implements the CCXML specification that was used for the IBM CCXML Interpreter alphaWorks release. This specification was based on the latest available working draft CCXML specification from the World Wide Web Consortium (W3C). However, it is important to note that the current W3C specification is somewhat different from the specification that was used in the IBM CCXML Interpreter. This tutorial will clearly state any specification differences that relate directly to the activities that you will be completing.

CCXML is a new language, so new that the W3C specification has not yet been finalised. Therefore, at the moment very few resources are designed to help users write CCXML applications. See Resources for links to additional information that we have identified.

The current specification level is Last Call Working Draft. For further information regarding the level and version of the CCXML specification, see Resources.

Section 2. CCXML overview

What is CCXML?

CCXML is an XML-based markup language that enables developers to add call control capabilities to telephony systems. A CCXML application allows sophisticated multiple-call handling and control, including the ability to place outgoing calls, to answer incoming calls, and to invoke voice applications. It allows for handling of a class of asynchronous events and advanced telephony operations involving substantial amounts of signals, status events, and message-passing. The language
is defined by the W3C (see Resources) and is currently at Last Call Working Draft status.

Previously, the handling of telephone calls has been determined by the proprietary base telephony system. The introduction of the CCXML language into the telephony industry provides users with a consistent and open standard for call control.

Key concepts

Two concepts are key to understanding the CCXML language: **connections and calls** and **the event driven nature of the language**.

- **Connections and calls**: A *connection* is the CCXML concept that represents a telephone call. Therefore, as soon as you have a connection in the system, it relates to a real-world telephone call in the system. A connection has an *input* that enables it to receive a media stream, and an *output* that can be directed at another connection or the telephone network. Every connection has a *state* and, based on external actions, can transition to other states. Some of the many connection states include IDLE, ALERTING, and CONNECTED. All connections start in the IDLE state and return to this state when all actions have been completed. For example, a connection in state CONNECTED will transition to state IDLE when the caller hangs up. (For further information on states, refer to the W3C CCXML specification.)

- **Event driven nature of the language**: *Event driven* means that the order of movement through CCXML applications is determined by events. Events can be handled asynchronously such that they are dealt with in the order in which they are received, without being lost or ignored. Events are described in more detail in Section 3, *CCXML language*.

What are CCXML applications?

Call control functionality for the telephony system is provided through writing and deploying CCXML applications.

A CCXML application can consist of one or more CCXML sessions, each running a separate CCXML script. A CCXML session is a similar concept to a Java thread, whereas a CCXML script is the code that is being run on that session (thread). CCXML scripts are deployed on a Web server and the URL of the script is provided to the configuration of the base telephony system.
Within CCXML applications, it is possible to perform sophisticated multiple-call handling and control. Consequently, CCXML applications allow for handling of asynchronous events and advanced telephony operations, which can involve substantial amounts of signals, status events, and message-passing. One example of advanced telephony operation is the use of Session Initiation Protocol (SIP) header information.

How does CCXML relate to VoiceXML?

CCXML and VoiceXML are separate and distinct languages. The languages are mutually independent, such that a VoiceXML application doesn't need to be invoked through a CCXML application and CCXML applications are not required to include VoiceXML applications.

However, the CCXML language has been designed to complement and to integrate with VoiceXML systems. VoiceXML dialogs can be invoked from CCXML applications, thus enabling the caller to interact with the Interactive Voice Response (IVR) system.

CCXML can provide call control capabilities that the VoiceXML specification does not. CCXML enables each active telephone call to have a dedicated VoiceXML interpreter. Previously, when a telephone call was transferred within VoiceXML, the second leg of the call did not have its own dedicated VoiceXML interpreter. Without the second VoiceXML interpreter, the functionality of the application was limited.

You can find more detail on the relationship between CCXML and VoiceXML in the CCXML specification (see Resources).

What is the CCXML Interpreter?

The CCXML Interpreter is the Java code that parses and interprets the information contained within CCXML scripts, and thus controls the execution of the script.

Essentially, the Interpreter translates common CCXML by sitting between the CCXML application and the proprietary base telephony platform. The Interpreter is a layer above the platform that tells the platform and telephony system what to do, based upon the information in the CCXML document.

