# Cisco Remote Expert Mobile Advanced Developer's Guide, Release 11.6 (1)

**First Published:** August 2017

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

# Contents

# Preface

# Change History

This table lists the major changes made to this guide. The most recent changes appear at the top.

| Changes | Section | Date |
|---|---|---|
| Initial release of document for Release 11.6(1) | | Aug 2017 |
| Added Bluetooth Support sections | **Bluetooth Support on page 44**<br>**Bluetooth Support on page 61** | |
| Added CallKit Integration section | **CallKit Integration on page 40** | |
| Updated chapters:<br>■ Creating the Web Application<br>■ Creating a Browser Client Application<br>■ Creating an iOS Client Application<br>■ Creating an Android Client Application<br>■ Creating an OSX Client Application<br>■ Creating a Windows Client Application<br>■ Creating a .NET Client Application | **Creating the Web Application on page 3**<br>**Creating a Browser Client Application on page 11**<br>**Creating an iOS Client Application on page 27**<br>**Creating an Android Client Application on page 49**<br>**Creating an OSX Client Application on page 67**<br>**Creating a Windows Client Application on page 75**<br>**Creating a Windows .NET Client Application on page 85** | |
| Added Application Event Distribution section | **Adding Application Event Distribution on page 20** | |

# About this Guide

This document outlines the steps to develop mobile and web applications that use Cisco Remote Expert Mobile (RE Mobile).

Developers using this guide should have experience in JavaScript, Objective C or Java, depending on the application type.

- Web—It is assumed that the developer is familiar with JavaScript, HTML and CSS.
- iOS—It is assumed that the developer is familiar with iOS, Xcode and Objective-C
- Android—It is assumed that the developer is familiar with Android, Java, and the Android SDK

This guide also assumes that you are familiar with basic contact center and unified communications terms and concepts.

Successful deployment of Remote Expert Mobile also requires familiarity with the information presented in the *Solution Design Guide for Unified Contact Center Enterprise, Release 11.6* (available at http://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucm/srnd/collab11/collab11.html). To review IP Telephony terms and concepts, see the documentation at the preceding link.

# Audience

The primary audience for this guide is developers who need to use the call control features of Remote Expert Mobile in their applications.

# Related Documents

Consult these documents for details of these subjects that are not covered in this guide.

| Subject | Link |
|---|---|
| *Compatibility Matrix* for information on which versions of which products are supported for a contact center enterprise solution. | https://www.cisco.com/c/en/us/support/customer-collaboration/unified-contact-center-enterprise/products-device-support-tables-list.html |
| *Cisco Unified Contact Center Enterprise Features Guide* for detailed information on the configuration and administration of integrated features in your solution. | http://www.cisco.com/c/en/us/support/customer-collaboration/unified-contact-center-enterprise/products-feature-guides-list.html |
| *Cisco Collaboration Systems Solution Reference Network Designs* for detailed information on the Unified Communications infrastructure on which your solution is built. | http://www.cisco.com/c/en/us/support/unified-communications/unified-communications-manager-callmanager/products-implementation-design-guides-list.html |

You can find the full documentation of each of the components in the Unified CCE solution at these sites:

| Component | Link |
|---|---|
| Cisco Unified Contact Center Enterprise | http://www.cisco.com/c/en/us/support/customer-collaboration/unified-contact-center-enterprise/tsd-products-support-series-home.html |
| Cisco Finesse | http://www.cisco.com/c/en/us/support/customer-collaboration/finesse/tsd-products-support-series-home.html |
| Cisco MediaSense | http://www.cisco.com/c/en/us/support/customer-collaboration/mediasense/tsd-products-support-series-home.html |
| Cisco SocialMiner | http://www.cisco.com/c/en/us/support/customer-collaboration/socialminer/tsd-products-support-series-home.html |
| Cisco Unified Customer Voice Portal | http://www.cisco.com/c/en/us/support/customer-collaboration/unified-customer-voice-portal/tsd-products-support-series-home.html |
| Cisco Unified Intelligence Center | http://www.cisco.com/c/en/us/support/customer-collaboration/unified-intelligence-center/tsd-products-support-series-home.html |
| Cisco Virtualized Voice Browser | http://www.cisco.com/c/en/us/support/customer-collaboration/virtualized-voice-browser/tsd-products-support-series-home.html |

# Organization of this Guide

The guide includes the following sections:

| | |
|---|---|
| **Introduction** | Provides details of other documentation for Remote Expert Mobile. |
| **Creating the Web Application** | Describes how to start creating an application to use with Remote Expert Mobile |
| **Creating a Browser Client Application** | Describes how to create an application to use in a web browser |
| **Creating an iOS Client Application** | Describes how to create an application to use on an Apple iPhone or iPad |
| **Creating an Android Client Application** | Describes how to create an application to use on an Android device |
| **Creating an OSX Client Application** | Describes how to create an application to use on an Apple computer |
| **Creating a Windows Client Application** | Describes how to create an application to use on a Windows computer |
| **Creating a Windows .NET Client Application** | Describes how to create an application using the .NET framework to use on a Windows computer |

# Obtaining Documentation and Submitting a Service Request

To receive new and revised Cisco technical content directly to your desktop, you can subscribe to the What's New in Cisco Product Documentation RSS feed. RSS feeds are a free service.

# Field Alerts and Field Notices

Cisco products may be modified or key processes may be determined to be important. These are announced through use of the Cisco Field Alerts and Cisco Field Notices. You can register to receive Field Alerts and Field Notices through the Product Alert Tool on Cisco.com. This tool enables you to create a profile to receive announcements by selecting all products of interest.

Log into www.cisco.com and then access the tool at http://www.cisco.com/cisco/support/notifications.html.

# Documentation Feedback

To provide comments about this document, send an email message to the following address: contactcenterproducts_docfeedback@cisco.com.

We appreciate your comments.

# Conventions

This document uses the following conventions:

| Convention | Indication |
|---|---|
| **boldface** font | Boldface font is used to indicate commands, such as user entries, keys, buttons, and folder and submenu names. For example:<br>■ Choose Edit > Find.<br>■ Click Finish. |
| *italic* font | Italic font is used to indicate the following:<br>■ To introduce a new term. Example: A *skill group* is a collection of agents who share similar skills.<br>■ A syntax value that the user must replace. Example: IF (*condition, true-value, false-value*)<br>■ A book title. Example: See the *Cisco Unified Contact Center Enterprise Installation and Upgrade Guide*. |
| [ ] | Elements in square brackets are optional. |
| { x \| y \| z } | Required alternative keywords are grouped in braces and separated by vertical bars. |
| [ x \| y \| z ] | Optional alternative keywords are grouped in brackets and separated by vertical bars. |
| string | A non-quoted sequence of characters. Do not use quotation marks around the string or the string will include the quotation marks. |
| `window font` | Window font, such as Courier, is used for the following:<br>■ Text as it appears in code or that the window displays. Example:<br>`<html><title>Cisco Systems, Inc. </title></html>` |
| < > | Angle brackets are used to indicate the following:<br>• For arguments where the context does not allow italic, such as ASCII output.<br>• A character string that the user enters but that does not appear on the window such as a password. |
| [ ] | Default responses to system prompts are in square brackets. |
| !, # | An exclamation point (!) or a pound sign (#) at the beginning of a line of code indicates a comment line. |

CHAPTER 1

# Introduction

Cisco Remote Expert Mobile is a software solution that enables personal and actionable customer interactions within mobile and web applications. See the following guide for further information:

- *Cisco Remote Expert Documentation Guide*

CHAPTER 2

# Creating the Web Application



See the *Remote Expert Mobile Architecture Guide* for details on how the Web Application fits into the Remote Expert Mobile architecture.

Before doing anything, the REM Advanced SDK enabled application needs to create a session on the server; the server exposes a session creation REST API for the purpose. Although it would be possible in principle to dispense with the Web Application, and expose that API directly to the applications, such an approach has serious drawbacks:

- There is no authentication of users - any application which presents a valid *Web Application ID* to the REST service is able to create or destroy a session.

**Note** The Web Application ID is a unique text string which identifies the web application to the Web Gateway, and confirms that the web application is allowed to create sessions. The Web Application ID must correspond with one on the list of acceptable Web Application IDs configured on the Web Gateway (see *Remote Expert Mobile Administration Guide*).

- There is no authorization of users - an application which can create a session can create a session with any or all permissions, whereas you may want some users to have more access to REM Advanced SDK facilities than others.

The recommended approach is to create a Web Application which users can log in to with a user name and credentials, and which will return the session token to authenticated users. The web application will:

- Authenticate users and determine the services available to them
- Create sessions on the Web Gateway

- End sessions on the Web Gateway

# Communication with the Client

While the Remote Expert Mobile defines the way in which both the web application and client communicate with the Web Gateway, it does not restrict how the web application communicates with the client.

The web application needs to be able to pass the token containing the session ID created by the Web Gateway to the client. However, whether this is done via a REST API, HTML, or any other method is up to you.

# Authenticating the User

How the Web Application authenticates the user is entirely in control of the application itself. The sample application provided with REM Advanced SDK uses a particularly simple scheme, in which an `XML` file which is deployed to the server contains the users and their capabilities (look at this in conjunction with the sample code):

1. The sample application's login servlet receives an HTTP request containing a user name and password, either as part of a JSON body, or as parameters to the request (`LoginServlet.handleLoginFromWebpage` and `BaseLoginServlet.getUserLoginSessionID`).
2. `getUserLoginSessionID` gets the capabilities of the user (`LoginHandler.getUserFromLoginCredentials`).
   - `getUserFromLoginCredentials` parses the request for user name and password (`LoginRequestParser.parse`).
   - `getUserFromLoginCredentials` checks the password and returns the capabilities of the user (`LoginHandler.getAuthorizedUser`)
3. `getUserLoginSessionID` creates a session for the user (`LoginHandler.createSessionForUser`) and returns the session token to the REM Advanced SDK application.

It's easy to see how this could be adapted to a scheme where the information about each user was held in a secure database, or on an LDAP server.

# Creating the Session

Once the user has been authenticated, the web application sends a `POST` request to the Web Gateway which describes the requested capabilities for the session, such as:

- Can the user make voice and video calls?
- Does the user have AED (Application Event Distribution) capabilities?

The request also contains the Web Application ID and some further information (not relating to the session's capabilities) about the session itself.

The message must be `POST`ed to one of the following URLs:

- `http://<reas address>:8080/sessions` for non-secure communications
- `https://<reas address>:8443/sessions` for secure communications

**Note** As the message contains the Web Application ID, Cisco recommends that this transaction is performed over HTTPS for security.

The content type of the `POST` message should be `application/json`, and the body must be formatted as a JSON string:

```
{
   "timeout":1,
   "webAppId":"WEBAPPCSDK-A8C1D",
   "allowedOrigins":["example.com"],
   "urlSchemeDetails":
   {
      "secure":true,
      "host":"wg.example.com",
      "port":"8443"
   },
   "voice":
   {
      "username":"jbloggs",
      "displayName":"Joseph",
      "domain":"example.com",
      "inboundCallingEnabled":true,
      "allowedOutboundDestination":"sip:user@example.com",
      "auth":
      {
         "username":"1234",
         "password":"123456",
         "realm":"example.com"
      }
   },
   "aed":
   {
      "accessibleSessionIdRegex":".*",
      "maxMessageAndUploadSize":"5000",
      "dataAllowance":"5000"
   },
   "uuiData":"0123456789ABCDEF"
}
```

where the members are as follows:

| Member | Description |
|---|---|
| `timeout` | The timeout period for the session, defined in minutes. If omitted, this is set to `1` by default.<br>**Note:** The valid timeout range is `1-15` minutes, setting it to any other value outside of this range will cause errors. |
| `webappid` | The unique ID that the web app passes to the Gateway to identify itself. The ID must be a minimum of 16 characters in length, and must also have been configured on the Gateway itself. |
| `allowedOrigins` | This represents the origins from which cross realm JavaScript calls are permitted. If null or empty, there is no restriction. This is a comma separated list. |

| Member | Description |
|---|---|
| `urlSchemeDetails` | The connection details the Remote Expert Mobile client library is configured to use to the Web Gateway. This is an object with three other settings. If these details are not provided, the default setting for each option is used:<br>■ `secure`<br>If `true`, connects using secure WebSockets (`wss`). The default value is `false`, for non-secure (`ws`).<br>■ `host`<br>Specifies the host name or IP address for the WebSocket to connect to. If not provided, the client uses the `<web_gateway_address>` that the Web Application used to issue the HTTP `POST` request. Typically, this value is set when a NAT firewall is placed between the clients and the gateway. This value should be set to the external host name or IP address.<br>■ `port`<br>Specifies the port that the WebSocket connects to. The default is set to `8443` if `secure` is `true` or `8080` if `secure` is `false`. |
| `voice` | The details regarding voice and video calling. If omitted, voice and video calling are disabled. It is an object with the following members:<br>■ `username`<br>The SIP user name, as would appear in the `From` header<br>■ `displayName`<br>The SIP display name, as it would appear in SIP messages. If this is omitted, no display name is set for the user.<br>■ `domain`<br>The corresponding SIP domain.<br>■ `inboundCallingEnabled`<br>Set inbound calling parameters to enable inbound calling. If this is omitted, inbound calling is enabled by default.<br>**Note:** If `inboundCallingEnabled` is set to `true`, a SIP `REGISTER` request is sent to the SIP network; therefore, a corresponding user must exist on the SIP network. This user's credentials should be entered in the `auth` section (see below).<br>If `inboundCallingEnabled` is set to `false`, a SIP `REGISTER` is not sent.<br>■ `allowedOutboundDestination`<br>This can be a single destination, for example `sip:bob@example.com` or can be the string `all` to allow unrestricted calling.<br>■ `auth`<br>The authentication credentials for voice and video calling. This section can be omitted if the gateway is a trusted entity in the SIP infrastructure; however, if it is omitted and the SIP is challenged, the registration fails.<br>— `username`<br>The user name you would register with. This is a mandatory setting for voice calling.<br>— `password`<br>The password used for registrations. This is a mandatory setting |

| Member | Description |
|---|---|
| | for voice calling.<br>— `realm`<br>The realm used for registrations.<br>**Note:** The `username` used in the `From` header can be the same as the `username` used for authentication. The `domain` specified in the `From` header can be the same as the `realm` used for authentication. |
| `aed` | The details related to AED. If this section is omitted, AED functionality is disabled. It is an object with the following members:<br>■ `accessibleSessionIdRegex`<br>A Java regular expression which defines the AED topic names which this session can subscribe to. The user will not be able to subscribe to any AED topic which does not match this expression.<br>■ `maxMessageAndUploadSize`<br>Limits the size of message (in bytes) a user can send, and the size (in bytes) of an individual data upload.<br>■ `dataAllowance`<br>The total data (in bytes) a user can have stored at any time, on all topics they are subscribed to. |
| `uuiData` | If provided, this string is used to populate SIP `INVITE` and `BYE` messages sent by the user with a `User-to-User` header. As an example, suppose the value of this parameter is `ABCD`. The REM Advanced SDK adds the header `User-to-User: ABCD`. If this parameter is omitted, no `User-to-User` header is added to SIP messages.<br>Examples of valid `uuiData` values are:<br>`abcdef;encoding=hex`<br>`abcdefghijk;encoding=blah;paramname=paramvalue`<br>`"abcdefghijk";encoding=blah` |

To be a valid JSON string for creating a session, the JSON must obey the following rules:

- The `webAppId` must always be included.
- At least one of `voice` or `aed` must be included.
- If `voice` is included, then it must include the `username` and `domain`.

# Examples

See the following examples of `POST` messages for examples of how to start sessions with specific capabilities:

## Voice and Video Calling

For voice and video calling, using all the default settings:

```
{
   "webAppId":"WEBAPPCSDK-A8C1D",
   "voice":
   {
      "username":"jbloggs",
      "displayName":"Joseph",
      "domain":"example.com"
   }
}
```

## Voice and Video Calling with URL

For voice and video calling, specifying URL scheme details and an allowed origin:

```
{
   "webAppId":"WEBAPPCSDK-A8C1D",
   "allowedOrigins":["example.com"],
   "urlSchemeDetails":
   {
      "secure":true,
      "host":"wg.example.com",
      "port":"8443"
   },
   "voice":
   {
      "username":"jbloggs",
      "displayName":"Joseph",
      "domain":"example.com",
      "inboundCallingEnabled":true,
      "allowedOutboundDestination":"sip:user@example.com",
      "auth":
      {
         "username":"1234",
         "password":"123456",
         "realm":"example.com"
      }
   }
}
```

## AED Only

For a client application with AED capabilities only:

```
{
   "webAppId":"WEBAPPCSDK-A8C1D",
   "aed":
   {
      "accessibleSessionIdRegex":".*",
      "maxMessageAndUploadSize":"5000",
      "dataAllowance":"5000"
   }
}
```

## Using the UUI

For a client application which passes a UUI in the SIP `INVITE`:

```
{
   "webAppId":"WEBAPPCSDK-A8C1D",
   ...
   "uuiData":"53656e736974697665652044617461;encoding=hex"
}
```

You might use this technique to put some sensitive data in the `User-to-User` header which the consumer application does not know (or at least, does not transmit). When the consumer application

requests a session token, the Web Application puts the sensitive data, known only to itself, in the `uuiData` element of the JSON which it sends to the session token servlet. The Web Gateway associates that data with the session it creates. When the consumer application makes a call to a SIP device using that session, the Web gateway populates the `User-to-User` header of the `INVITE` which it sends to the SIP device with the sensitive data. How the SIP device uses the data is a matter for the device itself, but it could be an authentication token which allows the call to be set up.

If the data which needs to be sent to the SIP device is not sensitive, the consumer application can send it to the Web Application, and the Web Application can copy it to the `uuiData` element of the JSON it uses to create the session token.

## JSON Response

When it has created the session, the Web Gateway responds with a JSON string containing configuration data for the client, which includes a session ID for the new session; the Web Application must pass this token to the client application. If the JSON submitted to the Web Gateway contains properties with names that are unknown to the Gateway, a list of those unknown properties is placed in an `unknownProperties` object. This object is omitted if there are no unknown properties:

```
{
   "sessionid" : "<very long string..>",
   "unknownProperties" : ["<propname1>","<propname2>",...]
}
```

# Ending the Session

The Web Application should end the session on the Web Gateway when it knows that it will no longer be needed. The sample application included with REM Advanced SDK does this in response to an explicit request from the REM Advanced SDK application to a logout servlet, but it could happen in response to a timer firing, or the call ending.

To end the session, the Web Application needs to send an HTTP `DELETE` request containing the session ID to the Web Gateway at one of the following URLs:

- `http://<reas address>:8080/gateway/sessions/session/id/<session-id>` for non-secure communications
- `https://<reas address>:8443/gateway/sessions/session/id/<session-id>` for secure communication


**Note**

- The response for the `DELETE` operation will be `204 No Content`. This is conventional in REST services, as nothing is returned in the response.
- This tears down any calls the user has active and invalidates the session.

CHAPTER 3

# Creating a Browser Client Application



Remote Expert Mobile enables you to develop browser-based applications offering users the following methods of communication:

- Voice and video calling
- Application Event Distribution

You can also enhance any existing browser-based offerings with these features.

Remote Expert Mobile provides you with a network infrastructure and JavaScript API which make use of technologies such as WebRTC to integrate seamlessly with your existing SIP infrastructure. The JavaScript API is delivered with its own reference javadocs available at

`<installation_directory>/Core_SDK/JavaScript_SDK/jsdoc`.

For more detailed discussion of the Remote Expert Mobile solution, refer to *Remote Expert Mobile Overview*.

**Note** Remote Expert Mobile is delivered with a sample application. This is available at `<installation_directory>/Core_SDK/Sample_Source`. All samples featured in this guide can be located there.

# Setting up a Project

**Note**

Before following this process, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **Creating the Web Application on page 3**.

In your development environment, start a new project and create the page to use to deliver your client application. The Remote Expert Mobile JavaScript SDK files were installed when you installed the Web Gateway, so will be automatically available to any client application which accesses the Web Gateway itself. Each page which uses the JavaScript API will need to include the following `script` tags:

```
<script src="http://<reas address>:<port>/gateway/scripts/adapter.js"/>
<script src="http://<reas address>:<port>/gateway/scripts/csdk-sdk.js"/>
```

where `<reas address>` is the IP address or host name of the REAS on which you have installed the Web Gateway, and `<port>` is the port to connect to to access it (usually `8080` for `http`). If the Gateway is set up for secure access only, use `https` instead of `http`, and `8443` for the port.

**Note**

`csdk-sdk.js` implements voice and video calling and AED; if you only want to implement a subset of this functionality, you can include only the script for the functionality you require:

- `csdk-phone.js` for voice and video calling (including floor control).
- `csdk-aed.js` for AED.

**Important**

If you are using a subset of functionality, ensure that you include `csdk-common.js` after the modules you require. If you are using only AED, and want to avoid prompting the user for access to their microphone and camera, ensure that you do not include `csdk-phone.js` (either directly or indirectly).

When the JavaScript runs, it creates an object called `UC` in the global namespace.

# Initializing the SDK

To set up all the functionality to which the user has access, you need to obtain a session ID from your Web Application (see **Creating the Web Application on page 3**), and initialize `UC` using it. Once the client has the Session ID, it must call the `start` method on the `UC` object.

To confirm that `UC` has initialized correctly, you can use the `onInitialised` method. To determine whether `UC` has failed to initialize, you should implement `onInitialisedFailed`.

```
//Get hold of the sessionID however your app needs to
var sessionID = getSessionID();
// Set up STUN server list
var stunServers=[{"url": "stun:stun.l.google.com:19302"}];

UC.onInitialised = function() {
   //Register listeners on UC
   UC.phone.onIncomingCall = function(call) {
      // perform tasks associated with incoming call
   };
    ⋮
};

UC.onInitialisedFailed = function() {
    ⋮
```

```
};

//Start UC session using the Session ID and stun server list
UC.start(sessionID, stunServers);
```

Once the `UC` object has initialized, the application has access to its `phone` and `aed` objects, which it can use to make and receive calls, or to access any other REM Advanced SDK functionality. Note that the above code assigns the `phone.onIncomingCall` function only in the `onInitialised` callback; this is typical - the `phone` and `aed` objects can only be used inside the `onInitialised` callback, or after it has been received.

**Note**

- STUN servers are not necessary if the Gateway is not behind a firewall, so that *Network Address Translation* is not needed, in which case the `stunServers` array can be empty. You can provide your own STUN server instead of the public Google one above; and you can provide more than one in the array, in which case they will be tried in sequence until it finds a working one.
- Your Web Application may fail to return a session ID (for instance, if it cannot authenticate the user). In these situations the user should be logged out and needs to log in again to start a new session (see **Creating the Session on page 4**).

# Checking Browser Compatibility

`UC.checkBrowserCompatibility(pluginInfoCallback)` checks the browser for compatibility with `UC`. This function is asynchronous - the function `pluginInfoCallback` is called to return the information.

