Cisco Remote Expert Mobile Version 11.5(1)

Advanced CSDK Developer's Guide

First Published: 2016-08-10

Preface

Change History

Changes	Date
Initial release	2016-08-10

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 *Cisco Collaboration Systems Solution Reference Network Designs (SRND)*. To review IP Telephony terms and concepts, see the documentation at the preceding link.

Organization of This Guide

This guide includes the following sections:

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

Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, using the Cisco Bug Search Tool (BST), submitting a service request, and gathering additional information, see *What's New in Cisco Product Documentation* at: http://www.cisco.com/c/en/us/td/docs/general/whatsnew/whatsnew.html.

Subscribe to *What's New in Cisco Product Documentation*, which lists all new and revised Cisco technical documentation, as an RSS feed and deliver content directly to your desktop using a reader application. The RSS feeds are a free service.

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco website at www.cisco.com/go/offices.

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
bold font	Commands and keywords and user-entered text appear in **bold** font.
italic font	Document titles, new or emphasized terms, and arguments for which you supply values are in *italic* font.
[ ]	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 nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.
`courier font`	Terminal sessions and information the system displays appear in `courier` font.
< >	Nonprinting characters such as passwords are in angle brackets.
[ ]	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.

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 Mobile—Documentation Guide*

Creating a web application

Before you develop a client application using the RE Mobile Client SDK (CSDK), it is essential that you create a web application. The web application does the following:

- Authenticates users and determines the services available to them.
- Creates sessions on the Web Gateway.
- Ends sessions on the Web Gateway.

Passing context data

Context data can be passed from the client application into the contact center when a support session is started. The context information can be included in the configuration object passed to the Assist CSDK `startSupport` method, or can be supplied by the server-side web application when creating a session. Typically, an application creates and manages a session itself when the context information contains sensitive data that should not be exposed on the client, for example an account number.

The context data value specified is placed in the SIP `User-to-User` header. It is the responsibility of application to encode the value and add the appropriate encoding parameter. The encoding is typically `hex`—as in the following example:

```
6369643d6173736973742d636f7272656c6174696f6e2d6964;encoding=hex
```

Note: The UUI can only be used when **Anonymous Consumer Access** is set to **Trusted**.

Creating a session

Important: Before a session is created, the web application must authenticate the user. The authentication method is entirely at your discretion—RE Mobile Client SDK (CSDK) offers no restrictions on the method used.

Once the user is authenticated, the web application provisions a session by sending an HTTP/HTTPS POST request to REAS describing the capabilities required.

This session provisioning request must contain a valid Web Application ID. The Web Application ID is a unique text string that identifies the web application to REAS, and confirms that the web application is allowed to create sessions. The Web Application ID is configured on the Web Gateway—see the *Cisco Remote Expert Mobile—Install and Config Guide*.

The session provisioning request must be POSTed to one of the following URLs:

- For non-secure communication:
  http://<web_gateway_address>:8080/gateway/sessions/session
- For secure communication:
  https://<web_gateway_address>:8443/gateway/sessions/session

Note: The message contains the Web Application ID, so we recommend that this transaction is performed over HTTPS for security.

Once the session is created, the REAS responds with a JSON string containing configuration data for the client, which includes a Session ID for the new session, and the connection details for the REAS cluster; this token needs to be passed to the client application, which uses it when starting a support session.

The body of the POST message must be formatted in JSON; see the following example:

```
{
    "timeout":1,
    "webAppId": "session-creation-service",
    "allowedOrigins": ["example.com"],
    "urlSchemeDetails": {
        "host": "re-mobile.example.com",
        "port": "443",
        "secure": true
    },
    "voice": {
        "username": "assist-correlation-id",
        "displayName": "An Anonymous User",
        "domain": "re-mobile.example.com",
        "inboundCallingEnabled": false
    },
    "allowedOutboundDestination": "destination",
    "uuiData": "abcdefhijk;encoding=hex",
    "additionalAttributes": {
        "AED2.metadata": {
            "role": "consumer"
        },
        "AED2.allowedTopic": "assist-correlation-id"
    }
}
```