IBM recently released an alpha version of the CCXML Interpreter for WebSphere Voice Response, which can be downloaded from the alphaWorks Web site (see Resources).
Section 3. CCXML language

Elements

The CCXML language is a flavour of XML that provides specific call control functionality. As with all variants of XML, CCXML consists of a specialised set of elements:

- An element consists of an opening and closing tag, with content between the two tags. Attribute information can be located in the content, or in the opening tag. For example, `<accept connectionid="evt.connectionid"/>`.

- An empty element is simply the opening and closing tags without content between them. For example, `<eventprocessor/></eventprocessor>`.

For a full list of CCXML elements, refer to the documentation that is included in the alphaWorks release of the IBM CCXML Interpreter for WebSphere Voice Response for AIX.

Events

As mentioned in CCXML overview, the CCXML language is driven by events. Now that we have introduced the concept of event driven, we can explain what an event is:

- An event can be a message that is sent from the telephony system to a CCXML application, thus driving the CCXML application to perform the appropriate call control function.

- An event can also be generated by a user-written CCXML application, by the CCXML Interpreter, or by a Voice application that has been invoked by the CCXML Interpreter.

Event handling
By "handling", we mean that events can be both sent and received. Event handling is achieved by using the `<eventprocessor>` and `<transition>` elements. Events that are waiting to be processed are stored in a queue. The `<eventprocessor>` element takes the first available event from this queue.

The `<eventprocessor>` element contains one or more `<transition>` elements. Each `<transition>` element is coded to recognise a specific event, and contains information appropriate to that event. When an event is processed by a `<transition>` element, the application runs the code that is contained between the opening and closing tags.

The following is a simple example that demonstrates how to use the `<eventprocessor>` and `<transition>` elements:

```xml
<eventprocessor>
  <!-- Transition for event A -->
  <transition>
    some code to be run for event A
  </transition>
  <!-- Transition for event B -->
  <transition>
    some code to be run for event B
  </transition>
</eventprocessor>
```

In this example, after the first `</transition>` tag is reached, the application will only move on to the second `<transition>` element if event B is the next event in the queue.

**ECMAScript**

ECMAScript, similar to JavaScript, is the scripting language used within CCXML.

The ECMAScript specification that is used in CCXML is an industry-standard programming language based on ECMAScript Compact Profile (ES-CP, also known as ECMA-327). ES-CP is tailored to resource-constrained devices and is used by CCXML to maximize efficiency (see Resources).

**ECMA variables** are used to store information about events. This information is available to a `<transition>` that is processing an event. Examples and further information about ECMA variables are provided in Event attributes.
Section 4. A simple CCXML application

Overview

The following sections of this tutorial go through the steps required to create a CCXML script. To better understand the fundamental principles involved in creating CCXML applications, you can create these scripts and run them with the alphaWorks release of the CCXML Interpreter. However, this isn't essential, and you will still develop an understanding of the central ideas by just reading through the sections.

The first example, which you should call "basic.ccxml", will answer a call and play a VoiceXML dialog to the caller.

The <xml> and <ccxml> elements

The first element in any CCXML script is the <xml> element, which specifies the version and encoding to be used for the script:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

The CCXML code then continues after this declaration. The entire CCXML script is enclosed by the <ccxml> element, which is the parent element of a CCXML script.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<ccxml version="1.0">

<eventprocessor>

<!-- Accepts an incoming call -->
<transition event="connection.ALERTING" name="evt">
  <accept connectionid="evt.connectionid"/>
</transition>

<!-- Starts VoiceXML dialog on a separate thread -->
<transition event="connection.CONNECTED" name="evt">
  <dialogstart src="welcome.vxml" type="application/xml+vxml"/>
</transition>

<!-- Disconnects the call when dialog finishes -->
```
You can place some initialisation code immediately after the opening `<ccxml>` element. This will be executed in script order when the session starts. When this initialisation code has been executed, the execution continues with the `<eventprocessor>` element.

### The `<eventprocessor>` and `<transition>` elements

Each CCXML script requires an `<eventprocessor>` element, which controls the execution of the script. The `<eventprocessor>` is also responsible for handling the event queue; it removes the first event from the queue, and selects which transition to execute.

The `<transition>` elements define which other CCXML elements are executed as a result of the received event. The `<transition>` elements are all contained within the `<eventprocessor>` element.

The example below illustrates how to handle an incoming call. The first event to be received from an incoming call is a `connection.ALERTING` event. This event is generated when a connection transitions to the alerting state. When the event is received, the call is accepted by using the `<accept>` element.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0">
  <eventprocessor>
    <!-- Accepts an incoming call -->
    <transition event="connection.ALERTING" name="evt">
      <accept connectionid="evt.connectionid"/>
    </transition>
  </eventprocessor>
</ccxml>
```

The `event` attribute of the `<transition>` element indicates which events will cause the `<transition>` to be executed. In the case above, the `<transition>` is selected when a `connection.ALERTING` event is received. The `name` attribute within the `<transition>` element is an ECMA variable, into which the event's
details will be saved.