`pluginInfoCallback` is called with an argument (`pluginInfo`), which is a JavaScript object with the following members:

| Member | Values |
|---|---|
| `pluginRequired` | ■ `true`<br>If the browser needs a plug-in to operate correctly<br>■ `false`<br>Otherwise |
| `status` | ■ zero length string<br>If `pluginRequired` is `false`<br>■ `installRequired`<br>If the plug-in is missing<br>■ `upgradeRequired`<br>If the plug-in is present but a new version is needed<br>■ `upgradeOptional`<br>If the plug-in is present and will work, but a newer version is available<br>■ `upToDate`<br>If the plug-in is present and is the latest available version |
| `restartRequired` | ■ `true`<br>If a plug-in is installed or upgraded, the browser will need to be restarted<br>■ `false`<br>The browser will not need to be restarted, or no plug-in is needed |
| `installedVersion` | ■ `none`<br>When `pluginRequired` is false or the plug-in is missing |

| Member | Values |
|---|---|
| | ■ string in form `x.y.z`<br>Where `x`, `y`, and `z` are integers |
| `minimumRequired` | ■ `none`<br>When `pluginRequired` is false or the plug-in is missing<br>■ string in form `x.y.z`<br>Where `x`, `y`, and `z` are integers. This is the minimum version of the plug-in which will work correctly with the version of REM Advanced SDK in use. |
| `latestAvailable` | ■ `none`<br>When `pluginRequired` is false or the plug-in is missing<br>■ string in form `x.y.z`<br>Where `x`, `y`, and `z` are integers. This is the latest version of the plug-in available on the server. |
| `pluginUrl` | ■ zero length string<br>If `pluginRequired` is `false`<br>■ URL string<br>The URL points to the location of the latest version of the plug-in on the server (the version indicated by `latestAvailable`). |

`UC.start` assumes the presence of a correct browser plug-in (if required). If this is not the case, an error may occur. If the plug-in information indicates that a plug-in or plug-in update is needed, the application will need to prompt the user to install it from the `pluginUrl` provided. See the sample application's `entry.js` file for the way this can be done.

**Note** The user may not have permission to install plug-ins. In this case, it will be the responsibility of their IT administrator to install the correct plugin, and the application should inform the user of the problem.

# Adding Voice and Video

All of the functions required to develop applications for browser-based voice and video are supported by the `UC.phone` object. This is an instance of the `Phone` class.

## Adding a Preview Window

If you want to add a preview window (a window which displays the video which is being sent to the other endpoint) before a call is established, you can call the `UC.phone.setPreviewElement` function. An appropriate time to do this is in the `UC.onInitialised` callback:

```
UC.onInitialised = function() {
   UC.phone.setPreviewElement(document.getElementById('local'));
};
```

Alternatively, you can wait until you have a call (see **Making a Call on the facing page** and **Receiving a Call on page 16**) before setting the preview element:

```
var call;
call = UC.phone.createCall(numberToDial);
call.onInCall = function() {
   call.setPreviewElement(document.getElementById('local'));
}
```

# Making a Call

The `phone` object provides a `createCall` method, to which your client application should provide the number to contact. This returns a new `call` object, on which you can set callbacks and call the `dial()` method, which initiates a call to the destination specified for the call. The `dial()` method takes two string parameters:

- `withAudio` - to define the direction of the audio stream in the call.
- `withVideo` - to define the direction of the video stream in the call.

For both parameters, the possible values are:

- `enabled` – for 2 way media
- `onlyreceive` – for 1 way media
- `disabled` – for no media

The default for both parameters is `enabled`, which provides backward compatibility and convenience, as the application need only call `dial` to establish 2 way communication on both voice and video (considered the normal case).


**Note**

`call.dial()` must only be called after the application has initialized the SDK and received the `UC.onInitialised` callback (see **Initializing the SDK on page 12**).

```
var call;
//A method to call from the UI to make a call
function makeCall(numberToDial) {
   //Create a call object from the framework and save it somewhere
   call = UC.phone.createCall(numberToDial);

   call.onInCall = function() {
      // Show video stream(s) in elements
      call.setPreviewElement(previewVideoElement);
      call.setVideoElement(remoteVideoElement);
   };

   //Set what to do when the remote party ends the call
   call.onEnded = function() {
      alert("Call Ended");
   };

   //Set up what to do if the callee is busy, inform your user etc
   call.onBusy = function() {
      alert("The callee was busy");
   };

   //Dial the call
   call.dial();
};

//A method to call from the UI to end a current call
function endCall() {
   call.end();
```

```
};
```

In order to use media in the call, the the application must provide a `div` element on the page where it will display remote video from the other endpoint, using the `setVideoElement` function. The `onInCall` call-back is a suitable place to do this; in the above code, it also calls `setPreviewElement` to display a copy of the local video (which is being sent to the far end).

We recommend that the application should override the following error methods to inform the user of call status, in the event that any issues occur when making a call:

- `onBusy`
- `onCallFailed`
- `onDialFailed`
- `onGetUserMediaError`
- `onNotFound`
- `onTimeout`

As shown above, to end the call the client should call the call object's `end` method.

## Receiving a Call

Overriding `onIncomingCall` allows REM Advanced SDK to notify the client application when it receives a call. The notification has a `Call` object as a parameter; the `Call` object contains details of the call in progress and some key methods which should be overridden.

In a simple application, showing some user feedback when this object is called enables a user to receive a call.

```
var call;
//Define what to do on incoming call
UC.phone.onIncomingCall = function(newCall) {
   var response = confirm("Call from: " + newCall.getRemoteAddress() +
   " - Would you like to answer?");
   if (response === true) {
      //What to do when the remote party ends the call
      newCall.onEnded = function() {
         alert("Call Ended");
      };
      //Remember the call to enable ending later
      call = newCall;
      //Specify where preview and remote video should be played/presented.
      call.setPreviewElement(previewVideoElement);
      call.setVideoElement(remoteVideoElement);
      //Answer
      newCall.answer();
   } else {
      //Reject the call
      newCall.end();
   }
};

//A method to call from the UI to end the call
function endCall() {
   call.end();
}
```

To answer the call, your client application needs to call the call object's `answer()` method.

The `answer()` method takes two string parameters:

- `withAudio` - defines the direction of the audio stream in the call.
- `withVideo` - defines the direction of the video stream in the call.

For both parameters, the possible values are:

- `enabled` – for 2 way media
- `onlyreceive` – for 1 way media
- `disabled` – for no media

The default for both parameters is `enabled`, which provides backward compatibility and convenience, as the application need only call `answer` to establish 2 way communication on both voice and video (considered the normal case).

To reject the call, your client application needs to call the `call` object's `end` method.

# Enabling Local Media

In order to send local media to the Web Gateway, the application must call `setLocalMediaEnabled`. The `setLocalMediaEnabled()` method supports two boolean parameters:

- `enableVideo` - enables a stream for the user's camera/webcam.
- `enableAudio` - enables a stream from the user's microphone.

The `setLocalMediaEnabled()` method also supports a single parameter JavaScript object which contains both the audio and video capabilities. This object contains two boolean parameters, `audio` and `video`, which can be set separately:

```
{"audio": true, "video": false}
```

# Adding a Stream

To enable the client you develop to play any audio and video provided by the framework, you must call `Call.setVideoElement`, passing in the element that is be used to display the video and the ID of the stream to be displayed:

```
call.onInCall = function() {
   call.setVideoElement(document.getElementById(remote'), 'streamid');
}
```

The stream ID is optional, and defaults to `'main'` if not provided. The application should only need to specify it if the framework is providing more than one video stream; in that case, the application should know what those stream IDs are, and may give the user some way to switch between them.

## The Size of the Video Window

In IE, the element which displays the video (whether remote or local) needs to have a minimum size; in IE the default height is 0, and it does not expand to accommodate the video stream when that starts. You can correct this with a CSS entry:

```
#remote > *, #local > * {
min-height: 500px;
width: 100%;
}
```

Note that you need to set the elements that are *immediate children* of the `remote` and `local` elements (assuming that `remote` and `local` are the elements which you will pass to `setVideoElement` and `setPreviewEement`); REM Advanced SDK will add a child to the `remote` and `local` elements, with the same size as its parent, and it is that child which will display the video. This applies to all of `Call.setPreviewElement`, `Phone.setPreviewElement`, and `Call.setVideoElement`.

## Ending a Call

If the user ends the call, the client application should call the `Call` object's `end` method.

In order to detect that the remote party has ended the call, the client application needs to override the `Call` object's `onEnded` callback method.

## Muting the Local Audio and Video Streams

During a call, the application can mute and unmute the local audio and video streams separately. Muting the stream stops that stream being sent to the remote party. The remote party's stream continues to play locally, however.

To mute either stream, use the `setLocalMediaEnabled(enableVideo, enableAudio)` methods of the `Call` object to toggle the audio and video streams. See **Enabling Local Media on the previous page**.

## Holding and Resuming a Call

If the user puts a call on hold, the client application should call the `call` object's `hold()` method.

To resume a call that currently on hold, the client application should call the `call` object's `resume()` method.

## Sending DTMF Tones

Your application can send DTMF tones on a call by using the `Call.sendDtmf` function:

```
call.sendDtmf("#123*", true);
```

The first parameter is a string representing the tones to send. Valid values for the tones are those characters conventionally used to represent the standard DTMF tones: `0123456789ABCD#*`. A comma character inserts a two-second pause into a sequence of tones. Alternatively, to send a single tone, the application can pass in a number from `0` to `9`.

The second parameter should be `true` if you want the tones to also be played back locally, and be audible to the user.

## Handling Multiple Calls

Applications developed with Remote Expert Mobile JavaScript SDK support multiple simultaneous calls:

- To make additional calls while another call is in progress, the client application would use the `UC.phone.createCall(numberToDial)` method (see **Making a Call on page 15**).
- To receive incoming calls while another call is in progress, the `UC.phone.onIncomingCall` method should be triggered (see **Receiving a Call on page 16**).

**Note** Multiple simultaneous calls are *not* supported on the IE or Safari plugins.

## Setting Video Resolution

The Remote Expert Mobile JavaScript SDK supports configuring the captured, and therefore sent, video resolution for video calls. The application can select one of a set of video resolutions, and apply it to the capture device. It can also configure the frame rate for capture. When it specifies a resolution and frame rate, REM Advanced SDK makes every effort to match those values where hardware allows.

**Note** The new resolution and frame rate only take effect for subsequent calls, and do not affect calls that are in progress.

## Enumerating Possible Resolutions

The application can get a list of possible resolutions from the `Phone` object using the `videoresolutions` array:

```
var lowestResolution = UC.phone.videoresolutions[0];
```

These values are an enumeration which list all supported resolutions:

| Enumeration Value | Width | Height |
|---|---|---|
| `videoCaptureResolution174x144` | `174` | `144` |
| `videoCaptureResolution352x288` | `352` | `288` |
| `videoCaptureResolution320x240` | `320` | `240` |
| `videoCaptureResolution640x480` | `640` | `480` |
| `videoCaptureResolution1280x720` | `1280` | `720` |

**Note**

When you set the resolution, the device's camera may not support that resolution. In that case the browser provides a different resolution. Currently, for example, Google Chrome defaults to a resolution of `640x480` if the requested resolution is not available.

## Setting the Resolution

The application can set he captured video resolution using the `setPreferredVideoCaptureResolution(resolution)` method of the `Phone` object. The value supplied must be one of the video resolutions presented in the `videoresolutions` array (see **Enumerating Possible Resolutions above**):

```
var hd720p = UC.phone.videoresolutions.videoCaptureResolution1280x720;
UC.phone.setPreferredVideoCaptureResolution(hd720p);
```

## Setting an Arbitrary Video Resolution

There may be circumstances when you want to set a specific video resolution not listed above. You can do this by specifying a JavaScript object which contains the width and height in pixels:

```
UC.phone.setPreferredVideoCaptureResolution({width:400,height:200});
```

The device must be able to support the resolution that you specify, or the browser provides a default resolution. Currently, for example, Google Chrome defaults to a resolution of `640x480` if the requested resolution is not available.

## Setting the Frame Rate

The application can set the captured video frame rate using the `setPreferredVideoFrameRate(rate)` method of the `Phone` object.

```
UC.phone.setPreferredVideoFrameRate(30);
```

# Handling User Media Issues

When a problem occurs obtaining the user media (i.e. microphone or camera) the Remote Expert Mobile provides 3 call backs on the `Call` object:

- `onOutboundAudioFailure` is called if there is a problem obtaining audio media (e.g. microphone is disabled or unplugged), but the call can continue without it. The application can decide whether to continue or end the call when it receives this callback.
- `onOutboundVideoFailure` is called if there is a problem obtaining video media (e.g. camera is

disabled or unplugged), but the call can continue without it. The application can decide whether to continue or end the call when it receives this callback.

- `getUserMediaError` is called when there is a terminal user media problem which has resulted in the call ending (e.g. if the user has denied permission to both the microphone and camera).


**Note**

REM Advanced SDK will only call the `onOutboundAudioFailure` and `onOutboundVideoFailure` methods if the relevant media type was requested when making the call, and the timing of these method callbacks will vary depending on the browser.

## Monitoring Call State

During call setup, the call will transition through several states, from the initial setup to being connected with media available (or failure). You can implement callbacks to get notifications of the transitions to some of these states, in order to provide feedback to the user or to take some other action.

The following table gives the available callbacks, on `Phone` and `Call` objects:

| Callback | Meaning |
|---|---|
| `Phone.onIncomingCall` | An incoming call is alerting (ringing). The callback provides the `Call` object as a parameter. See **Receiving a Call on page 16**. |
| `Call.onRinging` | An outgoing call is ringing at the remote end |
| `Call.onPending` | The call is connected, and waiting for media |
| `Call.onInCall` | The call is fully set up, including media |
| `Call.onBusy` | Dialed number is busy |
| `Call.onNotFound` | Dialed number is unreachable or does not exist |
| `Call.onTimeout` | There was no response from the dialed number within the network's timeout |
| `Call.onDialFailed` | Dialing the number failed. This may be due to several reasons, such as no media broker being available, or the capacity of the network being reached. The callback has an error code parameter which may give more information. |
| `Call.onCallFailed` | The call has errored. This may be due to no media, or a network failure, or some other reason. The callback supplies an error code as a parameter, which may give more information. |
| `Call.onEnded` | The call has ended |

# Adding Application Event Distribution

All of the functions required to develop applications for browser-based *Application Event Distribution* (AED) exist on the `UC.aed` object. The application can subscribe to a topic, and can send data (consisting of key-value pairs) or messages (simple text) to that topic, and have all other subscribers to that topic see the data and messages.

## Creating a Topic

To create a topic, the client application can call:

```
UC.aed.createTopic(topic);
```

where `topic` is a unique string identifier for the topic. This method call returns a `Topic` object.

## Subscribing to a Topic

Once it has created the topic, the client application can subscribe to a topic by calling:

```
topic.connect(timeout);
```

This method call triggers one of the following events on the client:

- `onConnectSuccess (data[])`
- `onConnectFailed`

The `onConnectSuccess` callback includes an array of data objects representing all the existing data (as key-value pairs) on the topic. Each object in the `data` array has the following members:

| Member | Description |
|---|---|
| `key` | Data's key |
| `value` | Data's value |
| `version` | A number indicating the data's version relative to other values that have been sent. Later versions have higher values. |
| `deleted` | Whether the data for this `key` was deleted. If this is `true`, `value` may be `null` or `undefined`. |

**Note**

The `key` and `value` elements of the data object must be strings.

Once it receives the `onConnectSuccess` callback, the application may receive `onTopicUpdate` notifications. `onTopicUpdate` will be fired each time anyone connected to the topic updates the topic's data or sends a message to it. The callback passes a JSON topic object which contains details of the new data update or message in the following format:

```
{
   "type": "topic",
   "name": "Pension",
   "data": [
      {"key":"dataKey", "value":"dataValue", "version":"0"},
      {...},
      {...},
      ...
   ],
   "message": "This my message!",
   "timeout": 120
}
```

This callback stops firing when the user unsubscribes from the topic (see **Unsubscribing from a Topic on the next page**).

The following code sample shows the code required to subscribe to a topic:

```
UC.onInitialised = function() {
   //create a topic
   var topic = UC.aed.createTopic('topic');

   topic.onConnectSuccess = function(data) {
      for (var i = 0; i < data.length; i++) {
         // Process each data object
      }
   }

   topic.onConnectFailed = function(message) {
      alert(message);
   }
```

```
    topic.onTopicUpdate = function(key, data, version, deleted) {
      // Store or display new data received
    }

    topic.connect();
};
```

**Note** The application can compare the `version` of a data object with the `version` of a stored version of the same data (i.e. which has the same `key`) to ensure that an older version does not replace a newer one.

## Unsubscribing from a Topic

The client can un-subscribe to a topic by calling

```
topic.disconnect(delete);
```

The `delete` argument is an optional `boolean`. If `true`, the topic will be removed from the server (disconnecting all users from the topic); if `false`, only the current user will be disconnected from the topic.

If `delete` is true, this method call triggers one of the following events on the client:

- `onDeleteTopicSuccess`
- `onDeleteTopicFailed`

`onTopicDeleted` is called on all the clients subscribing to the topic when a topic is successfully deleted from the server.

## Publishing Data to a Topic

The client can publish a key-value item of data to a topic.

```
topic.submitData(data_key, data_value);
```

Both `key` and `value` should be strings. This method call triggers one of the following events on the client:

- `onSubmitDataSuccess`
- `onSubmitDataFailed`

This action causes all clients subscribing to the topic to receive an `onTopicUpdate` event.

The following code sample shows the steps required to create a topic and publish data to it:

```
UC.onInitialised = function() {
  var topic = UC.aed.createTopic('topic');

  topic.onConnectSuccess = function(data) {
    // Submit new data when connected to topic
    topic.submitData('key_one', 'value');
  }

  topic.onConnectFailed = function(message) {
    alert(message);
  }

  topic.onSubmitDataSuccess = function(key, value, version) {
    // Log success
  }

  topic.onSubmitDataFailed = function(key, value, message) {
```

```
    // Notify user of failure
  }

  topic.onTopicUpdate = function(key, value, version, deleted) {
    // Display new data to user
  }

  topic.connect();
};
```

## Deleting Data from a Topic

The client can delete an item of data from a topic by specifying the item's key.

```
topic.deleteData(topic, data_key);
```

This method call triggers one of the following events on the client which called `deleteData`:

- `onDeleteSuccess`
- `onNotFound`.

A successful delete causes all clients subscribed to the topic to receive an `onTopicUpdate` event (with the `deleted` parameter set to `true`).

## Sending a Message to a Topic

The client can send a message containing data to the topic.

```
topic.sendMessage(msg);
```

The `msg` parameter is free-form text.

This method call triggers one of the following events on the client:

- `onSendMessageSuccess`
- `onSendMessageFailed`

This action causes all clients subscribed to the topic to receive an `onMessageReceived` event.

The following code sample shows the code required to send a message to all the clients subscribing to the topic:

```
UC.onInitialised = function() {
  topic.onConnectSuccess = function(data) {
    // Send message as soon as topic is connected
    topic.sendMessage(message_text);
  }

  topic.onSendMessageSuccess = function(message) {
    // Log success
  }

  topic.onSendMessageFailed = function(message, errorMessage) {
    alert(errorMessage + ", " + message);
  }

  topic.onMessageReceived(message) {
    // Display message to user
  }

  topic.connect();
```

```
};
```

# Ending the Session

To end the session, the client application needs to call in to the web application, which terminates the session as described in **Ending the Session on page 9**.

# Responding to Network Issues

As Remote Expert Mobile is network-based, it is essential that the client application is aware of any loss of network connection. When a network connection is lost, the server uses SIP timers to determine how long to keep the session alive before reallocating the relevant resources. Any application you develop should make use of the available callbacks in the Remote Expert Mobile API, and any other available technologies, to handle network failure scenarios.

## Reacting to Network Loss

In the event of network connection problems, the Remote Expert Mobile automatically tries to re-establish the connection. It will make seven attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to the `onConnectionRetry(attempt, delayUntilNextRetry)` method of the `UC` object precedes each of these attempts.

When all reconnection attempts are exhausted, the `UC` object receives the `onConnectivityLost` callback, and the retries stop.

If any of the reconnection attempts are successful, the `UC` object receives the `onConnectionReestablished` callback.

If REM Advanced SDK loses the network connection to the web, the `UC` object receives a `onNetworkUnavailable` callback. When the connection comes back, REM Advanced SDK attempts to re-establish the connection as above, and `onConnectionRetry` fires.

**Note** The retry intervals, and the number of retries attempted by the SDK are subject to change in future releases. Do not rely on the exact values given above.

## Network Quality Callbacks

The application can implement the `onCallQualityChanged` callback function on the`Call` object to receive callbacks on the quality of the network during a call:

```
call.onConnectionQualityChanged = function(connectionQuality) {
   // Show indication of quality
}
```

The `connectionQuality` parameter is a number between `0` and `100`, where `100` represents a perfect connection. The application might choose to show a bar in the UI, the length of the bar indicating the quality of the connection.

The SDK starts collecting metrics as soon as it receives the remote media stream. It does this every 5s, so the first quality callback fires roughly 5s after the remote media stream callback has been established.

The callback then occurs every time a different quality value is calculated, so if the quality is perfect then there is an initial quality callback with a value of `100` (after 5s), and then no further callback until the quality degrades.

# API Changes

| API Release | Description |
|---|---|
| 11.5 | Plug-in browser compatibility information is available through the `UC.checkBrowserCompatibility(pluginInfoCallback)` property. |
| 10.6 | Version information is available through the `UC.csdkVersion` property |
| | Additional connection/network callbacks on the UC object:<br>`onConnectionRetry`,<br>`onConnectionReestablished`,<br>`onNetworkUnavailable`. |
| | Added ability to determine inbound video quality.<br>Information about the quality of inbound video is available through the `onConnectionQualityChanged` method of the `Call` object. |
| | Added ability to set video resolution.<br>■ `UC.phone.setPreferredVideoCaptureResolution(...)`<br>■ `UC.phone.setPreferredVideoFrameRate(...)`<br>See **Setting Video Resolution on page 18** |
| | `onLocalMediaStream` and `onRemoteMediaStream` methods were on the phone object—both methods are now on the call method. |
| | The `dial()` and `answer()` methods on the call object take two boolean parameters:<br>■ withAudio<br>■ withVideo. |
| | `setLocalMediaEnabled()` supports two boolean parameters:<br>■ enableVideo<br>■ enableAudio<br>`setLocalMediaEnabled()` now also supports an optional single parameter JavaScript object in which audio, video and screensharing can be set. |
| | If the screenshare media stream cannot be acquired, `onScreenshareError()` is the call-back which is triggered |
| | After the browser acquires the localMediaStream object, the `onlocalMediaStream (localMediaStream)` callback is offered to the phone object. |

CHAPTER 4

# Creating an iOS Client Application



Remote Expert Mobile enables you to develop iOS applications offering users the following methods of communication:

- Voice and Video calling
- Application Event Distribution (AED).

