Using a CVP VoiceXML application to implement a logical shadow queue for ICM

Introduction

When calls are queuing in ICM, situations can arise in which greater visibility of the queue contents is required than can be provided by the default real-time queuing reports, for example:

- For queue position tracking and announcement
- To provide real-time data on queue status with immediate update
- To generate events as calls are added to and removed from queue
- To view caller information for calls in queue

One method that can be employed to address these requirements is to implement a virtual queue on the IVR platform that provides the queuing treatment. This virtual queue can take different forms but typically comprises a queue entry table in a common data repository into which the IVR application can send indications when calls commence queuing treatment and when calls are disconnected, either when the caller abandons or is transferred to an agent.

Other applications can then query the queue entry table for reporting purposes, for example, to retrieve calls in queue by type, longest/average waiting now, recent queue performance statistics, etc.

Mechanism

In the case of using the Customer Voice Portal (CVP) this approach is most easily implemented using a CVP Studio application to provide queuing treatment and a common database accessible to each CVP VoiceXML Server. The basic mechanism can be described as follows:

- Call is queued in the ICM
- The ICM script invokes a Run External Script node to pass control to the CVP Studio application that will provide the queuing treatment
- The name of the virtual queue is passed as incoming data to the CVP Studio application in the normal way using the ToExtVXML ECC variables. This queue name is simply a user-defined string to tag calls that are subject to the same queuing treatment within the ICM and should be considered logically in the same way as a tagging a group of people standing in line at a bus stop waiting for transport to a common destination. It doesn't matter how many different physical bus services could fulfill the journey; the key issue is that all the entries in the same logical queue can be fulfilled by the same thing. Commonly, this logical queue name would correspond to a call type in the ICM.
- The CVP Studio application either invokes the On Start Call custom Java class or executes a standard Studio Database node to insert a new entry into the queue table, specifying the logical queue name. A unique queue entry ID is returned and stored in session data.
- Normal queuing treatment proceeds according to the CVP Studio application flow with music and announcements presented to the caller as required.
- When the IVR call leg terminates on caller hangup or transfer to an agent, the On End Call custom Java class configured for this CVP application is invoked. This acts as an exit-handler and using the unique queue entry ID returned from the insert operation, it either performs an update on the database entry to mark it as ended or deletes it from the table depending on whether there is a requirement to subsequently access data for completed queue entries.

Paul Tindall, Cisco, Version 1.0, 10 March 2011

- The On End Call class will be executed regardless of how the application terminates with the exception of abnormal termination of the server software or hardware failure so even if the CVP application instance times-out, the On End Call class will tidy-up the database entry. To overcome the problem with server failure and the resultant orphaned queue entries the mechanism can be enhanced, for example, by updating the queue entry intermittently with a keep-alive time to flag the entry as still valid. Alternatively, the database can be modified externally to reset the table, purge or mark-closed orphaned entries following a system failure. In fact, using the intermittent update approach it's possible to forcibly invalidate the entire virtual queue and have it rebuilt automatically by those calls still queuing.

Implementation

The following section lists the steps involved in implementing the basic mechanism, in this case with MySQL as the database. Of course, any data repository that is accessible across all the CVP servers could be used, including other SQL databases, NoSQL, distributed data-grids etc but by far the simplest approach is to use JDBC and SQL onto one of the common database types as this doesn't require any additional CVP custom script elements and also fits well with standard reporting tools.

1 Database table

Create a database table for queue entries typically comprising at least the following fields. Configure the database as a JNDI datasource in context.xml on the CVP VoiceXML Server Tomcat instance.

starttime	Datetime this queue entry inserted
qname	Logical name of queue for this call
qentryid	Unique queue entry identifier (primary key)
callid	CVP call identifier, used for data joins if required for reporting
endtime	Datetime this queue entry ended