Where the parameters are defined as follows:

- `timeout`—The timeout period for the Session, defined in minutes. If omitted, this is set to `1` by default.
- `webAppId`—The Web Application ID that has been defined, which is 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.
- `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.
- `urlSchemeDetails`— This defines the domain part of the URL that a client application should use to connect to the REAS server. This is typically an FQDN which resolves to a public IP address, from which an inbound connection is enabled using a reverse proxy. If these details are not provided, the default setting for each option is used:
    - `secure`—If `true`, connects using secure WebSockets (wss). The default value is `false`, for non-secure (ws).
    - `host`—Specifies the hostname or IP address for the WebSocket to connect to. If not provided, the client uses the `<reas_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 Web Gateway. This value should be set to the external hostname or IP address.
    - `port`—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.
    - `username`—The SIP user name, as would appear in the `From` header. This is used as a `correlation-id` so that the consumer and agent sides of the call join the same assist session.

      Note: If the application is generating its own `correlation-id` it must be prefixed with the string `assist-` and followed by a 25-character unique alphanumeric string, for example, `assist-m2v7r3jpb0jsk5j28ok4b5o4s`
    - `displayname`—The SIP display name, as it would appear in SIP messages. If this is omitted, no display name is set for the user.
    - `domain`—The corresponding SIP domain. This should be set to the domain of the SIP entity to which REAS will send the `SIP INVITE` request.

- `InboundCallingEnabled`—Set inbound calling parameters to enable inbound calling. If this is omitted, inbound calling is enabled by default.

  Note: We recommend that this parameter is always set to `false` for REM.

  If `inboundCallingEnabled` is set to `true`, a `SIP REGISTER` request is sent to the internal SIP network; therefore, a corresponding user endpoint must exist on the internal SIP network. This user's credentials should be entered in the `auth:` section of the POST request to the Web Gateway.

  If `inboundCallingEnabled` is set to `false`, a `SIP REGISTER` is not sent.

- `AllowedOutboundDestination`—This can be a single destination, for example `sip:bob@example.com` or can be the string `all` to allow unrestricted calling.

- `Auth`—The authentication credentials for Voice and Video calling. This section can be omitted if the Web Gateway is a trusted entity in the SIP infrastructure; however, if it is omitted and the SIP is challenged, the registration fails.

  - `Username`—The user name you register with. This is a mandatory setting for voice calling.
  - `Password`—The password used for registrations. This is a mandatory setting for voice calling.
  - `Realm`—The realm used for registrations.

    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.

  - `AccessibleSessionIdRegex`—The java regex of sessionIds whose sessions a user is allowed to access.
  - `MaxMessageAndUploadSize`—Limits the size of message (in bytes) a user can send and the size (in bytes) of an individual data upload.
  - `DataAllowance`—The total data (in bytes) a user can have stored at any time.

- `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, if the value of this parameter is "`ABCD`", the CSDK adds the header "`User-to-User: ABCD`". If this parameter is omitted, no `User-to-User` header is added to SIP messages. See the following examples of valid uuiData values:

  ```
  abcdefhijk;encoding=hex
  abcdefghijk;encoding=blah;paramname=paramvalue
  "abcdefghijk";encoding=blah
  ```

- `additionalAttributes`—Additional attributes required for an Assist session.

  - `AED2.metadata.role`—This must be set to `consumer`.
  - `AED2.allowedTopic`—This must be the correlation id that is used in the `voice.username` attribute described above.

sessionid

A valid request returns a JSON object containing a `sessionid` text string that includes all the required configuration details. If the submitted JSON contains properties with names that are unknown to REAS, a list of those unknown properties is placed in `unknownProperties`. This property is omitted if there are no unknown properties:

```
{
"sessionid" : "<very long string...>",
"unknownProperties" : ["<p1>","<p2>",...]
}
```

This `sessionid` is used throughout your session, for example see Initializing the CSDK on page 11.

timeout

The timeout parameter determines the maximum allowed time interval (measured in minutes) that the client application may be unreachable from the Web Gateway. This parameter is optional—if omitted, the timeout defaults to 1 minute.
The timeout affects the web socket in the following ways.