Remote Expert Mobile provides you with an iOS SDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

To develop iOS applications using Remote Expert Mobile, Xcode 4.5 or later is required.

Information about the minimum version of iOS supported can be found in the *Release Notes*.

The Remote Expert iOS SDK is made up of the following classes:

- The top-level `ACBUC` class and its delegate protocol `ACBUCDelegate`.
- Two classes for voice and video calling:
  - `ACBClientPhone` and its delegate protocol `ACBClientPhoneDelegate`.
  - `ACBClientCall` and its delegate protocol `ACBClientCallDelegate`.
- Two classes for AED:
  - `ACBClientAED`
  - `ACBTopic` and its delegate protocol `ACBTopicDelegate`.

The iOS SDK reference documentation, including a full list of available methods and their associated call-backs, is delivered in the `docs.zip` file. Open `index.html` to view the API documentation.

# Setting up a Project

**Note** Before following this process, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **Creating the Web Application on page 3**.

To set up a project including the Remote Expert Mobile, you first need to create a new project and add iOS native frameworks to it:

1. Open Xcode and choose to create a **Single View Application**, giving your project an appropriate name. The following code samples use the example name `iOSCSDKSample`.
2. Click the **Build Phases** tab, and expand the **Link Binary with Libraries** section by clicking on the title.
3. Click the **+** button; the file explorer displays.
4. Select the following iOS native dependencies from the iOS folder:
   - `OpenGLES.framework`
   - `CoreVideo.framework`
   - `CoreMedia.framework`
   - `QuartzCore.framework`
   - `AudioToolbox.framework`
   - `AVFoundation.framework`
   - `UIKit.framework`
   - `Foundation.framework`
   - `CoreGraphics.framework`
   - `CFNetwork.framework`
   - `Security.framework`
   - `libicucore.dylib`.
   - `GLKit.framework`
   - `libsqlite3.dylib` (or equivalent alternative)
   - `libc++.dylib` (or equivalent alternative)

The dependencies you selected are now displayed in the **Link Binary with Libraries** section.

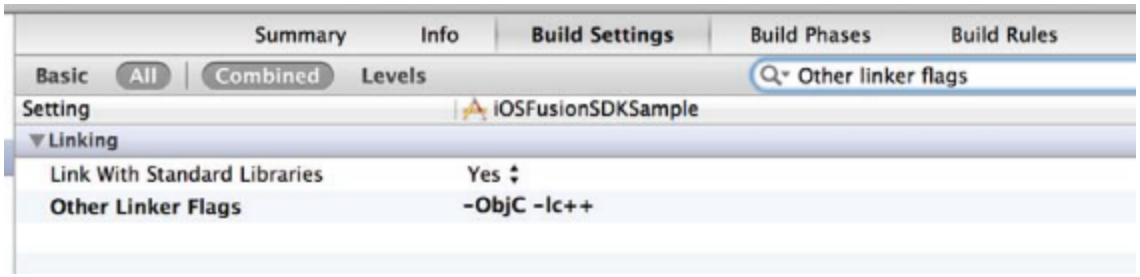
Now, you need to add the Remote Expert Mobile framework to your project.

1. Select your project and click the **Build Phases** tab.
2. Expand the **Link Binary with Libraries** section by clicking on the title.
3. Click the **+** button. When the file explorer displays, click **Add Other**.
4. Navigate to the `Frameworks/ACBClientSDK.framework` folder, select it and click **OK**.

To ensure that your project compiles, you need to configure its **Other Linker Flags** setting:

1. Select your project and click **Build Settings**.
2. Enter an appropriate term in the search field to find the **Other Linker Flags** setting, for example `Linker`. Click **Search**. **Other Linker Flags** display in the **Linking** section.
3. Configure **Other Linker Flags** with the following setting:

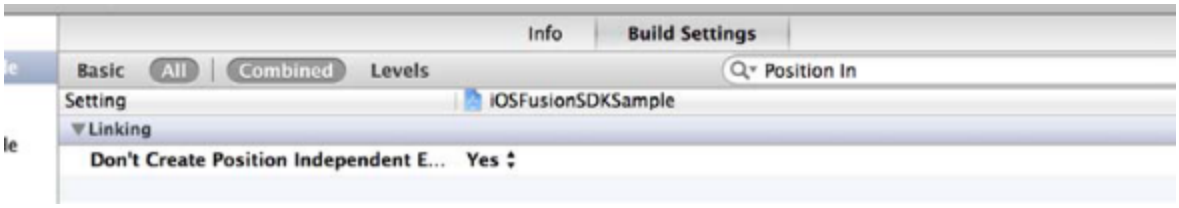
   ```
   -ObjC -lc++
   ```




**Note**

Some users have found problems with build failing due to text relocation issues. To solve this, also add the `-read_only_relocs suppress` flag to the above **Other Linker Flags**.

4. *Position Independent Executables* are incompatible with some codecs in the Remote Expert iOS SDK. In the **Linking** section, set **Don't Create Position Independent Executables** to `Yes`.




**Important**

When building a project created with Xcode 5 you may get linkage errors related to the Standard C++ library. This is caused by an Xcode 5 bug, which prevents it detecting the dependency the iOS SDK has on the C++ Standard Library. Either of the following actions can be taken to work-around this issue:

- Add `libstdc++.6.dylib` to the list of required libraries.
- Click the **Build Settings** tab, and set the **iOS Deployment Target** to an earlier version of iOS than iOS 7.0, but no earlier than the minimum iOS version supported by this version of the iOS SDK.

**Note**

For the best performance, we recommend that you build your application for all of the architectures that you are targeting. Ensure that all of your target architectures are listed in **Architectures** and **Valid Architectures** in your Xcode target build settings.

# iOS 9 and Xcode 7

Existing application binaries built with earlier versions of Xcode should continue to work without modification, although you may be prompted to trust the application or developer.

New or existing projects loaded into Xcode 7 requires changes before they build and run:

1. Disable the generation of bitcode

   `Enable Bitcode = NO.`

2. Add entries to your application's `plist` file to disable the new iOS 9 Application Transport Security feature - see the following for further information:

   https://developer.apple.com/library/prerelease/ios/technotes/App-Transport-Security-Technote/

**Note** The iOS sample application has been modified accordingly.

## Initializing the ACBUC Object

The application accesses the API initially via a single object, `ACBUC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see **Creating the Web Application on page 3**), and initialize the `ACBUC` object using it. Once it has received the Session ID, the client application must call the `ucWithConfiguration` method on the `ACBUC`:

```
- (void) initialize
{
   NSString* sessionId = [self getSessionId];
   ACBUC* uc = [ACBUC ucWithConfiguration:sessionId delegate:self];
   [uc startSession];
}

- (void) ucDidStartSession:(ACBUC *)uc
{
    ⋮
}
```

The delegate (in this case `self`) must implement the `ACBUCDelegate` protocol. Once the session has started, REM Advanced SDK will call the delegate method `ucDidStartSession`, and the application can make use of the `ACBUC` object. (If the session does not start, REM Advanced SDK will call one of the delegate's error methods.)

There is an alternative version of `ucWithConfiguration` for use if STUN is needed, which takes as an additional parameter, an `NSArray*` of STUN servers, each member an `NSString` in the form `stun:stun.1.google.com:19302`.

```
NSString* sessionId = [self getSessionId];
NSArray* stunServers = [NSArray
arrayWithObject:@"stun:stun.1.google.com:19302"];
ACBUC* uc = [ACBUC ucWithConfiguration:sessionId stunServers:stunServers
delegate:self];
[uc startSession];
```

**Note** STUN servers are not necessary if the Gateway is not behind a firewall, so that *Network Address Translation* is not needed. You can provide your own STUN server instead of the public Google one above; and you can provide more than one in the array, in which case they will be tried in sequence until REM Advanced SDK finds a working one.

# Adding Voice and Video

Once the application has initialized the `ACBUC` object, it can retrieve the `ACBClientPhone` object. It can then use the phone object to make or receive calls, for which it returns `ACBClientCall` objects. Each one of those objects has a delegate for notifications of errors and other events.

**Note** As of iOS 8.2, the iOS simulator does not support video and audio input, so in order to fully test your application with audio and video, you will have to deploy it to a real device.

## Requesting Permission to use the Microphone and Camera

On iOS 7.0 and higher, your application needs to ask the end user for permission to use the microphone and camera before they can make or receive calls. Also, because the microphone and camera permissions in iOS function at an application-level and not per call, you need to consider the most appropriate time to ask the end user for their permission. The answer they provide is remembered by iOS until your application is uninstalled or the permissions reset in the iOS Settings. Note also that the end user can change the microphone and camera permission for your application in iOS Settings too.

The iOS SDK provides a helper method to request access to the microphone and camera - see the `ACBClientPhone requestMicrophoneAndCameraPermission`. This method delegates to the iOS permission APIs, and you should typically call it before making or receiving calls. The first time you call this method, it displays an individual alert for each requested permission. Subsequent calls do not display an alert unless you have reset your privacy settings in iOS Settings.

When subsequently making or receiving a call, the iOS SDK checks whether the user has given the necessary permissions. For example, if you make an audio-only outgoing call, the end user only needs to have granted permission to use the microphone; if you want to receive an incoming audio and video call, the end user needs to have granted permission to use the microphone and camera.

If you attempt to make or answer a call with insufficient permissions, the application receives the optional `ACBClientCallDelegate didReceiveCallRecordingPermissionFailure` callback method, and the call ends.

**Note** The keys `NSCameraUsageDescription` and `NSMicrophoneUsageDescription` in your app `plist` file provide (part of) the text of the alert when the user is asked for permission to use the camera and microphone. On iOS 10 and higher, these keys are mandatory, and your application will fail if you do not provide them. See iOS SDK documentation for details.

## Making a Call

In the following example, the application makes a call (using `createCallToAddress` on the `ACBClientPhone` object) as soon as the session has started (see **Initializing the ACBUC Object on the previous page**):

```
- (void) ucDidStartSession:(ACBUC *)uc
{
   ACBClientPhone* phone = uc.phone;
   phone.delegate = aPhoneDelegate;
   phone.previewView = previewView;
   ACBClientCall* call = [phone createCallToAddress:calleeAddress
   withAudio:ACBMediaDirectionSendAndReceive
   withVideo:ACBMediaDirectionSendAndReceive delegate:aCallDelegate];
   call.videoView = aVideoView;
}
```

You can change the values of the `withAudio` and `withVideo` parameters to make the call as audio-only or video-only. Valid values are:

- `ACBMediaDirectionNone`
- `ACBMediaDirectionSendOnly`
- `ACBMediaDirectionReceiveOnly`
- `ACBMediaDirectionSendAndReceive`

**Note** The older form, `createCallToAddress:audio:video:delegate:`, which took two boolean values, is now deprecated.

# Receiving a Call

REM Advanced SDK invokes the `ACBClientPhoneDelegate didReceiveCall:` delegate method when it receives an incoming call. The application can answer the incoming call by calling its `answerWithAudio:andVideo:` method:

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
   [call setVideoView:videoView];
   [call answerWithAudio:ACBMediaDirectionSendAndReceive
   andVideo:ACBMediaDirectionSendAndReceive];
}
```

To reject the call, use `[call end]`.

You can change the values of the parameters to answer the call as audio-only or video-only. Valid values are:

- `ACBMediaDirectionNone`
- `ACBMediaDirectionSendOnly`
- `ACBMediaDirectionReceiveOnly`
- `ACBMediaDirectionSendAndReceive`

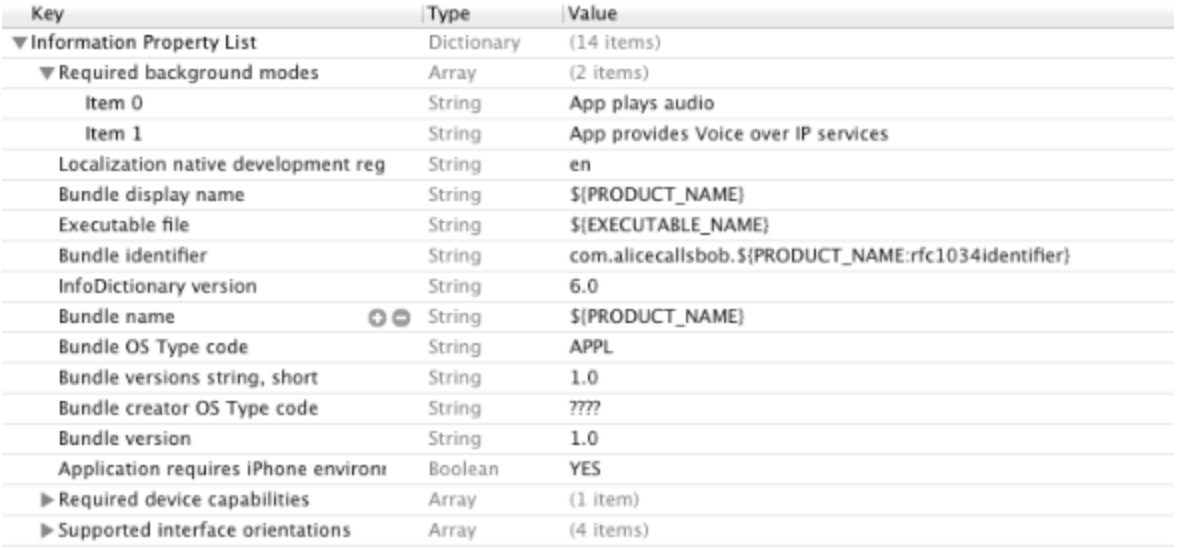
**Note**

- The older form, `answerWithAudio:video:`, which took two boolean values, is now deprecated.
- The audio and video options specified in the answer affect both sides of the call; that is, if the remote party placed a video call and the local application answers as audio only, then neither party sends or receives video.
- If your application plays its own ringing tone, please note that the iOS SDK makes calls to the `AVAudioSession sharedInstance` object when establishing a call. For this reason, we recommend waiting until you receive a call status of `ACBClientCallStatusRinging` (from `ACBClientCallDelegate didChangeStatus`) before calling `AVAudioSession sharedInstance` methods.

## Receiving Calls when the Client is in Background or Suspended Mode

If you require the application to continue receiving calls when in background or suspended mode, you need to add the following values to the **Required background modes** key in the application's `plist` file:

- App plays audio
- App provides Voice over IP services



| Key | Type | Value |
|---|---|---|
| Information Property List | Dictionary | (14 items) |
| Required background modes | Array | (2 items) |
| Item 0 | String | App plays audio |
| Item 1 | String | App provides Voice over IP services |
| Localization native development reg | String | en |
| Bundle display name | String | ${PRODUCT_NAME} |
| Executable file | String | ${EXECUTABLE_NAME} |
| Bundle identifier | String | com.alicecallsbob.${PRODUCT_NAME:rfc1034identifier} |
| InfoDictionary version | String | 6.0 |
| Bundle name | String | ${PRODUCT_NAME} |
| Bundle OS Type code | String | APPL |
| Bundle versions string, short | String | 1.0 |
| Bundle creator OS Type code | String | ???? |
| Bundle version | String | 1.0 |
| Application requires iPhone environ | Boolean | YES |
| Required device capabilities | Array | (1 item) |
| Supported interface orientations | Array | (4 items) |

## Video Views and Preview Views

In order to show video during a call, the application can set the `videoView` on the `ACBClientCall` object, and the `previewView` on the `ACBClientPhone` object. Each property is an `ACBView` object (which for iOS is equivalent to a `UIView`).

The `videoView` is used to render the remote party's video stream and is mandatory for a two-way video call. The `previewView` is an optional addition that renders the local party's video stream as it is being captured; this is the same stream that the remote party receives.

Initializing the `videoView` and `previewView` is optional and can be done at any time. If there are calls in progress when the application sets the properties, the changes take effect when the next video call is made.

When there is no video stream being sent or received, the `videoView` and `previewView` do not render any frames; video is only displayed when streaming.

## Ending a Call

If the user ends the call, the client application should call the `ACBClientCall` object's `end` method.

To receive notification that the remote party has terminated the call, the application must monitor the state of the call (see **Monitoring the State of a Call on page 71**) for the `ACBClientCallStatusEnded` state.

## Muting the Local Audio and Video Streams

During a call the application can mute or unmute one, or both, of the local audio and video streams. Muting the stream stops that stream being sent to the remote party; however, the user still receives any stream that the remote party sends.

To mute either stream, use one, or both, of the `enableLocalAudio` and `enableLocalVideo` methods of the call:

```
- (void) muteButtonPressed:(UIButton*)button
{
   [self.call enableLocalAudio:NO];
   [self.call enableLocalVideo:NO];
}
```

Each method takes a single boolean parameter. To restore media, call `enableLocalVideo` or `enableLocalAudio` with the parameter `YES`.

# Holding and Resuming a Call

During a call the application can put a call on hold (for example, in order to make or receive another call). Placing the call on hold pauses both the stream sent by the user and the stream sent by the remote party; only the party who placed the call on hold can resume it.

```
- (void) holdButtonPressed:(UIButton*)button
{
   [call hold];
}
- (void) resumeButtonPressed:(UIButton*)button
{
   [call resume];
}
```

# DTMF Tones

Once a call is established, an application can send DTMF tones on that call by calling the `playDTMFCode` method of the `ACBClientCall` object:

```
[call playDTMFCode:@"#123*" localPlayback:YES];
```

- The first parameter can either be a single tone, (for example, `6`), or a sequence of tones (for example, `#123,*456`). Valid values for the tones are those characters conventionally used to represent the standard DTMF tones: `0123456789ABCD#*`.

**Note** The comma indicates that there should be a two second pause between the `3` and the `*` tone.

- The second parameter is a boolean which indicates whether the tone is played back locally.

# Handling Multiple Calls

Applications developed with Remote Expert iOS SDK do not support multiple simultaneous calls:

# Setting Video Resolution

The Remote Expert Mobile supports configuring the captured, and therefore sent, video resolution for video calls. The application can select one of a set of video resolutions, and apply it to the capture device. It can also configure the frame rate for capture. When it specifies a resolution and frame rate, REM Advanced SDK makes every effort to match those values where hardware allows.

## Enumerating the Possible Resolutions

The application can get a list of possible resolutions from the `ACBClientPhone` object using the `recommendedCaptureSettings` method:

```
NSArray* recommendedSettings = [uc.phone recommendedCaptureSettings];
```

The array returned by this method contains an `ACBVideoCaptureSetting` object for each recommended setting. Each `ACBVideoCaptureSetting` specifies a resolution and a recommended frame rate for that resolution.

The supported resolutions are:

| Enumeration Value | Width | Height | Frame Rate |
|---|---|---|---|
| `ACBVideoCaptureResolution352x288` | `352` | `288` | `20` or `30` depending on device - see table below. |
| `ACBVideoCaptureResolution640x480` | `640` | `480` | `30` |
| `ACBVideoCaptureResolution1280x720` | `1280` | `720` | `30` |

The maximum resolution and frame rate available on each iOS device are as shown in the table below.

| Device Type | Maximum Resolution | Maximum Frame Rate |
|---|---|---|
| iPhone 4 and below<br>iPad 1 | No video support | |
| iPhone 4s<br>iPad 2<br>iPad 3<br>iPad mini | 352x288 | 20 |
| iPhone 5<br>iPhone 5c<br>iPhone 5s<br>iPad 4<br>iPad mini retina | 640x480 | 30 |
| iPad Air | 1280x720 | 30 |

If you set the resolution of frame rate to values higher than these, then the provided resolution or frame rate is the minimum of the requested value and the maximum value for the particular device.


**Important**

The table above is valid for the release 2.15 but may change in a future release.

## Setting the Resolution

The application can set the captured video resolution using the `preferredCaptureResolution` property of the `ACBClientPhone`object. The value supplied must be one of the resolutions presented in `recommendedCaptureSettings`, as described **Enumerating the Possible Resolutions on the previous page**

```
ACBVideoCaptureSetting chosenSetting = [recommendedSettings objectAtIndex:0];
uc.phone.preferredCaptureResolution = chosenSetting.resolution;
```

Alternatively, one of the values from the enumeration:

```
uc.phone.preferredCaptureResolution = ACBVideoCaptureResolution352x288;
```


**Note**

The video capture resolution only applies for the next call made with the phone object, and it does not affect calls currently in progress.

## Setting the Frame Rate

The application can set the captured video frame rate using the `preferredCaptureFrameRate` property of the `ACBClientPhone`object:

```
ACBVideoCaptureSetting chosenSetting = [recommendedSettings objectAtIndex:0];
uc.phone.preferredCaptureFrameRate = chosenSetting.frameRate;
```

Alternatively, it can attempt a custom frame rate:

```
uc.phone.preferredCaptureFrameRate = 20;
```

**Note** The video capture frame rate only applies for the next call made with the phone object, and it does not affect calls currently in progress.

# Dial Failures

REM Advanced SDK does not call the `ACBClientCallDelegate` failure methods (`didReceiveDialFailure` etc.) for failures caused by a timeout. This results in the client seeing the **Trying to call…** dialog, despite the call being inactive. To avoid this, handle these timeout errors using the status delegate methods; examples can be found in **Monitoring the State of a Call on page 71** and in particular to the callback:

```
(void) call:(ACBClientCall*)call didChangeStatus:(ACBClientCallStatus)status;
```

## Handling Device Rotation

The SDK automatically handles control of the video orientation.

**Note** The `setVideoOrientation` method of `ACBClientPhone` is now deprecated.

## Switching between the Front and Back cameras

By default, during video calls, REM Advanced SDK uses the front camera. The application can change this by calling the `setCamera` method of `ACBClientPhone`.

```
- (void) switchToBackCamera
{
   [self.phone setCamera:AVCaptureDevicePositionBack];
}
```

There are two parameters that can be passed to this method:

- `AVCaptureDevicePositionBack`
- `AVCaptureDevicePositionFront`.

These enumeration values are available by importing `<AVFoundation/AVCaptureDevice.h>`.

The camera setting persists between calls; if the back camera is enabled during a video call, the next video call will also use that camera.

The method can be called at any time; if there are no active video calls, the value takes effect when a video call is next in progress.

# Application Background Mode

When the user presses the **Home** button, presses the **Sleep/Wake** button, or the system launches another application, the foreground application transitions to the inactive state and then to the background state. If you are currently streaming video from your application, this is suspended when the application goes into background mode, and automatically resumes when the application returns to the foreground. Audio continues to be streamed when an application goes into background mode.

It is an application developer's responsibility to consider both functional and privacy implications, and decide whether their application should mute audio and video when transitioning to background mode (see **Muting the Local Audio and Video Streams on page 33**).

If you mute the video when in background mode, you must unmute in order to resume capture and streaming.


Note

The behavior of iOS is different to Android.

# Monitoring the State of a Call

A call transitions through several states, and the application can monitor these by assigning a delegate to the call:

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
   call.delegate = self;
    ⋮
}
```

Each state change fires the `call:didChangeStatus:` delegate method. As the outgoing call progresses toward being fully established, the application receives a number of calls to `didChangeStatus`, each time being passed one of the `ACBClientCallStatus` enumeration values.