When inside the <transition> element, the child elements are executed in script order, therefore the <accept> element is executed first. This element causes the telephony layer to connect the call. The attribute connectionid must contain the identifier of the connection on which the incoming call has been signaled. In this case, the signal is contained in the form of a connection.ALERTING event. You can obtain information about the connection from this event.

Event attributes

All events contain attributes, and you can find a full listing of these attributes in the W3C specification. For example, the connection.ALERTING event contains information about the name of the event, the connectionid, the protocol, an info attribute, and a connection attribute. The connection attribute provides access to the connection object, which contains more information about the connection.

To access the information, use the following notation:

- evt.name
- evt.connectionid
- evt.protocol

where evt is the name that is given to the event in the <transition> element.

In this way, the connectionid is obtained from the connection.ALERTING event. This ID is then used to tell the <accept> element which call to accept.

```xml
<!-- Accepts an incoming call -->
<transition event="connection.ALERTING" name="evt">
  <accept connectionid="evt.connectionid"/>
</transition>
```

The <dialogstart> element

If the <accept> element completes successfully, then the next event to be generated is a connection.CONNECTED event. This event can be handled in a similar way, but in this <transition> a dialog is about to be started on the call.
The same notation is used here to handle the connection.CONNECTED event. The <dialogstart> element is used inside the <transition> element. The <dialogstart> element results in a dialog being started, and the dialog being associated with the specified connection.

You specify a URI reference to the initial dialog document by using the src attribute. This is the dialog document that will be loaded and executed. The src attribute must be equal to an ECMAScript expression which returns a character string. The single quotes around "welcome.vxml" indicate that it is a string. If the single quotes were omitted, then the "welcome.vxml" is interpreted to be a variable and an attempt is made to evaluate this variable, resulting in errors. It is possible to specify that a variable contain the URI of the document.

The new dialog executes on a separate logical thread, and therefore the execution of this dialog does not block the processing of further events by the CCXML session.

The type attribute has also been specified in the <dialogstart> element. This is the MIME type of the document; in this case, you are using a VoiceXML document and have specified the appropriate MIME type.

The <disconnect> element

When the dialog finishes, the CCXML session receives a dialog.exit event. At this point the call can be disconnected.

In the <disconnect> element, it is necessary to specify the connectionid for the connection that is to be disconnected. This information is retrieved from the dialog.exit event to ensure that the call on which the dialog has just finished is the one to be disconnected.

Full code for basic.xml

All the elements in the first example have now been discussed. The CCXML script
that you have created answers a call, plays a dialog to the caller, and then
disconnects the call when the dialog finishes.

Here is the complete code:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<ccxml version="1.0">
    <eventprocessor>
        <!-- Accepts an incoming call -->
        <transition event="connection.ALERTING" name="evt">
            <accept connectionid="evt.connectionid"/>
        </transition>

        <!-- Starts VoiceXML dialog on a separate thread -->
        <transition event="connection.CONNECTED" name="evt">
            <dialogstart src="welcome.vxml" type="application/xml+vxml"/>
        </transition>