- It defines the time within which a client application must use the session token ('sessionid') to establish its initial connection with the Web Gateway. If the client does not connect to the provisioned session within this time, the Web Gateway discards the session, and the session token is considered stale.
- The web socket may be interrupted (for example, due to a network outage) after having been established. If the client re-establishes the connection within the timeout period, the Web Gateway keeps the session alive. If the client does not re-establish the connection within the timeout, the Web Gateway deletes the session, and un-registers from the services the client had previously been connected (for example, registrar).

Example

See the following example of POST messages for how to start a session for use with REM Assist:

```
{
    "timeout":1,
    "webAppId": "session-creation-service",
    "allowedOrigins": ["example.com"],
    "urlSchemeDetails": {
        "host": "re-mobile.example.com",
        "port": "443",
        "secure": true
    },
    "voice": {
        "username": "assist-m2v7r3jpb0jsk5j28ok4b5o4s",
        "displayName": "An Anonymous User",
        "domain": "re-mobile.example.com",
        "inboundCallingEnabled": false
    },
    "allowedOutboundDestination": "sip:dest@example.com",
    "uuiData": "abcdefhijk;encoding=hex",
    "additionalAttributes": {
        "AED2.metadata": {
            "role": "consumer"
        },
        "AED2.allowedTopic": " assist-m2v7r3jpb0jsk5j28ok4b5o4s "
    }
}
```

Mandatory fields

In the session creation JSON the following validation rules are applied:

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

Ending the session

To end the session, the web application needs to send a `DELETE` request containing the Session ID to REAS at one of the following URLs:

- For non-secure communication:
  http://<reas_address>:8080/gateway/sessions/session/id/<session_ID>
- For secure communication:
  https://<reas_address>:8443/gateway/sessions/session/id/<session_ID>

Note: This tears down an active support call and invalidates the session.

Communication with the client

While the RE Mobile Client SDK (CSDK) 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.

It is essential that the web application is able to pass the token containing the Session ID created by the Web Gateway to the client. However, you can choose whether this is using a REST API, HTML, or any other method. We recommend sending using an encrypted connection.

Creating a Browser Application

RE Mobile Client SDK (CSDK) enables you to develop browser-based applications offering users the following methods of communication:

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

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

CSDK 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 the following URL:

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

Note: RE Mobile Client SDK (CSDK) is delivered with a sample application. This is available at the following URL:

`<installation_directory>/Core_SDK/Sample_Source`

All samples featured in this guide are in the same location.

Setting up a project

Note: Before following this process, make sure that you have followed the steps described in Creating a session on page 6.

In your development application of choice, start a new project and create the page to use to deliver your client application. This page needs to contain some script tags containing the RE Mobile Client SDK (CSDK) JavaScript API:

```
<script src="http://<gateway_address>:<port>/gateway/adapter.js"></script>
<script src="http://<gateway_address>:<port>/gateway/client-sdk.js"></script>
```

`client-sdk.js` implements Voice and Video calling, and AED; if you only want to implement a subset of this functionality, include only the scripts for the functionality that you require:

- `csdk-phone.js` for Voice and Video calling.
- 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 want to avoid prompting the user for access to their microphone and camera, ensure that `csdk-phone.js` is not included (either directly or indirectly).

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

Initializing the CSDK

To set up all the functionality to which the user has access, you need to initialize `UC` using the Session ID provided by the Web Gateway when the session was created. Once the Session ID is received by the client, the `start` method must be called on the `UC`.

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

```
//Get hold of the sessionID however your app needs to
var sessionID = getMySessionID();

//Set up initialization success callback before calling start
UC.onInitialised = function() {
//perform tasks associated with successful
//initialization such as registering listeners on UC
//objects
};

//Set up initialization failure callback before calling start
UC.onInitialisedFailed = function() {
//perform tasks associated with initialization failure
//such as logging the user out of the web app. Note
//that the client should make a call to the controlling
//web application so that it can ensure that the
//session is ended on the Web Gateway.
};

UC.start(sessionID);
```

The failure to register for any available functionality, for example, Voice and Video calls, results in the whole session ending. In these situations the user should end and create the session again to start a new session
(see Creating a session on page 6).