Switching on the value of the enumeration allows the application to adjust the UI to give the user suitable feedback, for example by playing a local audio file for ringing or alerting:

```
- (void) call: (ACBClientCall*)call didChangeStatus:(ACBClientCallStatus)
status
{
   switch (status)
   {
      case ACBClientCallStatusRinging:
         [self playRingtone];
         break;
      case ACBClientCallStatusInCall:
         [self stopRinging];
         break;
      case ACBClientCallStatusEnded:
      case ACBClientCallStatusBusy:
      case ACBClientCallStatusError:
      case ACBClientCallStatusNotFound:
      case ACBClientCallStatusTimedOut:
         [self updateUIForEndedCall];
         break;
      default:
         break;
   }
}
```

The following table gives the possible status codes:

| Status code | Meaning |
|---|---|
| `ACBCallStatusSetup` | Call is in process of being set up |
| `ACBCallStatusAlerting` | The call is an incoming one which is alerting (ringing) |
| `ACBCallStatusRinging` | An outgoing call is ringing at the remote end |
| `ACBCallStatusMediaPending` | The call is connected, and waiting for media |
| `ACBCallStatusInCall` | The call is fully set up, including media |
| `ACBCallStatusBusy` | Dialed number is busy |
| `ACBCallStatusNotFound` | Dialed number is unreachable or does not exist |

| Status code | Meaning |
|---|---|
| ACBCallStatusTimedOut | Dialing operation timed out without a response from the dialed number |
| ACBCallStatusError | An error has occurred on the call. such the media broker reaching its full capacity, the network terminating the request, or there being no media. |
| ACBCallStatusEnded | The call has ended |

# Adding Application Event Distribution

The application initially accesses the API via a single object, ACBUC, from which other objects can be retrieved. ACBUC has an attribute named aed, which is the starting point for all *Application Event Distribution* operations.

To create an AED application, you need to:

1. Create an instance of ACBTopicDelegate and implement the callback methods.
2. Access the aed attribute to create and/or connect to an ACBTopic, supplying the delegate from the previous step.
3. Call methods on the topic object to change data on the topic.
4. Disconnect from the topic when you no longer want to receive AED notifications.

## Creating and Connecting to a Topic

The application can create a topic using the createTopicWithName:delegate: method on the AED object:

```
ACBTopic* topic = [uc.aed createTopicWithName:@"name" delegate:topicDelegate];
```

or the createTopicWithName:expiryTime:delegate: method:

```
ACBTopic* topic = [uc.aed createTopicWithName:@"name" expiryTime:5
delegate:topicDelegate];
```

The name of the topic is an NSString, and the expiryTime parameter is a time in minutes. A topic created with an expiry time will be automatically removed from the server after the topic has been inactive for that time. When created without an expiry time, the topic exists indefinitely, and has to be explicitly deleted. The delegate is an object conforming to the ACBTopicDelegate protocol.

**Important**

Either of these creates a client-side representation of a topic and automatically connects to it. If the topic already exists on the server, it will connect to that topic; if the topic does not already exist, it creates it.

### didConnectWithData

After connecting to the topic, the delegate will receive a didConnectWithData callback. (In the case of failure, it will receive a didNotConnectWithMessage callback with a message parameter (an NSString).) The didConnectWithData callback has a single data parameter containing all the data currently associated with the topic.

The data parameter is an NSDictionary which contains a value with the key data, which is an NSArray of NSDictionary objects, each of which contains a single data item with members called key and value. The application can iterate through the data items to display them to the newly connected user:

```
- (void)topic:(ACBTopic*)topic didConnectWithData:(NSDictionary*)data
{
   //topic data is an array containing all our key/value pairs
   NSArray *topicData = [data objectForKey:@"data"];
```

```
    if([topicData count] > 0)
    {
       //we can show our users the data in the topic as follows
       for(int i = 0; i < [topicData count] ; i++)
       {
          NSString* keyField = [[topicData objectAtIndex:i] valueForKey:@"key"];
          NSString* valueField = [[topicData objectAtIndex:i]
          valueForKey:@"value"];
          // Display key and value
       }
    }
}
```

# Disconnecting from a Topic

You can either disconnect from the topic without destroying it:

```
[topic disconnectWithDeleteFlag:FALSE];
```

or delete the topic from the server, which will also disconnect any other subscribers:

```
[topic disconnectWithDeleteFlag:TRUE];
```

When you delete the topic by calling `disconnectWithDeleteFlag:TRUE`, you will receive a `didDeleteWithMessage` callback, followed by a `topicDidDelete` callback.

## topicDidDelete

All clients connected to the topic receive a `topicDidDelete` callback when the topic is deleted from the server, either as a result of any client deleting it, or as a result of the topic expiring on the server (see **Creating and Connecting to a Topic on the previous page** for details of topic expiry). Once a topic has been deleted, the client should not call any of that topic's methods (which will fail in any case), and should consider itself unsubscribed from that topic. If a topic with the same name is subsequently created, it is a new topic, and the client will not be automatically subscribed to it.

# Publishing Data to a Topic

Once the application has connected to a topic, it can publish data on it. Data consists of name-value pairs:

```
[topic submitDataWithKey @"key_one" value:@"value"];
```

Having submitted the data, the delegate receives either a `didSubmitWithKey` or (in the case of failure) a `didNotSubmitWithKey` callback. Both callbacks contain the `key` and `value` which were submitted (successfully or unsuccessfully). The `didNotSubmitWithKey` callback also contains a `message` parameter giving more details of the failure; the `didSubmitWithKey` callback also contains a `version` parameter, an incrementing value which enables the application to check if the data it has just sent is the latest on the server.

In the case of a successful submission, the delegate also receives a `didUpdateWithKey` callback.

## didUpdateWithKey

A client receives a `didUpdateWithKey` callback when any client connected to the topic makes a change to a data item on that topic. The callback contains the `key`, `value`, and `version` parameters detailed previously (`value` contains the new value), and an additional `deleted` parameter, which will be `TRUE` if the data item has been deleted from the server (see **Deleting Data from a Topic below**).

# Deleting Data from a Topic

The client can delete the data item from the topic by calling:

```
[topic deleteDataWithKey:@"key_one"];
```

The delegate will receive either a `didDeleteDataSuccessfullyWithKey` callback (containing the `key` and `version`) or a `didNotDeleteDataWithKey` callback (containing a `message` indicating the cause of failure).

All clients subscribed to the topic will also receive a `didUpdateWithKey` callback, with the `deleted` parameter set to `TRUE`.

## Sending a Message to a Topic

A client application can send a message to a topic and have that message sent to all current subscribers:

```
[topic sendAedMessage:@"message to send"];
```

If it is successful, the delegate will receive a `didSendMessageSuccessfullyWithMessage` callback followed by a `didReceiveMessage` callback, both containing the message in the `message` parameter; if it is not successful, the delegate will receive a `didNotSendMessage` callback, containing an `originalMessage` and a `message` parameter.

### didReceiveMessage

The delegate will receive a `didReceiveMessage` callback whenever any connected client (including itself) sends a message to the topic. The only parameter is the `message` parameter (containing the text of the sent message) itself.

# CallKit Integration

CallKit is a framework which allows iOS apps with VOIP (Voice over Internet Protocol) functionality to integrate better with the native iOS user interface. This improves usability by allowing users to answer calls from the lock screen, and to use the native user interface to make calls. You can integrate CallKit with REM Advanced SDK applications by creating a `CXProvider` and a `CXCallController`, and calling their methods when REM Advanced SDK receives, makes, or ends a call.


**Note**

We recommend that in a production application you should separate the CallKit functionality from the REM Advanced SDK by putting it in its own class which contains and instantiates the `CXProvider` and `CXCallController` objects, and acts as the `CXProviderDelgate`. For simplicity (to avoid having to show the initialization of the object and other boilerplate code), this is not shown here; refer to the sample application for details.

## Making a Call

After creating a call in the usual way with REM Advanced SDK (see **Making a Call on page 31**), the application needs to:

1. Allocate a UUID to act as a call identifier in CallKit.
2. Create a handle for the remote address.
3. Create a `CXCallUpdate` and set any attributes., particularly the `remoteHandle` for the call
4. Call the `reportOutgoingCallWithUUID` method on the `CXProvider`
5. Create a transaction by calling the `requestTransaction` method of the `CXCallController` object, to inform CallKit of the start of the call.


**Note**

The application should save the UUID and the `ACBClientCall` object for use during the call and when the call ends.

```
- (void) makeCall:(ACBPhone *)phone
{
```

```
    ACBClientCall* call = [phone createCallToAddress:calleeAddress
    withAudio:ACBMediaDirectionSendAndReceive
    withVideo:ACBMediaDirectionSendAndReceive delegate:aCallDelegate];
    callid = [[NSUUID alloc] init];
    CXHandle* handle = [[CXHandle alloc] initWithType:CXHandleTypePhoneNumber
    value:call.remoteAddress];
     ⋮
    CXCallUpdate* update = [[CXCallUpdate alloc] init];
    update.remoteHandle = handle;
    update.hasVideo = @TRUE;
    NSDate* now = [[NSDate alloc] init];
    [provider reportOutgoingCallWithUUID:callid startedConnectingAtDate now];
     ⋮
    CXStartCallAction* cxCSA = [[CXStartCallAction alloc]
    initWithCallUUID:callid handle:handle];
    CXTransaction* txn = [[CXTransaction alloc] initWithAction: action];
    [controller requestTransaction:txn completion:errorFunction];
    [action fulfilWithDateStarted: now];
}
```

## Receiving a Call

When the application is notified by REM Advanced SDK of an incoming call, it should:

1. Allocate a UUID to act as a call identifier in CallKit.
2. Create a `CXCallUpdate` and set any attributes., particularly the `remoteHandle` for the call.
3. Call the `reportNewIncomingCallWithUUID` method of the `CXProvider` object.
4. Answer the call (using the `answerWith Audio WithVideo` method of the `ACBClientCall` object) when CallKit calls the `performAnswerCallAction` of the `CXProviderDelegate` associated with the `CXProvider` object.

**Note** The application should save the UUID and the `ACBClientCall` object for use during the call and when the call ends.

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
     ⋮
    callid = [[NSUUID alloc] init];
    CXCallUpdate* update - [[CXCallUpdate alloc] init];
    update.remoteHandle = [[CXHandle alloc] initWithType:CXHandleTypePhoneNumber
    value:call.remoteAddress];
    update.hasVideo = call.hasRemoteVideo;
    [provider reportNewIncomingCallWithUUID: callid update:update
    completion:errorFunction];
    currentCall = call;
}
```

And on the `CXProviderDelegate`:

```
- (void) provider: (CXProvider*) provider performAnswerCallAction:
(CXAnswerCallAction*) action
{
    [currentCall answerWithAudio:ACBMediaDirectionSendAndReceive
    andVideo:ACBMediaDirectionSendAndReceive];
    NSDate* now = [[NSDate alloc] init];
```

```
    [action fulfillWithDateConnected:now];
}
```

## Ending a Call

When REM Advanced SDK notifies the application that the call has ended (see **Monitoring the State of a Call on page 71**), the application needs to:

1. Call the `reportCall` method of the `CXProvider` with the callid of the call and an appropriate reason.
2. End the call.

```
- (void) call: (ACBClientCall*)call didChangeStatus:(ACBClientCallStatus)
status
{
    ⋮
   switch (status)
   {
       ⋮
      case ACBClientCallStatusEnded:
         NSDate* now = [[NSDate alloc] init];
         [provider reportCallWithUUID: callid endedAtDate:now
         reason:CXCallEndedReasonRemoteEnded];
         [call end];
         [currentCall end];
         break;
       ⋮
      default:
         break;
   }
    ⋮
}
```

The application should pass different values from the `CXCallEndedReason` enumeration to `reportCallWithUUID`, depending on the `status` value passed to `didChangeStatus`; the processing of `status` values which indicate that the call has ended is otherwise identical. (Some `ACBClientCallStatus` values do not indicate that the call has ended, and the application should process these differently.)

# Threading

All method invocations on the SDK, even to access read-only properties, must be made from the same thread. This can be any thread, and not necessarily the main thread of the application. Internally, the SDK may use other threads to increase responsiveness, but any delegate callbacks will occur on the same thread that is used to initialize the SDK.

# Self-Signed Certificates

If you are connecting to a server that uses a self-signed certificate, you need to add that certificate and the associated CA root certificate to the keychain on your client.

You can obtain the server certificate and CA root certificate through the REAS Administration screens. The *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html) explains how to view and export certificates. You need to extract the HTTPS Identity Certificate (server certificate) and the Trust Certificate (CA root certificate) that has signed your server certificate.

Once you have exported and downloaded the two certificates, you need to copy them to your client. Please follow the user documentation for your device to install the certificates.

You should then view the installed server certificate through the appropriate tool (**iOS Settings->General->Profiles** or **OSX Keychain**) and confirm that the server certificate is trusted. If it is, then your application should connect to the server.

Alternatively, you can use the `acceptAnyCertificate` method of the `ACBUC` object before calling `startSession`, although this should only be used during development:

```
ACBUC* uc = [ACBUC ucWithConfiguraton:sessionId stunServers:stunServers
delegate:self];
[uc acceptAnyCertificate:TRUE];
[uc startSession];
```

# Testing IPv6

Apple require that apps submitted to the Apple store support IPv6-only networks, and you should test this during development; see:

https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/UnderstandingandPreparingfortheIPv6Transition/UnderstandingandPreparingfortheIPv6Transition.html

Neither Media Broker nor REAS support IPv6 directly; however, you can configure Media Broker to give an IPv6 public address to the client, and then you can access both REAS and Media Broker through a NAT64 router. Apple laptops support providing a NAT64 Wi-Fi hotspot, as long as you are able to connect to your network through another interface such as an Ethernet cable - for details on enabling this, see the **Test for IPv6 DNS64/NAT64 Compatibility Regularly** section in the above link.

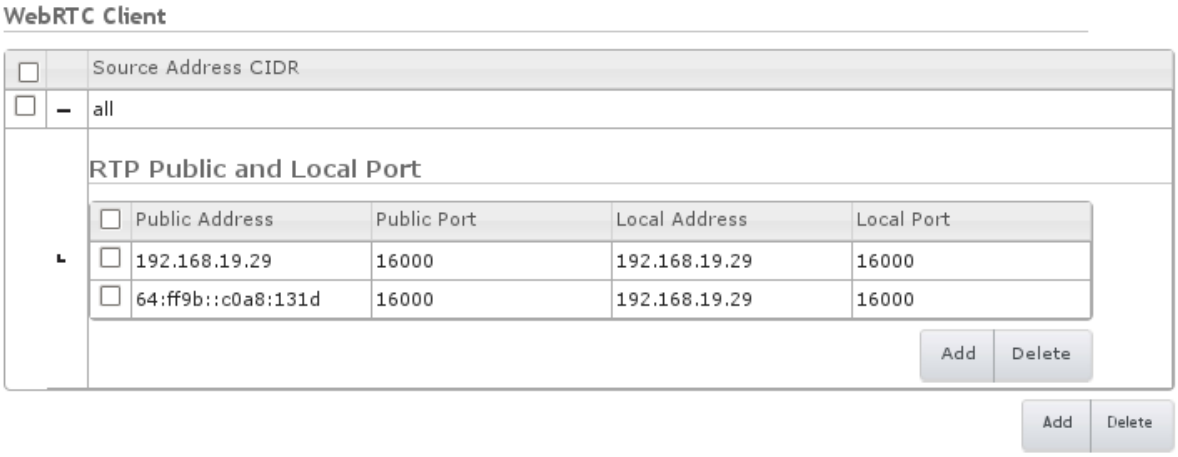
To configure Media Broker to give IPv6 addresses to the client, edit the Media Broker's settings:

1. In the configuration console, expand **WebRTC Client** settings
2. For each of the current public addresses click add, then enter an IPv6 equivalent in the public address.

   If using an Apple laptop hotspot, then the IPv6 address equivalent starts with

   ```
   64::ff9b::
   ```

   and is followed by the hexadecimal version of the IPv4 address. For example `c0a8:131d` is the equivalent of `192.168.19.29`




3. Duplicate the three other fields from the IPv4 port and address.


Note

Apple sometimes require testing an app in full during submission, in which case a public NAT64 is required - contact support for details on how to implement this.

# Bluetooth Support

The user can set the active audio device (speaker and microphone) for an iOS device, and REM Advanced SDK calls will use this setting by default. However, this behavior may not be appropriate while an REM Advanced SDK application is running; and in particular, the default behavior does not allow the call to switch to an alternative device if the active device fails (a particular problem with Bluetooth devices). The application can override the default behavior using the `ACBAudioDeviceManager` class; a single instance of this class is available on the `ACBClientPhone` object which controls the call. While this is in use, the application can:

- Define which audio output on the phone should handle the audio
- Define a default audio output on the phone, which will handle the audio if the preferred device is interrupted.
- Get a list of available audio outputs on the phone
- Determine which of the phone's audio outputs currently handles the audio


Note

This class has been added specifically to support the use of Bluetooth headsets, and we expect this to be its main use; accordingly, the examples assume that this is how it is being used. However, an application could also use this class to manage the audio output to the speakerphone, the internal speaker, and an external headphone set, and to explicitly *exclude* the use of Bluetooth headsets with the calls made by the application.

## Starting and Stopping ACBAudioDeviceManager

In order to use the methods on the `ACBAudioDeviceManager`, the application must first call the `start` method of the instance in the `ACBClientPhone` which is handling the call:

```
[uc.phone.audioDeviceManager start];
```

An appropriate place to do this is during initialization of the object which is to control the call.

Once the `ACBAudioDeviceManager` has been started, the application can call its methods to set the audio devices which the phone should use for calls made or received by the application. Calls which are not handled by the REM Advanced SDK application will be unaffected, and will use the phone's default behavior.

In order to return to the iOS device's default behavior without ending the call, the application can call `stop`:

```
[uc.phone.audioDeviceManager stop];
```


Note

While the audio device manager is active the application ***must not*** call the `setCategory` method of the call's `AVAudioSession` object. Doing so can cause unexpected behavior.

## Setting the Preferred Device

The application can set the preferred device for the call:

```
[uc.phone.audioDeviceManager setAudioDevice:
(ACBAudioDevice*) ACBAudioDeviceBluetooth];
```

The argument to the method must be one of the members of the `ACBAudioDevice` enumeration:

- `ACBAudioDeviceSpeakerphone`

  Audio will be sent to the loudspeaker in the phone, and will be audible to others in the vicinity. Audio input will be from the phone's internal microphone.

- `ACBAudioDeviceWiredHeadset`

  Audio will be sent to a device attached to the jack in the phone. If this device has a microphone, that will be used for audio input.

- `ACBAudioEarpiece`

  Audio will be sent to the internal speaker, and received from the internal microphone. The user will have to hold the phone to their ear during the call.

- `ACBAudioDeviceBluetooth`

  Audio will be sent to and received from a paired Bluetooth device.

- `ACBAudioDeviceNone`

  The application has no preference, and will accept the default behavior of the iOS device.

If the preferred device is available when the application calls `setAudioDevice`, the call will start using that device; if it is not available, there will be no immediate change, but if it later becomes available (the Bluetooth device is switched on or is otherwise recognized), then the audio will be switched to this device.

## Setting the Default Device

The application can set a fallback device in case the preferred device is unavailable:

```
[uc.phone.audioDeviceManager setDefaultDevice:
(ACBAudioDevice*) ACBAudioDeviceEarpiece];
```

The argument is one of the values from the `ACBAudioDevice` enumeration (see **Setting the Preferred Device on the previous page**).

Setting the default device establishes a fallback option in case the preferred device is temporarily unavailable. A common use would be:

```
[uc.phone.audioDeviceManager setAudioDevice:
(ACBAudioDevice*) ACBAudioDeviceBluetooth];

[uc.phone.audioDeviceManager setDefaultDevice:
(ACBAudioDevice*) ACBAudioDeviceEarpiece];
```

which would establish the Bluetooth headset as the preferred device, with the normal phone internal speaker and microphone as a fallback. With these settings in operation:

1. The call starts, but no Bluetooth headset is available. The call is sent to the internal speaker and microphone.
2. The Bluetooth headset is switched on. The phone switches the audio to the headset, and the user can put the phone down and continue the call.
3. The headset fails (perhaps the battery becomes too low). The application switches the call back to the internal speaker and microphone.
4. The user switches on another (fully powered) Bluetooth headset and pairs it with the phone. The audio switches to the new headset and the call continues on that device.

If the default device is also unavailable, the audio will be sent to whatever has been set as the active device on the phone (that is, it will fallback to the iOS default behavior).

## Listing Available Devices

The application can get a list of available audio devices by calling the `audioDevices` method:

```
NSMutableArray* devices = [uc.phone.audioDeviceManager audioDevices];
```

The resulting array contains members of the `ACBAudioDevice` enumeration, taken from the available inputs known to the `AVAudioSession`.

It can also find which device is currently set as the preferred audio device:

```
ACBAudioDevice* device = [uc.phone.audioDeviceManager selectedAudioDevice];
```

This will work whether the preferred device has been set explicitly (using `setAudioDevice`) or not.

# Responding to Network Issues

As the iOS SDK is network-based, it is essential that the client application is aware of any loss of connection. Remote Expert Mobile does not dictate how you implement network monitoring; however, the sample application uses the `SystemConfiguration` framework.

Depending on the nature of the issues with the network, the client application should react differently.

## Reacting to Network Loss

In the event of network connection problems, the SDK automatically tries to re-establish the connection. It will make seven attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to the `willRetryConnectionNumber:in:]` on the `ACBUCDelegate` precedes each of these attempts. The callback supplies the attempt number (as an `NSUInteger`) and the delay before the next attempt (as an `NSTimeInterval`) in its two parameters.

When all reconnection attempts are exhausted, the `ACBUCDelegate` receives the `ucDidLoseConnection` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in **Creating the Session on page 4**.

If any of the reconnection attempts are successful, the `ACBUCDelegate` receives the `ucDidReestablishConnection` callback.

Note that both the `willRetryConnectionNumber` and `ucDidReestablishConnection` are optional, so the application may choose to not implement them. The connection retries are attempted regardless.

**Note** The retry intervals, and the number of retries attempted by the SDK are subject to change in future releases. Do not rely on the exact values given above.

## Reacting to Network Changes

If the issues with the network are caused by a temporary loss of connectivity (for example, when moving between two Wi-Fi networks, or from a Wi-Fi network to a cellular data connection), the client application should not log out from the session and log back in (as described in **Reacting to Network Loss on page 73**) as the current session is lost.

To avoid this situation, the client application should register with iOS to receive notification of changes in network reachability. When iOS notifies the client application that the network has changed, the application should pass these details to the `ACBUC` instance.

When the client application starts, it should check for network reachability. When the network is reachable, the application calls `ACBUC setNetworkReachable:YES`; until this call is made, the application does not attempt to create a session.

If the network reachability drops after a session has been established, the client application needs to call `ACBUC setNetworkReachable:NO`.

If the network reachability changes from a cellular data connection to a Wi-Fi network, or *vice versa*, the client application needs to call `ACBUC setNetworkReachable:NO` followed by `ACBUC setNetworkReachable:YES` to disconnect from the first network and re-register on the second.

## Network Quality Callbacks

The application can implement the `didReportInboundQualityChange` callback on the `ACBClientCallDelegate` object to receive callbacks on the quality of the network during a call:

```
- (void) call:(ACBClientCall*)call didReportInboundQualityChange:
(NSUInteger)inboundQuality
{
   // Show indication of quality
}
```

The `inboundQuality` parameter is a number between `0` and `100`, where `100` indicates perfect quality. The application might choose to show a bar in the UI, the length of the bar indicating the quality of the connection.

The SDK starts collecting metrics as soon as it receives the remote media stream; that is, when it first fires the `callDidAddRemoteMediaStream:` delegate callback for the call. It does this every 5s, so the first quality callback fires roughly 5s after the remote media stream callback has been established.

The callback then occurs every time a different quality value is calculated, so if the quality is perfect then there is an initial quality callback with a value of `100` (after 5s), and then no further callback until the quality degrades.

# API Changes

| API Release | Description |
|---|---|
| 11.6 | Added ACBAudioDeviceManager to support Bluetooth devices |
| 11.5 | Added `UC.checkBrowserCompatibility(pluginInfoCallback)` |
| 10.6 | Sample modified to build and run on Xcode 7/iOS 9. |
| | Added `ACBDevice` class to provide access to recommended resolutions without requiring an `ACBClientPhone` instance. |
| | Support for arm64.<br>Extra libraries are now required at run time – see **Setting up a Project on page 28**. |
| | Added two new optional callbacks - `didReceiveDialFailureWithError` and `didReceiveCallFailureWithError` - to the `ACBClientCallDelegate` that pass a `NSError` object (containing `error code`, `reason`, etc.) to the application.<br>These supersede the existing `didReceiveDialFailure` and `didReceiveCallFailure` callbacks, which have now been deprecated. |
| | Support for requesting permission to use the microphone and camera, and ensuring calls can only be made/received if sufficient permissions have been granted. |
| | Support for self-signed certificates – see **Self-Signed Certificates on page 72** |
| | Extra callbacks during network loss. See **Reacting to Network Loss on page 73** :<br>`(ACBUC*)uc willRetryConnectionNumber:(NSUInteger)attemptNumber in: (NSTimeInterval)delay]` |
| | Added ability to determine inbound video quality.<br>Information about the quality of inbound video is available through the Call delegate:<br>`- (void) call:(ACBClientCall*)call didReportInboundQualityChange: (NSUInteger)inboundQuality;` |
| | `ACBClientCall`<br>`(void) hold;`<br>`(void) resume;`<br>`The hold and resume methods on the call object only attempt to hold` |

| API Release | Description |
|---|---|
| | or resume the call once the call has been established i.e. ACBClientCallStatusInCall. |
| | Added ability to set the video capture resolution.<br>The list of possible resolutions is available from the ACBClientPhone object via the recommendedCaptureSettings method:<br>NSArray* recommendedSettings = [uc.phone recommendedCaptureSettings];<br><br>Set the resolution using uc.phone.preferredCaptureResolution<br>Set the frame rate using uc.phone.preferredCaptureFrameRate |
| | The API now offers hold and resume methods on the call object. |
| | ACBClientAED<br>(ACBTopic *)createTopic:( NSString *)topicName<br>Replaced with two methods:<br>■ (ACBTopic *)createTopicWithName:(NSString *)topicName delegate: (id<ACBTopicDelegate>)delegate;<br>■ (ACBTopic *)createTopicWithName:(NSString *)topicName expiryTime:(int)expiryTime delegate: (id<ACBTopicDelegate>)delegate;<br>**Note:** The topic expiry time is set in minutes. |
| | ACBClientConversation<br>@property (readonly) NSString* contact;<br>Replaced with:<br>@property (readonly) NSString* contactAddress; |
| | ACBClientContact<br>@property (readonly) NSString* contactId;<br>Now replaced with:<br>@property (readonly) NSString* name<br>@property (readonly) NSString* address |
| | ACBTopic<br>(void)topic:(ACBTopic *)topic didNotSendMessageWithError:(NSString *)messageError errorMessage:(NSString *)errorMessage;<br>Now replaced with:<br>(void)topic:(ACBTopic *)topic didNotSendMessage:(NSString *)originalMessage message:(NSString *)message; |
| | ACBTopic<br>(void) connectWithTimeout:(int)timeout;<br>(void) connect;<br>Both methods have been removed, the functionality is now within ACBClientAED. |

CHAPTER 5

# Creating an Android Client Application



Remote Expert Mobile enables you to develop Android applications offering users the following methods of communication:

- Voice and Video calling
- Application Event Distribution (AED).

Remote Expert Mobile provides you with an Android SDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

To develop Android applications using Remote Expert Mobile, your system will need to conform to the system requirements listed at http://developer.android.com/sdk/index.html.

Information about the minimum version of Android supported can be found in the *Release Notes*.

The Android API reference documentation, including a full list of available methods and their associated callbacks, is delivered in the `docs` directory. Open `index.html` to view the API documentation.

**Note** The structure of an Android application revolves around different Activity objects, and the sample code included with the REM Advanced SDK Android SDK shows a typical structure. In the sample code, there is a `LoginActivity`, which gets the session token from the server (see **Creating the Session on page 4**); a `Main` Activity, which creates the `UC` object and connects to the session (see **Creating the UC Object below**); and an `InCallActivity`, which makes and receives calls, and works with those calls while they are in progress. AED operations (see **Adding Application Event Distribution on page 57**) are centralized in the `AEDFragment` and `AEDTopicManager` classes. We recommend that Android applications which make use of the REM Advanced SDK should separate their operations similarly; however, for simplicity this separation is not shown in the code snippets in the following section - see the sample code.

# Setting up a Project

**Note** Before following this process, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **Creating the Web Application on page 3**.

1. Create an Android Application project using either Android Studio or the Eclipse Android Developer Tools (ADT) plug-in.
2. Add `client-android-sdk.jar` to your project. This is found in the sample application's `libs` directory.

**Note** Only `client-android-sdk.jar` displays in the project; however it contains the following dependency jars and libraries:

- `libacbjnglpeerconnection.jar`
- `org.apache.http.legacy.jar`

3. Add the following to your `libs` folder:
   - `android-support-v4.jar`
   - `libacbjnglpeerconnection_so.so`

   The `android-support-v4.jar` is required if you want to write applications that use newer Android features but also support older devices which do not support those features by default.
4. In order to allow your project to access the required features on Android devices, include the following permissions in your `AndroidManifest.xml` file:
   - `android.permission.INTERNET`
   - `android.permission.RECORD_AUDIO`
   - `android.permission.CAMERA`
   - `android.permission.MODIFY_AUDIO_SETTINGS`

# Creating the UC Object

To set up all the functionality to which the user has access, the client application needs to obtain a session ID from your Web Application (see **Creating the Web Application on page 3**), and create a `UC` object using it. You create a `UC` object from a single object, `UCFactory`:

```
String sessionToken = getSessionToken();
UC uc = UCFactory.createUc(context, sessionToken, listener);
uc.setNetworkReachable(true);
uc.startSession();
```

In addition to the session token, `createUc` takes an `android.content.Context` object and an object which implements the `UCListener` interface. When the session starts, the listener receives an `onSessionStarted` callback.

**Note** If the application saves the session token in the instance state as soon as it is received from the Web Application, it can create or re-create the `UC` object as necessary in the `onCreate` method of the main `Activity`, even if the application has been put into the background before creating the `UC` object..

There is an alternative version of `createUc`, which takes a list of STUN servers in addition:

```
String sessionToken = getSessionToken();
List<String> stunServers = new List<String>();
stunServers.append("stun:stun.1.google.com:19302");
UC uc = UCFactory.createUc(context, sessionToken, stunServers, listener);
uc.setNetworkReachable(true);
uc.startSession();
```

**Note** STUN servers are not necessary if the Gateway is not behind a firewall, so that *Network Address Translation* is not needed. You can provide your own STUN server instead of the public Google one above; and you can provide more than one in the array, in which case they will be tried in sequence until REM Advanced SDK finds a working one.

# Adding Voice and Video

Once the application has created the `UC` object, it can be used to obtain an instance of the `Phone` and `AED` objects.

The application uses the `Phone` object to make or receive calls, for which it returns `Call` objects. Each key object in the API implements the listener pattern, that allows an application to be informed of the outcome of operations and other events.

The application can use the `AED` object to create AED Topics (see **Adding Application Event Distribution on page 57**).

## Making a Call

After the session starts (see **Creating the UC Object on the previous page**), the application can make a call using the `createCall` method of the `Phone` object:

```
Call call;

public void onSessionStarted()
{
   Phone phone = uc.getPhone();
   phone.addListener(this);
   call = phone.createCall(callee, audio, video, listener);
}
```

The `callee` parameter is a `String` containing the address to make the call to. `audio` and `video` are boolean parameters which indicate whether the call is to include audio and video streams. The `listener` parameter is an object implementing the `CallListener` interface, which receives notifications of events of interest on the call (such as when it is completely set up).

(The above code makes a call as soon as the session has started. More typically, the application would start a new `Activity` to gather the callee's number in response to the session starting, and another new `Activity` to make the call, once the callee's number was known. See the sample code included in the Android SDK for a more realistic architecture.)

## Receiving a Call

REM Advanced SDK invokes the `PhoneListener onIncomingCall` method when it receives an incoming call, passing a `Call` object as a parameter. The application can answer the incoming call by calling its `answer` method:

```
public void onIncomingCall(Call call)
{
   call.addListener(this);
   call.answer(audio, video);
}

public void onStatusChanged(Call call, CallStatus status)
{
   switch (status)
   {
       ⋮
      case IN_CALL:
         // Adjust UI
          ⋮
         break;
       ⋮
   }
}
```

`audio` and `video` are boolean parameters which indicate whether the call is to include audio and video streams. Note that the audio and video options specified in the answer will affect both sides of the call; that is, if the remote party placed a video call and the local application answers as video only, then neither party will send or receive audio. Adding a `CallListener` before answering the call allows the application to receive a notification when the call is completely set up, so that it can render video and audio streams.

To reject the call, call the `Call` object's `end` method.

## Video Views and Preview Views

In order to show video during a call, the application can call `setVideoView` on the `Call` object and `setPreviewView` on the `Phone` object. Both methods take a single `VideoSurface` parameter.

The application creates a `VideoSurface` using the `createVideoSurface` method of the `Phone` object, passing in the `android.content.Context`, the required dimensions (an `android.graphics.Point` object), and an object implementing `VideoSurfaceListener`. Initializing the `VideoSurface` is optional and can be done at any time, but typically the application would create video surfaces for the preview view and remote video view as soon as the `Phone` object is available, and then set them using the `Phone` object or a `Call` object:

```
void onCreate(Bundle savedInstanceState)
{
   // Restore information from instance state
    ⋮
   videoSurface = phone.createVideoSurface(this, videoSize, listener);
   previewSurface = phone.createVideoSurface(this, previewSize, listener);
   phone.setPreviewView(previewSurface);
   // Set up UI, cameras, etc.
    ⋮
}

void onStatusChanged(Call call, CallStatus status)
```

```
{
   switch (status)
   {
       ⋮
      case IN_CALL:
         // Show video
         call.setVideoView(videoSurface);
          ⋮
         break;
       ⋮
   }
}
```

The video view is used to render the remote party's video stream and is mandatory for a two-way video call. The preview view renders the local party's video stream as it's being captured; this is the same stream that the remote party will receive.

If there are calls in progress when the properties are set, the changes will take effect immediately and endure for future calls. If there are no active video calls, the change will take effect when a video call is next in progress.

When there is no video stream being sent or received, the video view and preview view render a full frame of green. Video appears only when there is video being streamed.

The application can set the camera to use as the local video source on the `Phone` object. See **Switching between the Front and Back Cameras on page 56**.

## Ending a Call

If the user ends the call, the client application should call the `Call` object's `end` method.

To detect that the remote party has ended the call, the client application must implement the `CallListener` interface in order to receive `onStatusChanged` notifications (see **Monitoring the State of a Call on page 56**).

**Note**

`CallListener.onRemoteMediaStream` will also be called at the end of a call (with argument `null`), as well as at the beginning of the call.

## Muting the Local Audio and Video Streams

During a call, the application can mute and unmute the local audio and video streams separately. Muting the stream stops that stream being sent to the remote party. The remote party's stream continues to play locally, however.

To mute either stream, use the `enableLocalAudio` and `enableLocalVideo` methods of the `Phone` object.

```
void onMuteButtonPressed()
{
   phone.enableLocalAudio(false);
   phone.enableLocalVideo(false);
}
```

To restore media, call the same method with the parameter set to `true`.

**Note**

- This will affect all calls and persists to subsequent calls. When a call starts, the streams will be muted as per the current `Phone` setting.
- Muting or unmuting a video stream in an audio-only call has no effect.

# Holding and Resuming a Call

During a call the application can put a call on hold (for example, in order to make or receive another call). Placing the call on hold pauses the stream sent by the user and the stream sent by the remote party; only the party who placed the call on hold can resume it.

```
void holdButtonPressed()
{
   call.hold();
}
void resumeButtonPressed()
{
   call.resume()
}
```

# Sending DTMF Tones

An application can send DTMF tones on the call by calling the `playDTMFCode` method on the `Call` object.

The first parameter to this call is a `String`, which can be either a single tone, (for example, `6`), or a sequence of tones (for example, `#123*456`). Valid values for the tones are those characters conventionally used to represent the standard DTMF tones: `0123456789ABCD#*`. A comma character inserts a two-second pause into a sequence of tones.

The second parameter should be `true` if you want the tones to be played back locally, so that the user of the application can hear them.

# Handling Multiple Calls

Applications developed with Remote Expert Android SDK do not support multiple simultaneous calls:

# Setting Video Resolution

The Remote Expert Mobile JavaScript SDK supports configuring the captured, and therefore sent, video resolution for video calls. The application can select one of a set of video resolutions, and apply it to the capture device. It can also configure the frame rate for capture. When it specifies a resolution and frame rate, REM Advanced SDK makes every effort to match those values where hardware allows.

## Enumerating the Possible Resolutions

The application can get a list of possible resolutions from the `Phone` object via the `getRecommendedCaptureSettings()` method:

```
List<PhoneVideoCaptureSetting> recommendedSettings =
phone.getRecommendedCaptureSettings();
```

The `List` returned by this method contains a `PhoneVideoCaptureSetting` object for each recommended setting. Each `PhoneVideoCaptureSetting` provides a resolution (from its `getResolution` method) and a recommended frame rate (from its `getFramerate` method) for that resolution.

The supported resolutions are:

| Enumeration Value | Width | Height |
|---|---|---|
| RESOLUTION_176x144 | 176 | 144 |
| RESOLUTION_352x288 | 352 | 288 |
| RESOLUTION_640x480 | 640 | 480 |
| RESOLUTION_960x720 | 960 | 720 |
| RESOLUTION_1280x720 | 1280 | 720 |


Note

The behavior on Android is different from iOS. On iOS, the SDK will not allow you to set the resolution to one that is not supported by the device. Due to the vast number of Android devices, we cannot know what devices can support a given resolution, and so the Android SDK allows the application to choose any supported resolution; there is, however, no guarantee that the phone will honor it.

## Setting the Resolution

The application can set the captured video resolution using the `setPreferredCaptureResolution` method of the `Phone` object. The single parameter is one of the `PhoneVideoCaptureResolution` enumeration:

```
phone.setPreferredCaptureResolution(PhoneVideoCaptureResolution.RESOLUTION_
640x480);
```

Alternatively, you can get it from a `PhoneVideoCaptureSetting` value (see **Enumerating the Possible Resolutions on the previous page**):

```
PhoneVideoCaptureSetting setting = phone.getRecommendedCaptureSettings().get
(0);
phone.setPreferredCaptureResolution(setting.getResolution());
```


Note

The video capture resolution will only apply to the next call made with the `Phone` object; it does not affect calls currently in progress.

## Setting the Frame Rate

The application can set the captured video frame rate using the `setPreferredCaptureFrameRate` method of the `Phone` object. It takes a single integer parameter:

```
phone.setPreferredCaptureFrameRate(20);
```


Note

The video capture frame rate will only apply for the next call made with the `Phone` object; it does not affect calls currently in progress.

# Handling Device Rotation

By default, when making video calls, REM Advanced SDK assumes the device is being held in portrait mode and is sending a portrait video stream. The application can change this by calling the `setVideoOrientation` method of the `Phone` object. The method takes a single parameter, which may be one of:

- `0`

  Equivalent to `Surface.ROTATION_0`
- `90`

  Equivalent to `Surface.ROTATION_90`

- `180`

  Equivalent to `Surface.ROTATION_180`

- `270`

  Equivalent to `Surface.ROTATION_270`

The orientation persists between calls; for example, if the orientation is set to `180` during a video call, the next video call will also have that orientation.

The method can be called at any time - if there are no active video calls, the value will take effect when a video call is next in progress.

The `setVideoOrientation` method affects the orientation of the stream being sent. If a preview window (which shows the stream being sent) is being used, it will also affect the orientation of the video being rendered in that view.

The `setVideoOrientation` method does not affect the orientation of the remote stream being rendered (assuming that the `videoView` property has been set). The orientation of the video view is handled automatically to match the orientation of the device.

## Switching between the Front and Back Cameras

By default, when making video calls, REM Advanced SDK uses the front-facing camera. The application can change this by calling the `setCamera` method on the `Phone` object, and passing the camera Id you wish to use - see the *Android API Guide* at http://developer.android.com/reference/android/hardware/Camera.html#open(int) for selecting the camera Id to use.

The camera setting persists between calls; that is, if you enable the rear-facing camera during a video call, the next video call will also use that camera.

The method can be called at any time; if there are no active video calls, the value will take effect when a video call is next in progress.

The REM Advanced SDK android sample app checks to see how many cameras there are on the device. If there is only 1 camera, it will use it, whether it is front-facing or back-facing. If there is more than 1 camera, it will use the first front-facing camera it can find. If there is more than 1 camera on the device, the sample app adds a **Camera Selection** menu to the **Options Menu** to allow the user to select between the front-facing and back-facing camera.

## Application Background Mode

When the user presses the **Home** button, presses the power button, or the system launches another application, the foreground application transitions to the inactive state and then to the background state. If you are currently streaming video from your application, this will continue when the application goes into background mode.

It is an application developer's responsibility to consider both functional and privacy implications, and decide whether their application should mute audio and video when transitioning to background mode.

**Note** The behavior of Android is different to that of iOS.

## Monitoring the State of a Call

During call setup, the call will transition through several states, from the initial setup to being connected with media available (or failure). You can monitor these states by setting the `CallListener` and implementing the `onStatusChanged` method.

Switching on the value of the `CallStatus` enumeration allows the application to adjust the UI to give the user suitable feedback, for example by playing a local audio file for ringing or alerting:

```
public void onStatusChanged(Call call, CallStatus status)
{
```

```
    switch (status)
    {
       case RINGING:
          playRingtone();
          break;
       case IN_CALL:
          stopRinging();
          break;
       case ENDED:
       case BUSY:
       case ERROR:
       case NOT_FOUND:
       case TIMED_OUT:
          updateUIForEndedCall();
          break;
       default:
          break;
    }
 }
```

You can also call the `getCallStatus` method on the `Call` object to get the status of the call.

The following table gives the possible status codes:

| Status Code | Meaning |
|---|---|
| `UNINITIALIZED` | The `Call` object has been created, but not initialized |
| `SETUP` | Call is in process of being set up |
| `ALERTING` | The call is an incoming one which is alerting (ringing) |
| `RINGING` | An outgoing call is ringing at the remote end |
| `MEDIA_PENDING` | The call is connected, and waiting for media |
| `IN_CALL` | The call is fully set up, including media |
| `BUSY` | Dialed number is busy |
| `NOT_FOUND` | Dialed number is unreachable or does not exist |
| `TIMED_OUT` | The dialing operation timed out without a response from the dialed number |
| `NO_MB_CAPACITY` | The media broker has reached its full capacity |
| `REQUEST_TERMINATED` | The request was terminated by the network |
| `TEMPORARILY_UNAVAILABLE` | Something necessary to set up the call was unavailable on the network |
| `MEDIA_UNAVAILABLE` | Unable to access media |
| `ERROR` | The call has errored |
| `ENDED` | The call has ended |

# Adding Application Event Distribution

The `UC` object also provides access to the `AED` object, which is the starting point for all *Application Event Distribution* (AED) operations.

An AED application will:

1. Access the `AED` object to create a `Topic` object
2. Add a `TopicListener` to the `Topic` object
3. Connect to the `Topic`
4. Call methods on the `Topic` object (apart from `disconnect`) to change data on the topic or send messages to other subscribers to the topic.
5. Disconnect from the topic when it no longer wants to receive notifications.

# Creating and Connecting to a Topic

The client application can create a topic using the `createTopic` method on the `AED` object. This call creates a client-side representation of a topic and automatically connects to it (for simplicity `this` implements the `TopicListener` interface):

```
uc.getAED().createTopic("my topic", this);
```

If it succeeds, the `TopicListener` will receive the `onTopicConnected` method. In the notification, you will receive a `Topic` object representing the topic, and a `Map` containing the data items (in the form of key-value pairs) which is currently associated with the topic. The application can add data, disconnect, send messages, and so on, by calling methods on the `Topic` object.

If the topic creation fails, the `TopicListener` receives an `onTopicNotConnected` notification.

**Note** You should give each topic a unique name. If the topic already exists on the server, you will simply be connected to it.

## Topic Expiry

There is another version of `createTopic` which takes an expiry time in addition to the topic name and the listener. This allows you to set an expiry time for the topic (in minutes); if the topic is inactive for that amount of time, the server will automatically delete it. A topic created without an expiry time will remain on the server indefinitely, until deleted explicitly by a subscriber.

## onTopicConnected

After connecting to the topic, the listener receives an `onTopicConnected` callback. (In the case of failure, it will receive an `onTopicNotConnected` callback, containing the topic and an error message.) The `onTopicConnected` callback has two parameters, the `Topic` object, and a `Map` which contains an object (under the key `data`), which represents all the data currently associated with the topic. This object is a `List` of `LinkedHashMap` objects, each of which contains the data item's values. The application can iterate through the data items to display them to the newly connected user:

```
public void onTopicConnected(Topic topic, Map<String, Object> data)
{
   List<LinkedHashMap<String, Object>> dataList =
   (List<LinkedHashMap<String, Object>>)data.get("data");
   Iterator<LinkedHashMap<String, Object>> it = dataList.iterator();
   LinkedHashMap<String, Object> pair;
   Boolean deleted;
   String key, value;
   while (it.hasNext())
   {
      pair = it.next();
      deleted = (Boolean)pair.get("deleted");
      key = (String)pair.get("key");
      value = (String)pair.get("value");
      // Display data
```

```
    }
}
```

# Unsubscribing from a Topic

You disconnect from a topic by calling the `Topic` object's `disconnect` method; it takes a single parameter which allows you to choose whether to delete the topic from the server or not. Passing in `false` for this parameter will leave the topic in place on the server; passing in `true` will destroy the topic on the server (and disconnect any other clients connected to the same topic).