        <!-- Disconnects the call when dialog finishes -->
        <transition event="dialog.exit" name="evt">
            <disconnect connectionid="evt.connectionid"/>
        </transition>
    </eventprocessor>
</ccxml>
```

Section 5. Adding voting capability

Overview

The previous example introduced the basic structure of a CCXML script.

This next example builds on the first to include some logic and to provide the
capability to count the number of calls that are made to different numbers. The example is going to keep track of the number of calls that are made to 1001, 1002, and any other number.

Variables

Within a CCXML script, you'll find two different variable scopes -- global scope and
transition scope:

**Global scope:** Variables in the global scope must be declared after the `<ccxml>` element but before the `<eventprocessor>` element. These variables can be referenced or changed from any point within the CCXML script. They exist while the script is loaded and are only visible within that script.

**Transition scope:** Variables in this scope are declared within a `<transition>` and are only available within, and for the lifetime of, that transition.

For this example, three global variables must be declared in order to keep track of the number of calls made to the different numbers.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0">

    <var name="count1001" expr="0"/>
    <var name="count1002" expr="0"/>
    <var name="countother" expr="0"/>

<eventprocessor>
    <!-- Accepts an incoming call -->
    <transition event="connection.ALERTING" name="evt">
        <accept connectionid="evt.connectionid"/>
    </transition>
    <!-- Starts VoiceXML dialog on a separate thread -->
    <transition event="connection.CONNECTED" name="evt">
        <dialogstart src='"welcome.vxml"' type="application/xml+vxml"/>
    </transition>
    <!-- Disconnects the call when dialog finishes -->
    <transition event="dialog.exit" name="evt">
        <disconnect connectionid="evt.connectionid"/>
    </transition>
</eventprocessor>

</ccxml>
```

The `name` attribute is used to specify the name of the variable.

The `expr` attribute is the initial value of this variable. `expr` can be any valid ECMAScript expression, and is an optional attribute.

In the example, the values are initialised to zero.

**Error handling**
As the CCXML script becomes more complicated, it can become trickier to detect mistakes. If an error is made in the CCXML code, this results in an error event. In the script, you should include a wildcard \(<transition>\) that's designed to handle these events.

The CCXML Interpreter selects, in script order, the first \(<transition>\) that satisfies a particular event. Place the wildcard \(<transition>\) at the end of the CCXML script, so that it doesn't interrupt expected script flow.

The final \(<transition>\) in the example should therefore be:

```xml
<transition event="error.*" name="evt">
  <log expr="evt.name + ' occurred, with error condition ' + evt.error"/>
</transition>
```

A wildcard in the form of an asterisk (*) has been used in the event attribute, so that the \(<transition>\) element will catch any event that starts with "error.". For example, error.ccxml is generated when the CCXML script contains a semantic error.

The \(<log>\) element results in output to a file called ccxml_app_log, which can be found in the var/dirTalk/DTBE/native/aix directory for the IBM alphaWorks release. The expr attribute of the \(<log>\) element is an ECMAScript expression that evaluates to a string. This allows variables to be evaluated and the results logged out. In the above example, the error attribute of the event and the event name are both logged out.

Logic elements

To count the number of calls, modify the connection.CONNECTED \(<transition>\) to include logic elements.

Note that the welcome.vxml dialog is played to all callers, regardless of the number dialed. This functionality is provided by the \(<dialogstart>\) element, as discussed in The \(<dialogstart>\) element. Since the dialog is executed on a separate thread, execution continues through the CCXML script using logic to count the number of calls.

The following \(<transition>\) shows how logic is used to count the number of calls made to different numbers.

```xml
<transition event="connection.CONNECTED" name="evt">
```
Every connection event contains a connection object and that object contains a number of different parameters; one of these parameters is the local number (the number dialed by the caller). This value is obtained by evaluating `evt.connection.local`, where `evt` is the name given to the event in the `<transition>`. In this example, `evt.connection.local` is used to determine which number is being called, and thus which variable to increment.

Three global variables have already been created using `<var>`. The value of these variables can now be changed by using the `<assign>` element, which is very similar to the `<var>` element. The `name` attribute specifies the name of the variable, and the `expr` attribute determines the new value of the variable.

The `<if>` and `<elseif>` elements each take a `cond` attribute. This is an ECMAScript expression which evaluates to true or false, as follows:

- The `<if>` evaluates to true when the value of `evt.connection.local` is 1001. This results in execution of all elements that are directly after the `<if>` and before an `<elseif>`, an `<else>`, or an `</if>`. In the example above, `<assign>` is the first element to be executed, which adds 1 to the global variable `count1001`. The new value of `count1001` is also logged.

- The `<elseif>` is tested when `evt.connection.local` doesn’t evaluate to 1001. If the number dialed is 1002, then the statements contained within the `<elseif>` tags are executed.

- The `<else>` element is executed when both the `<if>` and `<elseif>` conditions evaluate to false. In this example, 1 is added to the `countother` variable, and the number that was dialed is logged.