To use any of the RE Mobile Client SDK (CSDK) functionality, you need to call appropriate framework functions. These are documented in the JavaScript API javadocs. All of the functionality to which the user has access is available under the `UC` object.

Responding to network issues

As RE Mobile Client SDK (CSDK) 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 CSDK API, and any other available technologies, to handle network failure scenarios.

Reacting to network loss

In the event of network connection problems, the RE Mobile Client SDK (CSDK) automatically makes several attempts to re-establish the connection. These attempts are made at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. Each of these attempts is preceded by a `onConnectionRetry(attempt, delayUntilNextRetry)` method on the `UC` object.

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

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

If the network connection to the web has been lost, the `UC` object receives an `onNetworkUnavailable` callback. When the connection comes back, `onConnectionRetry` fires.

Note: The retry intervals, and the number of retries attempted by the CSDK, may change in future releases; do not rely on the exact values quoted above.

Network quality callbacks

During a call the application can receive callbacks on the quality of the network.

The Call object has the following method to support this:

```
       /**
            * Callback for when the quality of the connection from the remote end    * of the call changes.
            *
            * @param connectionQuality a number between 0 and 100 representing the
            *  current connection quality, where 100 is perfect
             */
            onConnectionQualityChanged: function(connectionQuality) {
                    console.log("onConnectionQualityChanged: The remote stream quality is now: " +
   connectionQuality);
            }
```

The CSDK 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 the remote media stream callback has been established.

The callback is then fired 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.

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 stream

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

The `setLocalMediaEnabled()` method supports the following boolean parameters:

- `enableVideo` to enable a stream for the user's camera or webcam.
- `enableAudio` to enable a stream from the user's microphone.

To support screen sharing, `setLocalMediaEnabled()` also supports a single parameter JavaScript object in which the audio, video and screensharing capabilities are set. This object consists of the following boolean parameters: `audio, video` and `screenshare`, for example:

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

Note: Audio and video calling can be enabled using either method, however screensharing can only be enabled using the JavaScript object.

The `onScreenshareError()` callback is triggered if the client application fails to acquire screenshare media.

Checking Browser Compatibility

`UC.checkBrowserCompatibility(pluginInfoCallback)` checks the browser for compatibility with UC. This function is asynchronous, the function `pluginInfoCallback` is called when the information is ready.

`pluginInfoCallback` is called with an argument (pluginInfo) defined as follows:

- pluginInfo.pluginRequired—Boolean
  - `true`—The browser requires a plug-in for UC to operate correctly
  - `false`—No browser plug-in is required
- pluginInfo.status—String
  - `''` *(zero length string)*—When `pluginInfo.pluginRequired` is false
  - `installRequired`—The plug-in is missing and a plug-in install is required
  - `upgradeRequired` —A plug-in is present but an upgrade of it is required
  - `upgradeOptional` —A valid plug-in is installed but an upgrade is available
  - `upToDate`—The latest plug-in is already installed
- pluginInfo.restartRequired—Boolean
  - `true`—If an install is performed, the browser will require a restart
  - `false`—If an install is performed, the browser will not require a restart (also when `pluginInfo.pluginRequired` is false)
- pluginInfo.installedVersion—String
  - `none`—When the plug-in is missing (also when `pluginInfo.pluginRequired` is false)
  - `<x>.<y>.<z>`—Where <x>, <y>, and <z> are integers
- pluginInfo.minimumRequired—String
  - `none`—When no plug-in is required (`pluginInfo.pluginRequired` is false)
  - `<x>.<y>.<z>`—Where <x>, <y>,and <z> are integers. This is the minimum version of the plug-in that is required for UC to operate correctly.
- pluginInfo.latestAvailable—String
  - `none`—When no plug-in is required (pluginInfo.pluginRequired is false)
  - `<x>.<y>.<z>`—Where `<x>`, `<y>`, and `<z>` are integers. This is the latest version of the plug-in available on the server, and the version addressed by `pluginInfo.pluginUrl`
- pluginInfo.pluginUrl—String
  - `''` *(zero length string)*—When `pluginInfo.pluginRequired` is false
  - `<url>`—Where <url> is the URL of the latest browser plug-in

`UC.start` assumes the presence of a correct browser plug-in (if required). If this is not the case, an error may occur.