```
...
topic.disconnect(true);
...
public void onTopicDeleted(Topic topic, String message)
{
   // Actions to take when topic successfully deleted
}
```

You will receive a notification, one of `onTopicDeleted` or `onTopicNotDeleted`, if you have chosen to delete the topic. Other users will receive an `onTopicDeletedRemotely` notification if the delete was successful.

**Note**

If you choose to merely unsubscribe from the topic without deleting it (by calling `topic.disconnect (false)`), then you will not see any notifications that the unsubscribe has been successful. Nor will other subscribers receive any notification that you have unsubscribed.

## onTopicDeletedRemotely

All clients connected to the topic receive a `onTopicDeletedRemotely` callback when the topic is deleted from the server, whether as a result of any client deleting it, or of the topic expiring on the server (see **Topic Expiry on the previous page** for details of topic expiry). Once a topic has been deleted, the client should not call any of that topic's methods (which will fail in any case), and should consider itself unsubscribed from that topic. If a topic with the same name is subsequently created, it is a new topic, and the client will not be automatically subscribed to it.

# Publishing Data to a Topic

Once connected to a topic, the client application can add data to it by calling the `submitData` method, passing in a key and a value.

```
...
topic.submitData("foo", "bar");
...
public void onTopicSubmitted(Topic topic, String key, String value, int
version)
{
   // Actions for data submitted successfully
}
```

If the submission is successful, the `TopicListener` will receive an `onTopicSubmitted` notification; otherwise, it will receive an `onTopicNotSubmitted` notification. The `onTopicSubmitted` notification contains the `topic`, `key`, and `value` of the data submitted, and a `version` parameter, which indicates how many times the `value` for this `key` has been changed. By tracking the `version`, you can check whether the data you have just submitted is actually the current data for the key on the topic.

When you submit data to the topic, users connected to the topic will receive an `onTopicUpdate` notification, containing the data which you have submitted.

### onTopicUpdated

A client receives a `onTopicUpdated` callback when any client connected to the topic makes a change to a data item on that topic. The callback contains the `key`, `value`, and `version` parameters detailed previously (`value` contains the new value), and an additional `deleted` parameter, which will be `true` if the data item was deleted from the server (see **Deleting Data from a Topic below**).

## Deleting Data from a Topic

The client application can delete data from the topic by calling the `deleteData` method, passing in the `key` under which the data is stored on the topic.

```
 ⋮
topic.deleteData("foo");
 ⋮
public void onDataDeleted(Topic topic, String message)
{
   // Actions when data successfully deleted
}
```

The application will receive notifications `onDataDeleted` or `onDataNotDeleted`. Other users will receive an `onTopicUpdated` notification in which the `deleted` parameter is `true`.

## Sending a Message to a Topic

The application can send a message to a topic using the `sendAedMessage` method.

```
 ⋮
topic.sendAedMessage("Hello World!");
 ⋮
public void onTopicSent(Topic topic, String message)
{
   // Actions to take when message successfully sent
}
```

If the message is sent successfully, you will receive an `onTopicSent` notification; otherwise, you will receive `onTopicNotSent`. Other users connected to the topic will receive an `onMessageReceived` notification.

### onMessageReceived

The `TopicListener` will receive an `onMessageReceived` callback whenever any connected client (including itself) sends a message to the topic. It contains the `topic` parameter, and the `message` parameter containing the text of the sent message.

# Self-Signed Certificates

If you are connecting to a server that uses a self-signed certificate, you have two options:

1. Make use of the `setTrustManager` and `setHostNameVerifier` methods on the `UC` object to perform your own validation (which could be to allow any connection) of the SSL connection.
2. Add the server certificate and the associated CA root certificate to the Credential Storage on your client.

You can obtain the server certificate and CA root certificate through the REAS Administration screens. The *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-installation-guides-list.html) explains how to view and export certificates. You will need to extract the HTTPS Identity

Certificate (server certificate) and the Trust Certificate (CA root certificate) that has signed your server certificate.

Once you have exported and downloaded the two certificates, they need to be copied to your client. Please follow the user documentation for your device to install the certificates.

You should then view the installed server certificate through the appropriate tool (**Android Settings->Security->Credential Storage**) and confirm that the server certificate is trusted. If it is, then your application should be able to connect to the server.

# Bluetooth Support

An REM Advanced SDK application can support Bluetooth devices using the `AudioDeviceManager` class, an instance of which is available on the `Phone` object which is handling the call. While this is in use, REM Advanced SDK will:

- Automatically send audio output to a Bluetooth headset when one becomes available.
- Send audio output to a wired headset if one is available and there is no Bluetooth device connected.
- Send audio output to another audio device if neither a headset nor a Bluetooth device is available.

Using the `AudioDeviceManager`, the application can:

- Receive notifications when devices become or cease to be available (such as when a wired headset is plugged in or unplugged, or when a Bluetooth device goes in or out of range), in order to change the above behavior. See **Using the Listener on the next page**.
- Define which audio output on the phone should handle the audio if neither a Bluetooth device nor a headset is available.
- Define a default audio output on the phone, which will handle the audio if neither a Bluetooth device nor a headset is available, and the application has not selected a specific audio device.
- Get a list of available audio outputs on the phone
- Determine which of the phone's audio outputs currently handles the audio

**Note** Internally, `AudioDeviceManager` uses methods of the Android `AudioManager` class. Application code can still call methods of this class directly, but does not need to call methods such as `setSpeakerPhoneOn`; instead, it can use `AudioDeviceManager.setAudioDevice`. When calling `AudioManager` methods directly, take care that doing so does not interfere with the workings of the `AudioDeviceManager`.

## Starting and Stopping AudioDeviceManager

The `Phone` object starts the `AudioDeviceManager` automatically, so that the application can use the `AudioDeviceManager` methods as soon as the `Phone` object is available:

```
AudioDeviceManager adm;

public void onSessionStarted()
{
    ⋮
  adm = uc.getPhone().getAudioDeviceManager();
  adm.addListener(this);
    ⋮
}
```

The application can call these methods to set the audio devices which the phone should use for calls made or received by the application. Calls which are not handled by the REM Advanced SDK application will be unaffected, and will use the phone's default behavior.

The listener interface (`this` in the above example) should be a class which implements `AudioDeviceManagerListener`, which has a single method, `getDeviceListChanged`. The `AudioDeviceManager` will call `getDeviceListChanged` when it becomes aware of a change to the list of available devices, or to the selected device. It will make the check for available devices (and possibly change the selected device, if the preferred device has become available) when a wireless headset is plugged in, or a Bluetooth headset becomes available; the application can also trigger the check by calling `updateAudioDeviceState`. See **Using the Listener below**.

There may be occasions when the application does not wish to use `AudioDeviceManager`. In order to return to the Android device's default behavior, the application can call `stop`:

```
phone.getAudioDeviceManager().stop();
```

To switch back to using the `AudioDeviceManager`, the application can call `start`:

```
phone.getAudioDeviceManager().start();
```

# Using the Listener

`AudioDeviceManager` sends output to available devices in the following order:

1. Bluetooth device
2. Wired headset
3. The device selected by the application (see **Setting the Audio Device on the facing page**)
4. The default device (see **Setting the Default Device on the facing page**)

When the set of available devices changes (for instance, if a wired headset is unplugged, or a paired Bluetooth device comes into range), REM Advanced SDK will start to send audio to the first available device in the above list. Thus, the audio will switch to a Bluetooth device as soon as one becomes available; and if a wired headset is plugged in (when a Bluetooth device is not connected), it will switch to that.

This (Bluetooth takes precedence over headset, which takes precedence over the speaker or earpiece) is the most likely requirement; but the application can override this behavior by setting the preferred device in the `AudioDeviceManagerListener.onDeviceListChanged` method:

```
AudioDeviceManager adm;

public void onSessionStarted()
{
    ⋮
   adm = uc.getPhone().getAudioDeviceManager();
   adm.addListener(this);
    ⋮
}

void onDeviceListChanged(Set<AudioDevice> availableDevices, AudioDevice
selectedDevice)
{
   if (availableDevices.contains(AudioDevice.WIRED_HEADSET)
      && (selectedDevice != AudioDevice.WIRED_HEADSET))
   {
      adm.setAudioDevice(AudioDevice.WIRED_HEADSET);
   }
}
```

The above code would send a call's audio to the wired headset, even when a Bluetooth device is connected.

**Note** The list of available devices which `AudioDeviceManager` passes to `onDeviceListChanged` will not necessarily contain all the devices which exist on the phone; see **Listing Available Devices below**.

## Setting the Audio Device

The application can set the device which handles audio for the call:

```
phone.getAudioDeviceManager().setAudioDevice(AudioDevice.BLUETOOTH);
```

The argument to the method must be one of the members of the `AudioDevice` enumeration:

- `NONE`

  The application has no preference, and will accept the default behavior of the phone.

- `SPEAKER_PHONE`

  Audio will be sent to the loudspeaker in the phone, and will be audible to others in the vicinity. Audio input will be from the phone's internal microphone.

- `WIRED_HEADSET`

  Audio will be sent to a device attached to the jack in the phone. If this device has a microphone, that will be used for audio input.

- `EARPIECE`

  Audio will be sent to the internal speaker, and received from the internal microphone. The user will have to hold the phone to their ear during the call.

- `BLUETOOTH`

  Audio will be sent to and received from a paired Bluetooth device.

**Note** The phone will not necessarily use the device which the application tries to set. When it sets the audio device, `AudioDeviceManager` will check to see whether the selected device is in the list of available devices, and if it is not, it will choose either another device or the default device set by the application (see **Setting the Default Device below**). It will do this without throwing an exception.

## Setting the Default Device

The application can set a fallback device in case the preferred device is unavailable:

```
phone.getAudioDeviceManager().setDefaultAudioDevice(AudioDevice.EARPIECE);
```

The argument is one of the values from the `AudioDevice` enumeration (see **Setting the Audio Device above**).

Setting the default device establishes a fallback option in case neither a Bluetooth device nor a wired headset is available. As such, the application can only set it to either `EARPIECE` or `SPEAKER_PHONE`, and if an earpiece is not available (a tablet will not have an earpiece, for instance), `AudioDeviceManager` will only allow it to set it to `SPEAKER_PHONE`. The default value (if the application never calls `setDefaultAudioDevice`) is `SPEAKER_PHONE`.

## Listing Available Devices

The application can get a list of available audio devices by calling the `getAudioDevices` method:

```
Set<AudioDevice> devices = phone.getAudioDeviceManager().getAudioDevices();
```

The resulting set contains members of the `AudioDevice` enumeration, taken from the connected devices known to the phone. The list of available devices will not necessarily contain all the devices which exist on the phone. In particular, if it includes `WIRED_HEADSET`, it will not include `EARPIECE` or `SPEAKER_`

PHONE, because it considers these to be mutually exclusive. For a phone, the list of available devices may be:

- BLUETOOTH, WIRED_HEADSET
- BLUETOOTH, SPEAKER_PHONE
- BLUETOOTH, SPEAKER_PHONE, EARPIECE
- WIRED_HEADSET
- SPEAKER_PHONE
- SPEAKER_PHONE, EARPIECE

A tablet is not likely to have an earpiece.

The application can also find which device is currently being used as the audio device:

```
AudioDevice device = phone.getAudioDeviceManager().getSelectedAudioDevice();
```

This will work whether the preferred device has been set explicitly (using setAudioDevice) or not.

# Responding to Network Issues

As Remote Expert Mobile is network-based, it is essential that the client application is made aware of any loss of network connection. When a network connection is lost, the server uses SIP timers to determine how long to keep the session alive before reallocating the relevant resources. Any application you develop should make use of the available callbacks in the Remote Expert Mobile API, and any other available technologies, to handle network failure scenarios.

To receive callbacks relating to network issues, the application must implement the UCListener interface.

## Reacting to Network Loss

In the event of network connection problems, the SDK automatically tries to re-establish the connection. Seven attempts are made at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to onConnectionRetry on the UCListener precedes each of these attempts. The callback supplies the attempt number and the delay before the next attempt in its two parameters.

When all reconnection attempts are exhausted, the UCListener receives the onConnectivityLost callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in **Creating the Session on page 4**.

If any of the reconnection attempts are successful, the UCListener receives the onConnectionReestablished callback.

**Note**

The retry intervals, and the number of retries attempted by the SDK are subject to change in future releases. Do not rely on the exact values quoted above.

## Reacting to Network Changes

If the issues with the network are caused by a temporary loss of connectivity (for example, when moving between two Wi-Fi networks, or from a Wi-Fi network to a cellular data connection), the client application should not log out from the session and log back in, as all session state will be lost.

To avoid this, an application should register with the OS to be told of network reachability changes using the ConnectivityManager class. When the OS notifies the application that the network has changed, it should pass these details on to the UC instance by calling the setNetworkReachable method.

When the application starts, it should check for reachability. When the network is reachable, call UC.setNetworkReachable(true). Until this call is made, REM Advanced SDK will not try to make a session connection to the server.

If the network reachability drops after a session has been established, the client application needs to call `UC.setNetworkReachable(false)`.

If network reachability changes from a cellular data connection to Wi-Fi or *vice versa*, call `UC.setNetworkReachable(false)` followed by `UC.setNetworkReachable(true)` to disconnect from the first network and re-register on the second.

## Network Quality Callbacks

During a call the application can receive callbacks on the quality of the network by implementing the `onInboundQualityChanged` method of the `CallListener`.

```
void onInboundQualityChanged(Call call, int inboundQuality)
{
   // Show indication of quality
}
```

The `inboundQuality` parameter is a number between `0` and `100`, where `100` indicates perfect quality. The application might choose to show a bar in the UI, the length of the bar indicating the quality of the connection.

The SDK starts collecting metrics as soon as the remote media stream is received. The metrics are collected every 5s, so the first quality callback fires roughly 5s after this remote media stream callback has fired.

The callback will then be fired every time a different quality value is calculated; so if the quality is perfect then there will be an initial quality callback with a value of `100` (after 5s), and then no further callback until the quality degrades.

# API Changes

| API Release | Description |
|---|---|
| 11.6 | Added `AudioDeviceManager` to support Bluetooth devices |
| 10.6 | Added `CallStatus` parameter to `onCallFailed` and `onDialFailed` methods of `CallListener` interface, to convey exact reason for the failure. |
| | Added `setTrustManager` method to the `UC` object, which allow a developer to set an alternative Trust Manager to be used when validating a secured Websocket connection between the client and gateway. |
| | Added `setHostnameVerifier` method to the `UC` object, which allow a developer to set an alternative Hostname Verifier to be used when validating a secured Websocket connection between the client and gateway. |
| | Additional callback to `CallListener`: `onInboundQualityChanged` when the quality of the remote stream changes. |
| | Support for multiple calls including moving `enableLocalAudio`/`enablelocalVideo` from the `Call` object to the `Phone` object.<br>The `VideoSurfaceListeneronFrameSizeChanged` method has been changed to include an additional `VideoSurface` parameters because there will potentially be multiple remote video surfaces that will need to be differentiated.<br>The `setPreviewView` method has been removed from the `Call` object and `setVideoView` has been removed from the `Phone` object. |
| | Additional callback to `UCListener`: `onConnectionRetry` when attempting to reconnect the websocket connection to the Gateway. |
| | **VideoSurface**<br>VideoSurface create (Context, Point, VideoSurfaceListener);<br>A `VideoSurface` is now created using the `createVideoSurface` method of the `Phone` |

| API Release | Description |
|---|---|
| | interface. |
| | **VideoSurface**<br>abstract modifier<br>onPause/onResume<br>GLSurfaceView.Renderer methods<br>`VideoSurface` is no longer an abstract class as there are no circumstances in which an application developer would be expected to implement their own `VideoSurface` class.<br>The `onPause`/`onResume` and `GLSurfaceView.Renderer` methods have been removed to simplify the API and avoid confusion. |
| | **Call**<br>setPreviewSurface(VideoSurface);<br>setVideoSurface(VideoSurface);<br>Methods renamed to `setPreviewView` and `setVideoView` to ensure that they are consistent with identical methods on the `Phone` interface. |
| | Added ability to set the video capture resolution.<br>The list of possible resolutions is available from the `Phone` object using `getRecommendedCaptureSettings`<br>Set the resolution using `Phone.setPreferredCaptureResolution`<br>Set the frame rate using `Phone.setPreferredCaptureFrameRate` |
| | The API now offers `hold` and `resume` methods on the `Call` object. |

CHAPTER 6

# Creating an OSX Client Application



Remote Expert Mobile enables you to develop OSX applications offering users the following methods of communication:

- Voice calling

Remote Expert Mobile provides you with an OSX SDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

To develop OSX applications using Remote Expert Mobile, Xcode 4.5 or later is required.

The Remote Expert OSX SDK is made up of the following classes:

- The top-level `ACBUC` class and its delegate protocol `ACBUCDelegate`.
- Two classes for voice calling:
  - `ACBClientPhone` and its delegate protocol `ACBClientPhoneDelegate`.
  - `ACBClientCall` and its delegate protocol `ACBClientCallDelegate`.

The OSX SDK reference documentation, including a full list of available methods and their associated callbacks, is delivered in the `docs.zip` file. Open `index.html` to view the API documentation.

# Setting up a Project

**Note** Before following this process, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **Creating the Web Application on page 3**.

To set up a project including the Remote Expert Mobile, you first need to create a new project and add OSX native frameworks to it:

1. Open Xcode and choose to create a **Cocoa Application**, giving your project an appropriate name. The following code samples use the example name 'WebrtcClientOSX'.
2. Click the **Build Phases** tab, and expand the **Link Binary with Libraries** section by clicking on the title.
3. Click the **+** button to display the file explorer.
4. Select the following OSX native dependencies from the OSX folder:
   - `Cocoa.framework`
   - `QuartzCore.framework`

The dependencies you selected are now displayed in the **Link Binary with Libraries** section.

Now, you need to add the Remote Expert Mobile framework to your project.

1. Select your project and click the **Build Phases** tab.
2. Expand the **Link Binary with Libraries** section by clicking on the title.
3. Click the **+** button. When the file explorer displays, click **Add Other**.
4. Navigate to the `Frameworks/ACBClientSDK.framework` folder, select it and click **OK**.

# Initializing the ACBUC Object

The application accesses the API initially via a single object, `ACBUC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see **Creating the Web Application on page 3**), and initialize the `ACBUC` object using it. Once it has received the Session ID, the client application must call the `ucWithCofiguration` method on the `ACBUC`:

```
- (void) initialize
{
   NSString* sessionId = [self getSessionId];
   ACBUC* uc = [ACBUC ucWithConfiguration:sessionId delegate:self];
   [uc startSession];
}

- (void) ucDidStartSession:(ACBUC *)uc
{
    ⋮
}
```

The delegate (in this case `self`) must implement the `ACBUCDelegate` protocol. Once the session has started, REM Advanced SDK will call the delegate method `ucDidStartSession`, and the application can make use of the `ACBUC` object. (If the session does not start, one of the delegate's error methods will be called.)

# Adding Voice

Once the application has initialized the `ACBUC` object, it can retrieve the `ACBClientPhone` object. It can then use the phone object to make or receive calls, for which it returns `ACBClientCall` objects. Each

one of those objects has a delegate for notifications of errors and other events.

# Making a Call

In the following example, the application makes a call (using `createCallToAddress` on the `ACBClientPhone` object) as soon as the session has started (see **Initializing the ACBUC Object on the previous page**):

```
- (void) ucDidStartSession:(ACBUC *)uc
{
   ACBClientPhone* phone = uc.phone;
   phone.delegate = aPhoneDelegate;
   phone.previewView = previewView;
   ACBClientCall* call = [phone createCallToAddress:calleeAddress
   withAudio:ACBMediaDirectionSendAndReceive withVideo:ACBMediaDirectionNone
   delegate:aCallDelegate];
   call.videoView = aVideoView;
}
```

You can change the values of the `withAudio` parameter to make the call one way; the `withVideo` parameter should always be `ACBMediaDirectionNone`. Valid values are:

- `ACBMediaDirectionNone`
- `ACBMediaDirectionSendOnly`
- `ACBMediaDirectionReceiveOnly`
- `ACBMediaDirectionSendAndReceive`

**Note** The older form, `createCallToAddress:audio:video:delegate:`, which took two boolean values, is now deprecated.

# Receiving a Call

REM Advanced SDK invokes the `ACBClientPhoneDelegatedidReceiveCall:` delegate method when it receives an incoming call. The application can answer the incoming call by calling its `answerWithAudio:andVideo:` method:

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
   [call answerWithAudio:ACBMediaDirectionSendAndReceive
   video:ACBMediaDirectionNone]
}
```

To reject the call, use `[call end]`.

You can change the values of the parameters to answer the call with a specific direction for audio. Valid values are:

- `ACBMediaDirectionNone`
- `ACBMediaDirectionSendOnly`
- `ACBMediaDirectionReceiveOnly`
- `ACBMediaDirectionSendAndReceive`


Note

- The older form, which took two boolean values, is now deprecated.
- The options specified in the answer affect both sides of the call; that is, if the remote party placed an audio and video call, and the local application answers as audio only (as an OSX application must), then neither party sends or receives video.
- If your application plays its own ringing tone, please note that the OSX SDK makes calls to the `AVAudioSessionsharedInstance` object when establishing a call. For this reason, we recommend waiting until you receive a call status of `ACBClientCallStatusRinging` (from `ACBClientCallDelegatedidChangeStatus`) before calling `AVAudioSessionsharedInstance` methods.
- Video is not supported in this release of the OSX Remote Expert Mobile, so only `ACBMediaDirectionNone` is valid for the video direction.

## Video Views and Preview Views

Video is not supported in this release of the OSX Remote Expert Mobile.

## Muting the Local Audio Stream

During a call the application can mute or unmute the local audio stream. Muting the stream stops it being sent by the user to the remote party; however the user will still receive any stream that the remote party sends.