This logic provides the ability to count the number of calls made to different numbers, and hence provides the functionality needed for a simple voting application. All callers hear the same dialog, but the dialed number is counted.
Note: The implementation of the `<if>`, `<elseif>`, and `<else>` elements provided in the IBM alphaWorks release differs from the W3C Last Call Working Draft specification.

Event catching

The error handling topic introduced the concept of wildcard transitions to catch error events. The same principle can be used to catch any additional events for which there is no specified `<transition>.

By entering a `<transition>` at the bottom of the script, with a wildcard as the event name, any events that are not caught by specific `<transition>`s will be caught. You can then use some logging in this `<transition>` to detect the event. This is useful when developing new CCXML scripts to ensure that all events have been handled if required.

The following `<transition>` results in logging for each unhandled event:

```
<transition event="*" name="evt">
  <log expr="evt.name + ' event was caught'"/>
</transition>
```

You can put empty transitions into the CCXML script to prevent certain events from triggering the wildcard `<transition>`. The following events were all generated by the voting example:

```
<transition event="ccxml.loaded"/>
<transition event="dialog.started"/>
<transition event="connection.DISCONNECTED"/>
```

- The `ccxml.loaded` event is generated when a new CCXML session is started and the CCXML script is loaded. It provides the opportunity to start execution without the need for a call.
- The `dialog.started` event is generated when a `<dialogstart>` element completes successfully.
- The `connection.DISCONNECTED` event is generated after a successful `<disconnect>`, or to provide notification that the call has been disconnected by the platform.

With these transitions in place in the `voting.ccxml` example, any other unexpected events will be caught and logged.
The complete code

This section has introduced CCXML variables, error handling, event catching, and logic elements. The voting.ccxml application counts the number of calls made to different telephone numbers.

Here is the complete code:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0">
  <!-- Global variable to store the number of calls to 1001 -->
  <var name="count1001" expr="0"/>
  <var name="count1002" expr="0"/>
  <var name="countother" expr="0"/>
  <eventprocessor>
    <transition event="ccxml.loaded"/>
    <!-- Accepts an incoming call -->
    <transition event="connection.ALERTING" name="evt">
      <accept connectionid="evt.connectionid"/>
    </transition>
    <!-- Starts VoiceXML dialog on a separate thread -->
    <!-- Counts the number of times 1001 and 1002 are dialed -->
    <!-- Logs the dialed number -->
    <transition event="connection.CONNECTED" name="evt">
      <dialogstart src="welcome.vxml" type="application/xml+vxml"/>
      <if cond="evt.connection.local=='1001'">
        <assign name="count1001" expr="count1001 + 1"/>
        <log expr="Call number ' +count1001+ ' to 1001'"/>
      </else if cond="evt.connection.local=='1002'">
        <assign name="count1002" expr="count1002 + 1"/>
        <log expr="Call number ' +count1002+ ' to 1002'"/>
      </else if>
      <else>
        <assign name="countother" expr="countother + 1"/>
        <log expr="Call number ' +countother+ ' to another number - ' +evt.connection.local"/>
      </else>
    </if>
  </transition>
  <transition event="dialog.started"/>
  <!-- Disconnects the call when dialog finishes -->
  <transition event="dialog.exit" name="evt">
    <disconnect connectionid="evt.connectionid"/>
  </transition>
  <transition event="connection.DISCONNECTED"/>
  <!-- Catches any errors -->
  <transition event="error.*" name="evt">
    <log expr="evt.name + ' occurred, with error condition ' + evt.error"/>
  </transition>
</ccxml>
```
Section 6. Dynamic script refreshing

The exercise

The final example modifies the two previous CCXML files, basic.ccxml and voting.ccxml. The changes result in the file being dynamically updated every 12 hours. A typical real-world scenario for this application would be when votes are only counted from 8am to 8pm. Outside of these times, callers would not be allowed to place votes, and hence a different CCXML script must be used.

The <send>, <fetch>, and <goto> elements are introduced in this section.

Setting up a timer

The first step is to use the functionality of the <send> element's delay attribute to set up a timer. The <send> element is used to send messages to another CCXML Interpreter or to an external event processor. For this exercise, an event is sent to the current session. The timer must be set as soon as the current script is loaded, therefore the send must occur in the ccxml.loaded transition.

Modify the code in basic.ccxml to contain the following:

```xml
<transition event="ccxml.loaded" name="evt">
  <send target="evt.sessionid" data="'refresh'" delay="43200s"/>
</transition>
```

- The target determines where the event is sent, and in this example, identifies the current session.
The data attribute specifies the name of the event that is sent.

The delay attribute is used to specify a 12-hour period. The time format is CSS2 format.

A timer is now set and ready. The <send> element returns immediately with a send.SEND_SUCCESSFUL event. However, the refresh event is not dispatched until the timer interval elapses.

To handle the send.SEND_SUCCESSFUL event, add another transition. This can simply be an empty <transition> element:

```
<transition event="send.SEND_SUCCESSFUL" name="evt"/>
```

Changing the script

To handle the refresh event, add a new <transition> to the script:

```
<transition event="refresh" name="evt">
  <fetch next="'voting.ccxml'"/>
</transition>
```

The refresh event is sent when the timer expires. When the event arrives, the above <transition> is executed. At this point, the CCXML script that is being executed must be updated so that callers can start to place votes.

Changing the script is a two-stage process:

- Fetch the new script
- Enable execution of the new script

**Fetch the new script**

You need to issue a <fetch> request. This element finds, loads, and parses the new CCXML script.

The next attribute of the <fetch> element uses an ECMAScript expression to specify the URI of the script to be fetched. In this example, voting.ccxml is fetched. Note that the voting.ccxml script is fetched asynchronously, so execution continues to the next event in the queue.
When the `<fetch>` has completed successfully, a `fetch.done` event is added to the queue.

Enable execution of the new script

Now, perform a `<goto>`. The `<goto>` can be executed after the `<fetch>` element completes. The following `<transition>` is required:

```xml
<transition event="fetch.done" name="evt">
  <goto fetchid="evt.fetchid"/>
</transition>
```

The `<goto>` transfers control of the session to the script that is identified by the `fetchid`. The target script replaces the current script, and all event sources that are associated with the current script are inherited by the target script. Execution of the current script then ends.

Changes required in the voting application

Change the `voting.ccxml` application to have similar functionality to the modified `basic.ccxml` application. In `voting.ccxml`, change the `ccxml.loaded` `<transition>` so that it is the same as the `ccxml.loaded` `<transition>` in `basic.ccxml`. Then add the following two transitions:

```xml
<transition event="refresh" name="evt">
  <fetch next="basic.ccxml"/>
</transition>

<transition event="fetch.done" name="evt">
  <goto fetchid="evt.fetchid"/>
</transition>
```

When the modified `basic.ccxml` and `voting.ccxml` are run, callers hear the same dialog played at any time of day. However, the votes of callers are only counted for 12 hours a day.

In this example, you have not specified in which 12 hours votes are counted. The 12-hour period commences when a `ccxml.loaded` event is received.

The complete code: modified `basic.ccxml`

This section has introduced the `<send>`, `<fetch>`, and `<goto>` elements. This provides the capability to dynamically change the CCXML document that is being
executed.

The following is the complete code for basic.ccxml, incorporating the changes made in this section:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0">
  <eventprocessor>
    <!-- Initiates a send event with 12-hour delay -->
    <transition event="ccxml.loaded" name="evt">
      <send target="evt.sessionid" data="'refresh'" delay="43200s"/>
    </transition>
    <!-- Accepts an incoming call -->
    <transition event="connection.ALERTING" name="evt">
      <accept connectionid="evt.connectionid"/>
    </transition>
    <!-- Starts VoiceXML dialog on a separate thread -->
    <transition event="connection.CONNECTED" name="evt">
      <dialogstart src="'welcome.vxml'" type="application/xml+vxml"/>
    </transition>
    <!-- Disconnects the call when dialog finishes -->
    <transition event="dialog.exit" name="evt">
      <disconnect connectionid="evt.connectionid"/>
    </transition>
    <!-- Fetches the new document when the sent event is received -->
    <transition event="refresh" name="evt">
      <fetch next="'fetch.ccxml'"/>
    </transition>
    <!-- Replaces the current document with the new one -->
    <transition event="fetch.done" name="evt">
      <goto fetchid="evt.fetchid"/>
    </transition>
  </eventprocessor>
</ccxml>
```