MySQL example:

```
CREATE TABLE `qentries` (
  `starttime` timestamp NOT NULL default '0000-00-00 00:00:00',
  `qname` varchar(32) NOT NULL default '',
  `qentryid` int(10) unsigned NOT NULL auto_increment,
  `callid` varchar(32) NOT NULL default '',
  `endtime` datetime default NULL,
  PRIMARY KEY  (`qentryid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

2 Adding a queue entry

Pass control from the ICM script to the CVP Studio queuing treatment application specifying the logical name of the queue as one of the data items passed from ICM to CVP. Use one of the following two methods to add a new queue entry into the database.

- Use a standard Database script element to invoke a stored procedure to perform a database INSERT operation and return the unique queue entry ID to the script. While it isn't absolutely essential to use a stored procedure, that method does allow multiple operations to be performed via a single script element if, for example, the unique ID is allocated by the database and would need to be retrieved using a separate SELECT operation. Use of database stored procedures can make the CVP Studio script simpler and therefore less prone to introduction of errors.
- Configure the On Start Call custom Java class in the CVP Studio application Endpoint Settings to do this automatically when the application is invoked. This approach has the advantage that nothing related to queue entry handling is added explicitly to the queuing treatment script.

An example of an On Start Call class is shown below. In this example, the logical queue name is read from the session data item "qname" and the new entry ID is stored in session data "qid". Note that data is passed from ICM to CVP Studio applications in the form of key-value pairs with each data item pair stored as CVP application session data in the name of the key. The database is specified using a JNDI datasource, in this case "jdbc/test" which should already have been configured. The custom class is located in C:\Cisco\CVP\VXMLServer\common\lib or \classes as appropriate.

```
package com.cisco.cvp.vxml;

import com.audium.server.AudiumException;
import com.audium.server.proxy.StartCallInterface;
import com.audium.server.session.CallStartAPI;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

/**
 * The On Start Call Class is called when a call is first received before the
 * call flow begins.
 */
public class Qstart implements StartCallInterface {

    public void onStartCall(CallStartAPI callStartAPI) throws AudiumException {
        try {
            Context envCtx = (Context) new InitialContext().lookup("java:comp/env");
            DataSource ds = (DataSource) envCtx.lookup("jdbc/test");
            Connection conn = ds.getConnection();

            String qname = (String) callStartAPI.getSessionData("qname");
            String callid = (String) callStartAPI.getSessionData("callid");

            Statement s = conn.createStatement();
            s.executeUpdate("insert into qentries values (NULL,'" + qname + "',NULL,'" + callid + "',NULL)");
            ResultSet rs = s.executeQuery("select LAST_INSERT_ID() qentry");

            rs.first();
            int qid = rs.getInt("qentry");
            callStartAPI.setSessionData("qid", Integer.toString(qid));

            conn.close();

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

3 Removing a queue entry

Configure the On End Call custom Java class in the CVP Studio application Endpoint Settings. An example of an On End Call class is shown below. In this example, the queue entry is retrieved from session data "qid" saved from the insert operation and the queuing end time is updated in the database entry for that queue ID. The custom class is located in C:\Cisco\CVP\VXMLServer\common\lib or \classes as appropriate.

```
package com.cisco.cvp.vxml;

import com.audium.server.AudiumException;
import com.audium.server.proxy.EndCallInterface;
import com.audium.server.session.CallEndAPI;

import java.sql.Connection;
import java.sql.Statement;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

/**
 * The On End Call Class is called if the caller hung up, the application hung
 * up on the caller, the session was invalidated, or the session timed out on its
 * own (due to some error).
 */

public class Qend implements EndCallInterface {

    public void onEndCall(CallEndAPI callEndAPI) throws AudiumException {
        try {

            Context envCtx = (Context) new InitialContext().lookup("java:comp/env");
            DataSource ds = (DataSource) envCtx.lookup("jdbc/test");
            Connection conn = ds.getConnection();

            String qid = (String) callEndAPI.getSessionData("qid");
            Statement s = conn.createStatement();
            s.executeUpdate("update qentries set endtime=NOW() where qentryid=" + qid);
            conn.close();

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Additional considerations

The developer/integrator should be mindful of the following considerations when adopting this approach for shadowing call queues.

- It is wise to use a default queue name if none is passed from the ICM. In this way, calls in queue can still be made visible in the queue table under a default or unaccounted category. The CVP application name is readily available from CVP application internal data and is certainly one possibility.

- Tomcat database connection pool parameters should be configured to handle the required throughput and the database connectivity should be tested under load as in the case of any backend database integration.
- If entries are not automatically deleted from the database then the table size should be managed externally, and data purged at intervals as necessary. As an alternative, it is also feasible to include some additional database cleanup operations in the On Call End class for execution on an intermittent scheduled basis.
- A mechanism should be implemented to allow queuing data to be reset after a systems failure or ungraceful shutdown. Alternatively, an automated resynchronisation approach could be adopted as described already.