To mute the audio stream use the `enableLocalAudio` method of the call:

```
- (void) incomingCallReceived:(ACBClientCall*)call
{
   self.call = call;
   [call answerWithAudio:YES video:NO];
}
- (void) muteButtonPressed:(UIButton*)button
{
   [self.call enableLocalAudio:NO];
}
```

## Holding and Resuming a Call

During a call the application can put a call on hold (for example, in order to make or receive another call). Placing the call on hold pauses both the stream sent by the user and the stream sent by the remote party; only the party who placed the call on hold can resume it.

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
   [call answerWithAudio:YES video:NO]
}
- (void) holdButtonPressed:(UIButton*)button
{
   [call hold];
}
- (void) resumeButtonPressed:(UIButton*)button
{
   [call resume];
}
```

## DTMF Tones

Once a call is established, an application can send DTMF tones on that call by calling the `playDTMFCode` method of the `ACBClientCall` object:

```
[call playDTMFCode:@"#123*" localPlayback:YES];
```

- The first parameter can either be a single tone, (for example, `6`), or a sequence of tones (for example, `#123,*456`). Valid values for the tones are those characters conventionally used to represent the standard DTMF tones: `0123456789ABCD#*`.

**Note** The comma indicates that there should be a two second pause between the `3` and the `*` tone.

- The second parameter is a boolean which indicates whether the tone is played back locally.

## Handling Multiple Calls

Applications developed with Remote Expert OSX SDK do not support multiple simultaneous calls:

## Monitoring the State of a Call

A call transitions through several states, and the application can monitor these by assigning a delegate to the call:

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
   call.delegate = self;
    ⋮
}
```

Each state change fires the `call:didChangeStatus:` delegate method. As the outgoing call progresses toward being fully established, the application receives a number of calls to `didChangeStatus`, each time being passed one of the `ACBClientCallStatus` enumeration values.

Switching on the value of the enumeration allows the application to adjust the UI to give the user suitable feedback, for example by playing a local audio file for ringing or alerting:

```
- (void) call: (ACBClientCall*)call didChangeStatus:(ACBClientCallStatus)
status
{
   switch (status)
   {
      case ACBClientCallStatusRinging:
         [self playRingtone];
         break;
      case ACBClientCallStatusInCall:
         [self stopRinging];
         break;
      case ACBClientCallStatusEnded:
      case ACBClientCallStatusBusy:
      case ACBClientCallStatusError:
      case ACBClientCallStatusNotFound:
      case ACBClientCallStatusTimedOut:
         [self updateUIForEndedCall];
         break;
      default:
         break;
   }
```

```
}
```

The following table gives the possible status codes:

| Status code | Meaning |
|---|---|
| `ACBCallStatusSetup` | Call is in process of being set up |
| `ACBCallStatusAlerting` | The call is an incoming one which is alerting (ringing) |
| `ACBCallStatusRinging` | An outgoing call is ringing at the remote end |
| `ACBCallStatusMediaPending` | The call is connected, and waiting for media |
| `ACBCallStatusInCall` | The call is fully set up, including media |
| `ACBCallStatusBusy` | Dialed number is busy |
| `ACBCallStatusNotFound` | Dialed number is unreachable or does not exist |
| `ACBCallStatusTimedOut` | Dialing operation timed out without a response from the dialed number |
| `ACBCallStatusError` | An error has occurred on the call. such the media broker reaching its full capacity, the network terminating the request, or there being no media. |
| `ACBCallStatusEnded` | The call has ended |

# Threading

All method invocations on the SDK, even to access read-only properties, must be made from the same thread. This can be any thread, and not necessarily the main thread of the application. Internally, the SDK may use other threads to increase responsiveness, but any delegate callbacks will occur on the same thread that is used to initialize the SDK.

# Self-Signed Certificates

If you are connecting to a server that uses a self-signed certificate, you need to add that certificate and the associated CA root certificate to the keychain on your client.

You can obtain the server certificate and CA root certificate through the REAS Administration screens. The *Remote Expert Mobile Installation and Configuration Guide, Release 11.6 (1)* (available at http://www.cisco.com/c/en/us/support/customer-collaboration/remote-expert-mobile/products-programming-reference-guides-list.html) explains how to view and export certificates. You need to extract the HTTPS Identity Certificate (server certificate) and the Trust Certificate (CA root certificate) that has signed your server certificate.

Once you have exported and downloaded the two certificates, you need to copy them to your client. Please follow the user documentation for your device to install the certificates.

You should then view the installed server certificate through the appropriate tool (**iOS Settings->General->Profiles** or **OSX Keychain**) and confirm that the server certificate is trusted. If it is, then your application should connect to the server.

Alternatively, you can use the `acceptAnyCertificate` method of the `ACBUC` object before calling `startSession`, although this should only be used during development:

```
ACBUC* uc = [ACBUC ucWithConfiguraton:sessionId stunServers:stunServers
delegate:self];
[uc acceptAnyCertificate:TRUE];
[uc startSession];
```

# Responding to Network Issues

As the OSX SDK is network-based, it is essential that the client application is aware of any loss of connection. Remote Expert Mobile does not dictate how you implement network monitoring; however, the

sample application uses the `SystemConfiguration` framework.

Depending on the nature of the issues with the network, the client application should react differently.

## Reacting to Network Loss

In the event of network connection problems, the SDK automatically tries to re-establish the connection. It will make seven attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to the `willRetryConnectionNumber:in:]` on the `ACBUCDelegate` precedes each of these attempts. The call-back supplies the attempt number (as an `NSUInteger`) and the delay before the next attempt (as an `NSTimeInterval`) in its two parameters.

When all reconnection attempts are exhausted, the `ACBUCDelegate` receives the `ucDidLoseConnection` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in **Creating the Session on page 4**.

If any of the reconnection attempts are successful, the `ACBUCDelegate` receives the `ucDidReestablishConnection` callback.

Note that both the `willRetryConnectionNumber` and `ucDidReestablishConnection` are optional, so the application may choose to not implement them. The connection retries are attempted regardless.

**Note**

The retry intervals, and the number of retries attempted by the SDK are subject to change in future releases. Do not rely on the exact values given above.

## Reacting to Network Changes

If the issues with the network are caused by a temporary loss of connectivity (for example, when moving between two Wi-Fi networks, or from a Wi-Fi network to a cellular data connection), the client application should not log out from the session and log back in (as described in **Reacting to Network Loss above**) as the current session is lost.

To avoid this situation, the client application should register with iOS to receive notification of changes in network reachability. When iOS notifies the client application that the network has changed, the application should pass these details to the `ACBUC` instance.

When the client application starts, it should check for network reachability. When the network is reachable, the application calls `ACBUC setNetworkReachable:YES`; until this call is made, the application does not attempt to create a session.

If the network reachability drops after a session has been established, the client application needs to call `ACBUC setNetworkReachable:NO`.

If the network reachability changes from a cellular data connection to a Wi-Fi network, or *vice versa*, the client application needs to call `ACBUC setNetworkReachable:NO` followed by `ACBUC setNetworkReachable:YES` to disconnect from the first network and re-register on the second.

# API Changes

| API Release | Description |
|---|---|
| 10.6 | Support for self-signed certificates – see **Self-Signed Certificates on the previous page** |
| | Introduces OSX Remote Expert Mobile with support for audio-only calls. |

CHAPTER 7

# Creating a Windows Client Application



Remote Expert Mobile enables you to develop Windows applications that offer users the ability to make voice and video calls.

Remote Expert Mobile provides you with a Windows SDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

The Remote Expert Windows SDK is made up of the following main classes:

- The top-level `UC` class and it's corresponding `UCListener` interface.
- `Phone` and `PhoneListener`, for creating and receiving calls.
- `Call` and `CallListener`, for working with calls.
- `AED`, `Topic`, and `TopicListener`, for working with Application Event Distribution

The listener classes are interface classes that define pure virtual functions that the application is expected to implement in order to receive the notifications defined by that interface.

# Setting up a Project

**Note** Before following this process, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **Creating the Web Application on page 3**.

To set up a project including the Remote Expert Mobile, you first need to create a new project and add SDK native libraries and headers:

1. Extract the Client SDK archive to a directory that will be referred to by your new project, as described in the following steps,
2. Open Visual Studio, and select **File->New Project...** ,
3. Select your preferred project template, name etc then click **OK** to create the project,
4. In the **Solution Explorer** view, right-click the project name and choose **Properties** from the menu,
5. Select **All Configurations** from the **Configuration:** combo box,
6. In the **C/C++ ->General** section, add the CSDK `csdk-sample\target\lib\fcsdk\include` directory (from step 1) to the **Additional Include Directories**,
7. In the **Linker->Input** section, add the `csdk-sample\target\lib\fcsdk\Release\ClientSDKWin.lib` library (from step 1) to the **Additional Dependencies**.

# Initializing the UC and Starting the Session

The application initially accesses the API via a single object, `UC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see **Creating the Web Application on page 3**), and initialize the `UC` object using it. Once it has received the Session ID, the client application must create the `UC` object, then call its `StartSession` method:

```
std::string stun = "[{'url': 'stun:stun.1.google.com:19302'}]";
uc = std::make_unique<UC>(sessionID, stun, this);
uc->StartSession();
```

The first line in this sample creates the `UC` object using the session ID string obtained from the Gateway. It also registers `this` as the `UCListener` object, which implies that `this` must be an instance of a class that extends the `UCListener` interface class. Of course, you may wish to have a different object act as the `UCListener`.

The second parameter is a list of STUN servers in the form of a JSON string. STUN servers are not necessary if the Gateway is not behind a firewall, so that *Network Address Translation* is not needed, in which case the array can be empty (`"[]"`). You can provide your own STUN server instead of the public Google one above; and you can provide more than one in the array, in which case they will be tried in sequence until REM Advanced SDK finds a working one.

In the second line, the `UC` tries to start the session; the `UCListener` object will receive asynchronous notification that the session has started or failed:

```
void MyUCListener::OnSessionStarted()
{
   uc->GetPhone()->SetListener(this);
}
```

The implementation of `OnSessionStarted` obtains the `Phone` class from the `UC`, and registers `this` as the listener. This implies that `this` must be a class that extends the `PhoneListener` interface class. Again, you may use a different class as the phone listener.

The primary purpose of the `PhoneListener` is to receive notification of incoming calls (see **Receiving a Call on the facing page**).

The failure notification is `OnSessionNotStarted()` .

# Adding Voice and Video

Once the application has created the `UC` object, it can retrieve the `Phone` object. Once the session has started, the `Phone` object can be used to make or receive calls, which are represented by `Call` objects. Each of the `UC`, `Phone`, and `Call` objects have corresponding listener interfaces that the application can implement in order to receive error notifications and other events.

## Adding a Preview Window before a Call is made

If you want to add a preview window (a window which displays the video which is being sent to the other endpoint) before a call is established, you can call the `Phone::SetPreviewViewName` method. An appropriate time to do this is in the `UCListener::OnSessionStarted` callback:

```
void MyUCListener::OnSessionStarted()
{
   uc->GetPhone()->SetPreviewViewName("local");
};
```

## Making a Call

Once you have created the session, you can start making calls. This is simply a case of asking the `Phone` object to create a `Call`:

```
CallPtr call = uc->GetPhone()->CreateCall("1234",
   MediaDirection::SEND_AND_RECEIVE,
   MediaDirection::SEND_AND_RECEIVE, this);
```

In this example, `1234` is the number we are dialing, the two `MediaDirection` parameters indicate that we want the call to be an audio and video call respectively, and the final parameter registers `this` as the `CallListener`.

The `CreateCall()` function returns the newly created call (`CallPtr` is a `typedef` for `std::shared_ptr<Call>`).

Note that while `CreateCall` immediately returns a `Call` object, the `Call` object cannot be used until the `CallListener` receives the asynchronous notification that the call creation has succeeded.

You can change the values of the media direction parameters to make the call audio only or video only. They must be members of the `MediaDirection` enumeration:

- `NONE`
- `SEND_ONLY`
- `RECEIVE_ONLY`
- `SEND_AND_RECEIVE`

**Note** The older form of `CreateCall`, which took two boolean values, is now deprecated.

## Receiving a Call

REM Advanced SDK invokes the `PhoneListener` object that you registered with the `Phone` object when an incoming call is received:

```
void MyPhoneListener::OnIncomingCall(CallPtr call)
{
   call->Answer(MediaDirection::SEND_AND_RECEIVE, MediaDirection::SEND_AND_
   RECEIVE);
}
```

In this example the application auto-answers the call; most applications will notify the user before invoking either `call->Answer()` to accept, or `call->End()` to reject, the call.

The two parameters to the `Answer()` method indicate that whether or not we want to answer the call with audio and video respectively. They are members of the `MediaDirection` enumeration:

- `NONE`
- `SEND_ONLY`
- `RECEIVE_ONLY`
- `SEND_AND_RECEIVE`

**Note** The older form of `Answer`, which took two boolean values, is now deprecated.

## Displaying Video

Your application can display video received from the remote peer, as well as a preview of the locally captured video that is sent to the remote peer. Displaying video in your application is achieved using two classes provided by the Windows SDK: `VideoView` and `ImagePipeServer`.

The `VideoView` class is provided by the SDK to paint video image data to a device context (`HDC`) provided by your application.

The `ImagePipeServer` is a base class that your application will need to provide a concrete implementation of in order to receive image data and pass it on to the `VideoView`:

```
shared_ptr<VideoView> remoteView = make_shared<VideoView>("Remote Video");
view->SetRefreshViewFunc(std::bind(&MyController::RedrawRemoteView, this));
shared_ptr<MyPipeServer> remotePipe = make_shared<MyPipeServer>("Remote
Video", remoteView);
call->SetVideoViewName("Remote Video");
```

On the first line here we create a `VideoView` with a unique name for our remote view. We then set a refresh function for the view; this refresh function is invoked every time there is a new video frame to render; the application should implement it to invalidate the user interface element that displays the video. Finally, we create an instance of our subclass of `ImagePipeServer`, giving it the details of the `VideoView`, and then tell the `Call` object the name of the `VideoView`.

When the SDK receives a video frame from the remote peer, it invokes `SendImageToView()` on your `ImagePipeServer` subclass. Your implementation of this method must pass the image data on to the `VideoView`:

```
void MyPipeServer::SendImageToView()
{
   VideoViewImageData img = GetImageData();
   remoteView->SetImageData(img);
   RefreshViewFunc func = remoteView->GetRefreshViewFunc();
   func();
}
```

This code updates the video data and triggers your refresh function. When your user interface video element redraws, you can simply pass the `HDC` to the `VideoView` to paint the new video frame:

```
remoteView->Paint(hdc, region);
```

## Muting Local Audio and Video Streams

Muting the local streams will stop audio or video from being sent to the remote party (audio and video received from the remote party is not affected).

The following example shows an application muting both audio and video in response to a mute button being pressed in the user interface:

```
void MyUIController::MuteButtonPressed()
{
   call->EnableLocalAudio(false);
```

```
    call->EnableLocalVideo(false);
}
```

You can also mute the audio and video streams as soon as the call is answered:

```
void MyPhoneListener::OnIncomingCall(CallPtr call)
{
    call->Answer(MediaDirection::RECEIVE_ONLY, MediaDirection::RECEIVE_ONLY);
}
```

## Holding and Resuming a Call

Your application can place a call on hold, and subsequently resume the call. When a call is on hold, no audio or video is played to either end of the call. Only the party that placed the call on hold can resume it.

```
void MyUIController::HoldButtonPressed()
{
    call->Hold();
}
void MyUIController::ResumeButtonPressed()
{
    call->Resume();
}
```

## Sending DTMF Tones

Your application can send DTMF tones on a call:

```
call->SendDTMF("#123*", true);
```

This example sends five tones sequentially. To send a single tone just use a single-character string.

The boolean parameter indicates whether or not you want the tones to also be played back locally, ie, be audible to the user of your application.

Valid values in the string parameter are those conventionally used to denote DTMF tones: `0123456789#*`. A comma character inserts a two-second pause into a sequence of tones.

## Handling Multiple Calls

Applications developed with the Remote Expert Windows SDK do not support multiple simultaneous calls.

## Setting Video Resolution

Your application can configure both the resolution and frame rate of the video that is sent to the remote party in a video call. You set the video resolution on the `Phone` object:

```
phone->SetPreferredCaptureResolution(VideoCaptureResolution::RESOLUTION_
1280x720);
```

The `VideoCaptureResolution` enumeration class defines the set of resolutions available to your application.

You can also set the frame rate:

```
phone->SetPreferredCaptureFrameRate(20);
```

Note that changes to the video resolution and frame rate apply only to new calls; the resolution of existing calls are not affected.

## Monitoring the State of a Call

During call setup, the call will transition through several states, from the initial setup to being connected with media available (or failure). You can monitor these states using the `CallListener`. Switching on

the value of the `CallStatus` enumeration allows the application to adjust the UI to give the user suitable feedback; for example by playing a local audio file for ringing or alerting:

```
void MyCallListener::OnStatusChanged(CallStatus status)
{
   switch (status)
   {
      case RINGING:
         playRingtone();
         break;
      case IN_CALL:
         stopRinging();
         break;
      case ENDED:
      case BUSY:
      case CALL_ERROR:
      case NOT_FOUND:
      case TIMED_OUT:
         updateUIForEndedCall();
         break;
      default:
         break;
   }
}
```

`CallStatus` is an enumeration class that enumerates all of the possible states:

| Status Code | Meaning |
|---|---|
| `UNINITIALIZED` | The `Call` object has been created but not initialized |
| `SETUP` | Call is in process of being set up |
| `ALERTING` | An incoming call is alerting (ringing) at the user's end. |
| `RINGING` | An outgoing call is ringing at the remote end |
| `MEDIA_PENDING` | The call is connected, and waiting for media |
| `IN_CALL` | The call is fully set up, including media |
| `BUSY` | Dialed number is busy |
| `NOT_FOUND` | Dialed number is unreachable or does not exist |
| `TIMED_OUT` | The dialing operation timed out without a response from the dialed number |
| `CALL_ERROR` | The call has errored. This may be because something (such as a media broker) was unavailable, or because of network conditions, or for some other reason. More information is not available. |
| `ENDED` | The call has ended |

The `CallListener` interface defines other notification functions that allow your application to detect call quality changes and dial/call failures.

# Adding Application Event Distribution

*Application Event Distribution* is contained in the `AED` and `Topic` objects. There is also a `TopicListener` object which receives information of changes which have been made to the topics.

In order to use Application Event Distribution, you must first obtain an `AED` object from the `UC` object.

```
acb::AEDPtr aed = uc->GetAED();
```

Initialization of the UC object is exactly the same as for voice and video calling (see **Initializing the UC and Starting the Session on page 76**).

# Creating a Topic

Once you have obtained an AED object, the next thing to do is to create a topic.

```
acb::TopicPtr topic = aed->CreateTopic(name, listener);
```

or

```
acb::TopicPtr topic = aed->CreateTopic(name, expiry, listener);
```

The name is a std::string which uniquely identifies the topic on the server, and the expiry is a time in minutes. If you create the topic with an expiry time, it will be automatically removed from the server after it has been inactive for that time. When created without an expiry time (by the first method), the topic exists indefinitely, and needs to be deleted explicitly.

The listener is an object which descends from the TopicListener object. It contains a number of call-back methods through which inform the application about things which happen on the topic.

If the topic identified by the name parameter already exists on the server, you will be connected to it.

## OnTopicConnected

After creating a topic, the listener should receive an OnTopicConnected callback (in case of failure, the listener will receive an OnTopicNotConnected callback instead). The parameters to the callback are a TopicPtr identifying the topic, and a TopicDataPtr. The TopicDataPtr points to a structure containing all the existing data for the topic (obviously, if the topic is newly created, this is empty). The data on a topic consists of name-value pairs, and you can either look for the value of a data item that you know will be there:

```
const acb::TopicDataValue* const value = data->GetValue(name);
```

or call GetStart() to get a TopicDataIterator pointing at the beginning of the data, and so get all the data values:

```
acb::TopicData::TopicDataIterator it = data->GetStart();
if (!it.isEnded())
{
   do
   {
      std::string name = it.GetKey();
      acb::TopicDataValue value = it.GetValue();
   } while (it.Next());
}
```

A TopicDataValue can contain one of a number types of data object (std::string, bool, int, or double), and provides methods (GetAsString(), GetAsInt(), etc.) for retrieving the actual value; it also provides a GetType() method which returns a std::type_info object. Currently, however, it will always be a std::string.

# Publishing Data to a Topic

Once the client application has created and/or connected to the topic, it can publish data on it. Data consists of name-value pairs:

```
std::string name = "name";
std::string value = "value";
topic→SubmitData(key, value);
```

Having submitted the data, the listener will receive either an OnTopicSubmitted or an OnTopicNotSubmitted (in the case of failure) callback. Both the key and the value are a std::string. In the case of a successful submission, there will also be an OnTopicUpdated callback when the data is sent to all clients connected to the topic. The OnTopicSubmitted callback includes a

`version` parameter, enabling clients to know when they receive an `OnTopicUpdate` callback, whether it refers to data they have just submitted or not (if it does, the `version` parameters will be the same). For a newly created data items, the `version` will be `0`.

The client application can also change the value of an existing data item by calling `SubmitData`. In this case, the callbacks (`OnTopicSubmitted` and `OnTopicUpdated`) will include a `version` parameter greater than `0`.

### OnTopicUpdated

The client receives the `OnTopicUpdated` callback when any client makes a change to a data item on a topic (adding, deleting, or changing the value associated with it), and contains information about the change:

```
OnTopicUpdated(acb::TopicPtr topic, std::string name, std::string value, int version, bool deleted);
```

The `topic`, `name`, and `value` parameters are as detailed previously (the `value` parameter is the new value); `deleted` will be `true` if the data item has been removed from the topic (see **Deleting Data from a Topic below** for details about deleting data). The `version` parameter is an integer which increases with every change made to the value of the data item. It enables the client application to ignore updates for data items if those updates have values earlier than the value it currently has for the same data item.

## Deleting Data from a Topic

The client can delete the name-value pair from the topic by calling `acb::Topic::DeleteData (std::string key)`. The client will receive either an `OnDataDeleted` followed by an `OnTopicUpdated` callback, or an `OnDataNotDeleted` callback (in the case of failure).

## Sending a Message to a Topic

A client application can send a message to a topic, and have that message sent to all current subscribers to the topic.

```
std::string message = "a message";
topic->SendMessage(message);
```

If it successfully sends the message, the listener will receive an `OnTopicSent` followed by an `OnMessageReceived` callback, both containing the topic and the message; if it is not successful, the client will receive an `OnTopicNotSent` callback containing the topic, the message which failed, and an error message.

### OnMessageReceived

The `OnMessageReceived` callback is received by all connected clients when any client connected to the topic (including itself) successfully sends a message to the topic. The parameters include the `topic` and the `message` itself (as a `std::string`).