The complete code: modified voting.ccxml

The following is the complete code for voting.ccxml, incorporating the changes made in this section:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0">
  <!-- Global variable to store the number of calls to 1001 -->
</ccxml>
```
<!-- Initiates a send event with 12-hour delay -->
<transition event="ccxml.loaded" name="evt">
  <send target="evt.sessionid" data="'refresh'" delay="43200s"/>
</transition>

<!-- Accepts an incoming call -->
<transition event="connection.ALERTING" name="evt">
  <accept connectionid="evt.connectionid"/>
</transition>

<!-- Starts VoiceXML dialog on a separate thread -->
<!-- Counts the number of times 1001 and 1002 are dialed -->
<!-- Logs the dialed number -->
<transition event="connection.CONNECTED" name="evt">
  <if cond="evt.connection.local=='1001'">
    <assign name="count1001" expr="count1001 + 1"/>
    <log expr="'Call number ' +count1001+ ' to 1001'"/>
  </if>
  <if cond="evt.connection.local=='1002'">
    <assign name="count1002" expr="count1002 + 1"/>
    <log expr="'Call number ' +count1002+ ' to 1002'"/>
  </if>
  <else>
    <assign name="countother" expr="countother + 1"/>
    <log expr="'Call number ' +countother+ ' to another number - ',
      +evt.connection.local"/>
  </else>
</transition>

<!-- Disconnects the call when dialog finishes -->
<transition event="dialog.exit" name="evt">
  <disconnect connectionid="evt.connectionid"/>
</transition>

<!-- Fetches the new document when the sent event is received -->
<transition event="refresh" name="evt">
  <fetch next="basic.ccxml"/>
</transition>

<!-- Replaces the current document with the new one -->
<transition event="fetch.done" name="evt">
  <goto fetchid="evt.fetchid"/>
</transition>

<!-- Catches any errors -->
<transition event="error.*" name="evt">
  <log expr="evt.name + ' occurred, with error condition ' + evt.error"/>
</transition>

<!-- Catches any missed events -->
<transition event="*" name="evt"/>
Section 7. Summary

What you have done

The CCXML language is a specialised set of XML elements that offer call control capability to telephony systems.

After a brief introduction to the CCXML language, you learned how to use CCXML to create a basic.ccxml application, and then extend that into a CCXML voting application. To introduce you to more of the CCXML language, we showed you how to modify and extend two simple applications in order to demonstrate dynamic script loading capabilities.

This tutorial has presented you with the basics of CCXML, to enable you to start writing simple CCXML applications.
Resources

Learn

• Read the W3C CCXML specification, Voice Browser Call Control: CCXML Version 1.0, which describes the CCXML language. Note that the content of any Working Draft specification is liable to change (http://www.w3.org/TR/ccxml/).

• Read the VoiceXML 2.0 specification, which specifies the Voice Extensible Markup Language. VoiceXML dialogs can be invoked by CCXML applications (http://www.w3.org/TR/2004/PR-voicexml20-20040203/).

• For additional background on VoiceXML, read these developerWorks tutorials:
   • "Deep into VoiceXML, Part 1" by Harsha Srivatsa (October 2002)
   • "Deep into VoiceXML, Part 2" by Harsha Srivatsa (February 2003)

• Check out Standard ECMA-327 which defines the ECMAScript Compact Profile (ES-CP) scripting language (http://www.ecma-international.org/publications/standards/Ecma-327.htm). ECMAScript expressions are used within CCXML.

• Read VoiceXML: Introduction to Developing Speech Applications by James A. Larson.

• Browse for books on these and other technical topics.

• Find more XML resources on the developerWorks XML zone.

• Finally, find out how you can become an IBM Certified Developer in XML and related technologies.

Get products and technologies

• Download and install the Call Control XML Interpreter from the IBM alphaWorks site (http://www.alphaworks.ibm.com/tech/ccxml). This tool can help you to complete and make practical use of the examples in this tutorial.

About the authors

Susan Jackson
Susan Jackson is a developer for IBM WebSphere Voice, based at IBM Hursley (UK). She is a member of the IBM CCXML team that delivered the alphaWorks release of the IBM CCXML Interpreter. Susan is a recent graduate hire at IBM Hursley. Susan,
MSci Hons, read Theoretical Physics at Durham University.

Hannah Parker
Hannah Parker is a member of the WebSphere Platform Messaging Test organisation at IBM Hursley. Previously, she worked as an Information Developer for IBM WebSphere Voice, responsible for providing the documentation for the IBM CCXML Interpreter alphaWorks delivery. Hannah is a recent graduate hire at IBM Hursley. Hannah, BSc Hons, read Cognitive Science at Exeter University.