## Disconnecting from a Topic

The client application can disconnect from a topic:

```
topic->Disconnect();
```

or delete it altogether:

```
topic->Disconnect(true);
```

The optional parameter to the `Disconnect` method is a boolean which, if `true`, will cause the topic to be deleted from the server (the default value is `false`). The listener will receive either an `OnTopicDeleted` followed by an `OnTopicDeletedRemotely` callback, or an `OnTopicNotDeleted` callback. Apart from the topic which has been deleted, `OnTopicDeleted` includes a message parameter. The message is not particularly helpful, and is probably best ignored. The `OnTopicNotDeleted` callback also includes an `error` parameter (a `std::string`) which is more useful.

### OnTopicDeletedRemotely

`OnTopicDeletedRemotely` is received by all clients connected to a topic when it is deleted from the server, either as a result of any client calling `Disconnect(true)`, or as a result of it expiring. The only parameter it includes is the topic which has been deleted. Once a topic has been deleted, the client should not call any of that topic's methods (which will fail in any case), and should consider itself unsubscribed from that topic. If a topic with the same name is subsequently created, it is a new topic, and the client will not be automatically subscribed to it.

# Responding to Network Issues

As Remote Expert Mobile is network-based, it is essential that the client application is made aware of any loss of network connection. When a network connection is lost, the server uses SIP timers to determine how long to keep the session alive before reallocating the relevant resources. Any application you develop should make use of the available callbacks in the Remote Expert Mobile API, and any other available technologies, to handle network failure scenarios.

To receive callbacks relating to network issues, the application must use the `UCListener` class.

## Reacting to Network Loss

In the event of network connection problems, the SDK automatically tries to re-establish the connection. It will make seven attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to `OnConnectionRetry` on the `UCListener` precedes each of these attempts. The callback supplies the attempt number (as a `uint8_t`) and the delay in seconds before the next attempt (as a `uint16_t`) in its two parameters.

When all reconnection attempts are exhausted, the `UCListener` receives the `OnConnectionLost` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in **Creating the Session on page 4**.

If any of the reconnection attempts are successful, the `UCListener` receives the `OnConnectionReestablished` callback.

Of these three notifications, only `OnConnectionLost()` is required to be implemented by the application. The other two are optional.

**Note** The retry intervals, and the number of retries attempted by the SDK are subject to change in future releases. Do not rely on the exact values given above.

## Network Quality Callbacks

The application can implement the `OnInboundQualityChanged` method of the `CallListener` to receive callbacks on the quality of the network during a call:

```
void MyCallListener::OnInboundQualityChanged(int inboundQuality)
{
   // Show indication of quality
}
```

The `inboundQuality` parameter is a number between `0` and `100`, where `100` indicates perfect quality. The application might choose to show a bar in the UI, the length of the bar indicating the quality of the connection.

The SDK starts collecting metrics as soon as it receives the remote media stream. It does this every 5s, so the first quality callback fires roughly 5s after this remote media stream callback has fired.

The callback then fires every time a different quality value is calculated; so if the quality is perfect, there will be an initial quality callback with a value of `100` (after 5s), and then no further callback until the quality degrades.

**Note** The `CallListener` class provides a skeleton implementation (which does nothing) of `OnInboundQualityChanged`, so there is no need for the application to implement it.

CHAPTER 8

# Creating a Windows .NET Client Application



Remote Expert Mobile includes a wrapper for the library that works with native Windows applications that allows the creation of Windows applications that use the .NET Framework, such as C# or VB.NET.

The wrapper generally follows the native SDK library, provides similar functionality, and works with similar objects. It departs from the native SDK in some areas in which it makes sense for the wrapper to adopt a convention that works better with the .NET Framework. Managed versions of the classes provided by the wrapper are contained within the namespace `CSDKCLR` (CSDK Common Language Runtime) and are prefixed by CLI_ to indicate that they use the Common Language Infrastructure.

Interfaces are prefixed by ICLI_ in keeping with conventional naming of interfaces used by the .NET framework.

The examples are expressed in the C# language.

## Setting up a Project

**Note** Before following this process, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **Creating the Web Application on page 3**.

This section provides guidance on setting up a project using Visual Studio 2013 that imports the CSDK-CLR wrapper library.

1. Create a new .NET project (typically a Visual C# Windows Forms application or a Visual Basic Windows Forms application).
2. Within Solution Explorer (use the View menu to display it if it is not shown), open the project node, then open its **References** node.
3. Right-click on the **References** node and click on **Add reference...**.
4. The SDK contains a file `CSDK-CLR.dll`. Click on the **Browse...** button, browse to the file and click on **Add**.

# Initializing the CLI_UC and Starting the Session

The application initially accesses the API via a single object, `CLI_UC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see **Creating the Web Application on page 3**), and initialize the `CLI_UC` object using it. Once it has received the Session ID, the client application must create the `CLI_UC` object, then call its `StartSession` method:

```
public sealed class UCOwner : CSDKCLR.ICLI_UCListener
{
   private CSDKCLR.CLI_UC mUC;
   public void MakeTheCliUcObject(String sessionID, String stunServers)
   {
      mUC = new CSDKCLR.CLI_UC(sessionID, stunServers, this);
      mUC.StartSession();
   }
   /// The class needs to implement ICLI_UCListener
   /// so that it can act upon its callbacks.
}
```

If the callbacks are defined by a different object that implements `ICLI_UCListener`, then you should substitute a reference to that object for `this` in the call to the `CLI_UC` constructor.

Note that you must dispose of the `CLI_UC` object, in common with many of the objects provided by the CLI wrapper, when it is no longer useful. Do this by calling its `Dispose` method:

```
mUC.Dispose();
```

The object that implements the `ICLI_UCListener` interface has to provide implementations for the following callbacks:

```
void OnSessionStarted();
void OnSessionNotStarted();
```

One of the two above functions is called asynchronously to indicate the success or failure of the session starting operation. The other function callbacks in the interface are self-explanatory:

```
void OnConnectionRetry(byte attemptNumber, ushort delayInSeconds);
void onConnectivityLost();
void OnConnectionReestablished();
void OnUnknownWebsocketMessage(string s);
```

# Adding Voice and Video

Once the application has created the `CLI_UC` object, it can retrieve the `CLI_Phone` object from it, and use it to make or receive calls, which are represented by `CLI_Call` objects. Each of the `CLI_UC`, `CLI_Phone` and `CLI_Call` objects have corresponding listener interfaces that the application can implement in order to receive error notifications and other events.

## Adding a Preview Window before a Call is made

If you want to add a preview window (a window which displays the video which is being sent to the other endpoint) before a call is established, you can call the `Phone.SetPreviewViewName` method. An

appropriate time to do this is in the `OnSessionStarted` callback:

```
public void OnSessionStarted()
{
   mUC.GetPhone().SetPreviewViewName("local");
};
```

# Making a Call

Once you have created the session, you can start making calls. This is simply a case of asking the `CLI_Phone` object to create a `CLI_Call`:

```
CLI_Call call = mUC.GetPhone().CreateCall("1234",
   CLI_MediaDirection.SEND_AND_RECEIVE,
   CLI_MediaDirection.SEND_AND_RECEIVE, this);
```

In this example, `1234` is the number we are dialing, the two `CLI_MediaDirection` parameters indicate that we want the call to be an audio and video call respectively, and the final parameter registers `this` as the `ICLI_CallListener`.

**Note** There is an alternative version of `CreateCall`, which does not take the final `ICLI_CallListener` parameter. If you use this version, you will need to call `SetListener` on the resulting `CLI_Call` object in order to receive notifications about the call.

The `CreateCall` function returns the newly created call as a `CLI_Call` object, but the application should not use it until the `ICLI_CallListener` receives an asynchronous notification indicating that the call creation has succeeded.

You can change the values of the media direction parameters to make the call audio only or video only. They must be members of the `CLI_MediaDirection` enumeration:

- `NONE`
- `SEND_ONLY`
- `RECEIVE_ONLY`
- `SEND_AND_RECEIVE`

**Note** The older form of `CreateCall`, which took two boolean values, is now deprecated.

# Receiving a Call

REM Advanced SDK will invoke the `ICLI_PhoneListener` object that you registered with the `CLI_Phone` object when an incoming call is received:

```
public void OnIncomingCall(CLI_Call call)
{
   call.Answer(MediaDirection.SEND_AND_RECEIVE, MediaDirection.SEND_AND_
   RECEIVE);
}
```

In this example the application auto-answers the call; most applications will notify the user before invoking either `call.Answer()` to accept, or `call.End()` to reject, the call.

The two parameters to the `Answer` method indicate that whether or not we want to answer the call with audio and video respectively. They are members of the `CLI_MediaDirection` enumeration:

- `NONE`
- `SEND_ONLY`

- RECEIVE_ONLY
- SEND_AND_RECEIVE

**Note** The older form of `Answer`, which took two boolean values, is now deprecated.

## Displaying Video

Your application can display video received from the remote peer and a preview of the locally-captured video that is sent to the remote peer.

The SDK uses named pipes internally to route frame information from a receiving (producer) thread to a consuming thread, These activities are separated within the SDK and you need to connect them to allow video to be displayed.

The `CLI_Phone` object has a method, `SetPreviewViewName` (associated with local video) and the `CLI_Call` object has a method, `SetVideoViewName` (associated with remote video). Each takes a string that comprises part of the name of the pipe. File naming conventions should be followed when assigning a name to each string. This prepares the sources of the pipes.

In order to display the video, you need to extend `CLI_ImageDataPipe`. The constructor of `CLI_ImageDataPipe` takes a string whose name needs to match the name provided to `SetPreviewViewName` or `SetVideoViewName`. You need to override the method `SendImageToView` which returns `void` and takes no arguments. REM Advanced SDK calls it whenever it receives a new frame from the pipe and makes it available for painting.

To obtain the frame, call the method `GetImageData`, which returns a `CLI_VideoViewImageData` object that contains the image data. You can render the frame onto a `CLI_VideoView` object. The `CLI_VideoView` has a method, `SetImageData`, that takes the `CLI_VideoViewImageData` object that came from `GetImageData` as its argument; it associates the image data with the `CLIVideoView` object. It also has a method, `Paint`, which renders the associated frame data - the window's `onPaint` method should call it, passing in its `PaintEventArgs` object.

When each frame is ready to display, the `CLI_VideoView` calls an `ICLI_ViewRefresher` object's `RefreshViewFunc`. You can associate an `ICLI_ViewRefresher` with the `CLI_VideoView` by calling its `SetRefreshViewFunc` method. The usual behavior of the `RefreshViewFunc` method is to invalidate the region of the client area which renders the video.

## Muting Local Audio and Video Streams

Muting the local streams stops audio or video from being sent to the remote party. Audio and video received from the remote party is not affected.

The following example shows an application muting both audio and video in response to a mute button being pressed in the user interface:

```
public void MyUIController.MuteButtonPressed()
{
   call.EnableLocalAudio(false);
   call.EnableLocalVideo(false);
}
```

The audio and video streams can also be muted as soon as the call is answered as follows:

```
call.Answer(CLI_MediaDirection.RECEIVE_ONLY, CLI_MediaDirection.RECEIVE_ONLY);
```

## Holding and Resuming a Call

Your application can place a call on hold, and subsequently resume the call. When a call is on hold, no audio or video is played to either end of the call. Only the party that placed the call on hold can resume it:

```
public void MyUIController.HoldButtonPressed()
{
   call.Hold();
```

```
}
public void MyUIController.ResumeButtonPressed()
{
   call.Resume();
}
```

## Sending DTMF Tones

Your application can send DTMF tones on a call by using the `SendDTMF` method provided by the `CLI_Call` object. It can be used as follows:

```
call.SendDTMF("#123*", true);
```

The example sends five tones sequentially. To send a single tone, use a single-character string.

The `boolean` parameter indicates whether you want the tones to also be played locally, i.e. to be audible to the user of your application.

Valid values in the string parameter are those conventionally used to denote DTMF tones: `0123456789#*`. A comma character inserts a two-second pause into a sequence of tones.

## Handling Multiple Calls

Applications developed with the Remote Expert .NET SDK do not support multiple simultaneous calls.

## Setting Video Resolution

Your application can configure both the resolution and frame rate of the video that is sent to the remote party in a video call. You can set the video resolution on the `CLI_Phone` object:

```
phone.SetPreferredCaptureResolution(CLI_VideoCaptureResolution.RESOLUTION_
1280x720);
```

Available options at the time of writing are

- `RESOLUTION_AUTO`
- `RESOLUTION_352x288`
- `RESOLUTION_640x480`
- `RESOLUTION_1280x720`

You can also set the frame rate:

```
phone.SetPreferredCaptureFrameRate(20);
```

Note that changes to the video resolution and frame rate apply only to new calls. Calling these APIs while a call is in progress has no effect.

## Monitoring the State of a Call

During call setup, the call will transition through several states, from the initial setup to being connected with media available (or failure). You can monitor these states using the `ICLI_CallListener` object using the method `OnStatusChanged(CLI_CallStatus newStatus)`. `CLI_CallStatus` is an enumeration with the following values:

| Status Code | Meaning |
|---|---|
| `UNINITIALIZED` | The `Call` object has been created but not initialized |
| `SETUP` | Call is in process of being set up |
| `ALERTING` | An incoming call is alerting (ringing) at the user's end. |
| `RINGING` | An outgoing call is ringing at the remote end |
| `MEDIA_PENDING` | The call is connected, and waiting for media |
| `IN_CALL` | The call is fully set up, including media |
| `BUSY` | Dialed number is busy |

| Status Code | Meaning |
|---|---|
| NOT_FOUND | Dialed number is unreachable or does not exist |
| TIMED_OUT | The dialing operation timed out without a response from the dialed number |
| CALL_ERROR | The call has errored. This may be because something (such as a media broker) was unavailable, or because of network conditions, or for some other reason. More information is not available. |
| ENDED | The call has ended |

There are also other callback functions that allow your application to detect call quality changes (`OnInboundQualityChange` which provides a number from `0` to `100` representing the call quality; `100` is best) and dial or call failures (`OnDialFailed` and `OnCallFailed`).

# Adding Application Event Distribution

*Application Event Distribution* is contained in the `CLI_AED` and `CLI_Topic` objects. Asynchronous notification of events occurs by callback to any class that implements the `ICLI_TopicListener` interface.

In order to use Application Event Distribution, you must first obtain a `CLI_AED` object from the `CLI_UC` object:

```
CLI_AED aed = uc.GetAED();
```

Initialization of the `CLI_UC` object is exactly the same as for voice and video calling (see **Initializing the CLI_UC and Starting the Session on page 86**).

## Creating a Topic

You can create a topic using the `CreateTopic` API provided by the `CLI_AED` object as follows:

```
aed.CreateTopic(name, listener);
```

or

```
aed.CreateTopic(name, expiry, listener);
```

The `name` argument is a string that uniquely identifies the topic on the server. If you use the form of the API that includes an `expiry` argument, the expiry period is in minutes. In that case, the topic will be removed from the server after the topic has been inactive for that period of time. When created without an expiry time (by the first method), the topic exists indefinitely, and needs to be deleted explicitly.

The `listener` is an object that implements `ICLI_TopicListener`, the methods of which provide notification to your application of events that concern the topic.

If the topic identified by the name argument already exists on the server, you are connected to it.

### OnTopicConnected

After creating a topic, the listener should receive an `OnTopicConnected` callback (in case of failure, the listener receives an `OnTopicNotConnected` callback instead). The arguments to the callback are a `CLI_Topic` identifying the topic and a `CLI_TopicData` that contains all existing data for the topic.

The data contained within a topic consists of key-value pairs. If you know the name of a key for which you want the associated value, you can retrieve it as follows:

```
CLI_TopicDataValue value = topicData.GetValue(name);
```

where the `name` argument is a string containing the key name.

You can also iterate through the keys with a simple `foreach` loop:

```
foreach (CLI_TopicDataElement element in topicData)
{
   string key = element.key;
   CLI_TopicDataValue value = element.value;
   int version = element.version;
```

```
    bool deleted = element.deleted;
}
```

A method that is probably most useful during testing and debugging is `CLI_TopicDataElement.ToString` which in the case of `CLI_TopicDataElement` objects is overridden to provide a readable representation of the object state.

`CLI_TopicDataValue` wraps an `acb::TopicDataValue` object. It contains at most one value that is a `string`, a `double`, an `int`, or a `bool`. If it doesn't contain a value, it represents an empty value. Note that only `string` values can be set using the current API.

The type is returned in string form by using `CLI_TopicDataValue.GetType()` which returns a `string` identifying the type. You can obtain the value using `GetAsString(defaultValue)`, `GetAsBool()`, `GetAsInt(defaultValue)` or `GetAsDouble(defaultValue)` and if the value does not exist, the default value is returned (or `false` in the case of `GetAsBool()`). You can also call `GetAsInt()` or `GetAsDouble()` in which case the default value returned is `0`.

# Publishing Data to a Topic

Once the client application has created and/or connected to a topic, it can publish data to the topic. Data consists of key-value pairs.

```
string key = "name";
string value = "value";
topic.SubmitData(key, value);
```

Having submitted the data, the listener receives either an `OnTopicSubmitted` or an `OnTopicNotSubmitted` (in the case of failure) callback. Both the `key` and the `value` are values of type `string`. In the case of a successful submission, an `OnTopicUpdated` callback is also triggered when the data is sent to all clients connected to the topic; see **OnTopicUpdated below** for details.

The client application can also change the value of an existing data item by calling `SubmitData`. In this case, the callbacks (`OnTopicSubmitted` and `OnTopicUpdated`) include a `version` parameter greater than `0`.

## OnTopicUpdated

The client receives the `OnTopicUpdated` callback whenever any client makes a change to a data item on a topic (adding, deleting or changing the associated value) and contains information about the change:

```
OnTopicUpdated(CLI_Topic topic, string key, string value, int version, bool deleted);
```

The `topic`, `name`, and `value` parameters are as detailed previously. The `version` parameter enables clients to know, when they receive this callback, whether it refers to data they have just submitted (if it does, the `version` parameters are equal). For a newly-created data item, the version is `0`, and it is incremented upon every change.

# Deleting Data from a Topic

The client can delete a key-value pair from a topic by calling `CLI_Topic.DeleteData("keyName")`. The client receives either an `OnDataDeleted` callback followed by an `OnTopicUpdated` callback, or an `OnDataNotDeleted` callback (in the case of failure).

# Sending a Message to a Topic

A client application can send a message to a topic and have that message sent to all current subscribers to the topic.

```
topic.SendMessage("message");
```

If it successfully sends the message, the listener receives an `OnTopicSent` callback followed by an `OnMessageReceived` callback, both containing the topic and the message. If it is not successful, the client receives an `OnTopicNotSent` callback containing the `topic`, the `message` which failed, and an `error` message.

### OnMessageReceived

The `OnMessageReceived` callback is received by all connected clients when any client connected to the topic (including itself) successfully sends a message to the topic. The arguments include the topic and the message itself (as a `string`).

## Disconnecting from a Topic

The client application can disconnect from a topic:

```
topic.Disconnect();
```

or delete it altogether:

```
topic.Disconnect(true);
```

If the `Disconnect` method is called with the boolean argument set to `true`, the topic is deleted from the server. Calling `Disconnect` without an argument is equivalent to calling `Disconnect(false)`, and in either case the topic is not deleted.

If the application attempts to delete the topic from the server, the listener receives an `OnTopicDeleted` callback followed by an `OnTopicDeletedRemotely` callback if successful, or an `OnTopicNotDeleted` callback otherwise. `OnTopicDeleted` includes a message string as one of its arguments which is not likely to be useful. `OnTopicNotDeleted` includes a message string that provides information about the error.

### OnTopicDeletedRemotely

`OnTopicDeletedRemotely` is received by all clients connected to the topic when it is deleted from the server either as a result of any client calling `Disconnect(true)`, or because it expires. It passes the `CLI_Topic` which has been deleted. Once a topic has been deleted, the client should not call any of the topic methods and should consider itself unsubscribed from that topic. If a topic with the same name is subsequently created, it is a new topic, and the client is not automatically subscribed to it.

# Responding to Network Issues

As Remote Expert Mobile is network-based, it is essential that the client application is made aware of any loss of network connection. When a network connection is lost, the server uses SIP timers to determine how long to keep the session alive before reallocating the relevant resources. Any application you develop should make use of the available callbacks in the Remote Expert Mobile API, and any other available technologies, to handle network failure scenarios.

To receive callbacks relating to network issues, the application must implement the `ICLI_UCListener` interface.

## Reacting to Network Loss

In the event of network connection problems, the SDK automatically tries to re-establish the connection. It will make seven attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to `OnConnectionRetry` on the `ICLI_UCListener` precedes each of these attempts. The callback supplies the attempt number (as a `System::Byte`) and the delay in seconds before the next attempt (as a `System::UInt16`) in its two parameters.

When all reconnection attempts are exhausted, the `ICLI_UCListener` receives the `OnConnectionLost` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in **Creating the Session on page 4**.

If any of the reconnection attempts are successful, the `ICLI_UCListener` receives the `OnConnectionReestablished` callback.

**Note** The retry intervals, and the number of retries attempted by the SDK are subject to change in future releases. Do not rely on the exact values quoted above.

## Network Quality Callbacks

The application can implement the `OnInboundQualityChange` method of the `ICLI_CallListener` interface to receive callbacks on the quality of the network during a call:

```
public void OnInboundQualityChange(int inboundQuality)
{
   // Show indication of quality
}
```

The `inboundQuality` parameter is a number between `0` and `100`, where `100` indicates perfect quality. The application might choose to show a bar in the UI, the length of the bar indicating the quality of the connection.

The SDK starts collecting metrics as soon as it receives the remote media stream. It does this every 5s, so the first quality callback fires roughly 5s after this remote media stream callback has fired.

The callback then fires every time a different quality value is calculated; so if the quality is perfect then there will be an initial quality callback with a value of 100 (after 5s), and then no further callback until the quality degrades.

# Appendix: Error Codes

Some of the APIs give access to an error code from the Gateway. These are as follows:

| Code | Meaning |
|---|---|
| `NOMATCH` | The `Offer`, `Answer`, or `Offer Request` is not for a new session and does not correspond to an existing session |
| `REFUSED` | The initial `Offer` was refused |
| `BUSY` | There is already an `Offer` or `Offer Request` pending, so this `Offer` cannot be handled |
| `TIMEOUT` | The outbound request failed because of a SIP timeout |
| `NOTFOUND` | The outbound request failed because the callee could not be found |
| `FAILED` | The request failed for some other reason. |

# Acronym List

| Item | Description |
|---|---|
| **CODEC** | “Coder-decoder" encodes a data stream or signal for transmission and decodes it for playback in voice over IP and video conferencing applications. |
| **CSDK** | Remote Expert Mobile Client SDKs. Includes three distinct SDKs for iOS, Android and web/JavaScript developers. |
| **REAS** | Remote Expert Mobile Application Server |
| **REMB** | Remote Expert Mobile Media Broker |
| **UC** | Unified Communications |
| **WebRTC** | Web Real Time Communications for communications without plug-ins |