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 boolean parameters:

- `withAudio`—to add an audio stream to the call.
- `withVideo`—to add a video stream to the call.

Passing two `false` values for these parameters results in an `onDialFailed()` error method. If no parameters are passed, an audio and video call is dialled to ensure backwards compatibility.

Note: `call.dial()` must only be called once the CSDK has been initialized and the `UC.onInitialised` callback has been called. See Initializing the CSDK on page 11.

```
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);
//Specify where preview and remote video should be played/presented.
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();
};
```

If any issues occur when making a call, it is recommended that the following error methods should be overridden to inform the user of the call status:

- 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 the client application to be notified once a call is received. This passes a `call` object which 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) {
//Specify where preview and remote video should be played/presented.
call.setPreviewElement(previewVideoElement);
call.setVideoElement(remoteVideoElement);

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;

//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 boolean parameters:

- `withAudio`—to add an audio stream to the call.
- `withVideo`—to add a video stream to the call.

Passing two `false` values for these parameters results in an `onCallFailed()` error method. If no parameters are passed, a call is answered with audio and video to ensure backwards compatibility.

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

Muting the local audio and video streams

During a call, it is possible to mute and unmute the local audio and video streams separately. Muting the stream stops that stream being sent to the remote party, but the remote party's stream continues to play locally.

To mute either stream, use the `setLocalMediaEnabled(enableVideo, enableAudio)` methods of the `Call` object to toggle the audio and video streams.

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.

Handling multiple calls

Applications developed with RE Mobile Client SDK (CSDK) 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 14).
- To receive incoming calls while another call is in progress, the `UC.phone.onIncomingCall` method should be triggered (see Receiving a call on page 14).

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 provided call object's `onEnded` method.

Setting Video Resolution

The RE Mobile Client SDK (CSDK) JavaScript supports configuring the captured, and hence sent, video resolution for video calls. One of a set of video resolutions can be selected and applied to the capture device. Additionally the frame rate for capture can be configured. When a resolution and frame rate are specified, the CSDK 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 the Possible Resolutions

The list of possible resolutions is available from the Phone object using the "`videoresolutions`" array:

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

These values are an enumeration which list all supported resolutions.

The supported resolutions are:

Enumeration Value	Resolution (pixels)
`videoCaptureResolution174x144`	`width: 174, height: 144`
`videoCaptureResolution352x288`	`width: 352, height: 288`
`videoCaptureResolution320x240`	`width: 320, height: 240`
`videoCaptureResolution640x480`	`width: 640, height: 480`
`videoCaptureResolution1280x720`	`width: 1280, height: 720`

Note: When you set the resolution, the user's camera may not support that resolution. In that case, the browser provides a different resolution—Google Chrome currently defaults to a resolution of 640x480 if the requested resolution is not available, and Firefox currently only supports 640x480 resolution.

Setting the Resolution

The captured video resolution can be set using the `setPreferredVideoCaptureResolution(resolution)` method of the phone object. The value supplied must be one of the video resolutions presented in the "`videoresolutions`" array described above.

```
var hd720p =
UC.phone.videoresolutions.videoCaptureResolution1280x720;

UC.phone.setPreferredVideoCaptureResolution(hd720p);
```

Setting the Frame Rate

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

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

API changes

API Release	Description
10.6	Added Chrome-specific screen sharing and Floor control functionality.
	Version information is available through the `UC.csdkVersion` property.

	Additional connection/network callbacks on the UC object:
	`onConnectionRetry`
	`onConnectionReestablished onNetworkUnavailable`

	Added ability to determine inbound video quality. Information about the quality of inbound video is available through the `onConnectionQualityChanged` method of the `Call` object.

	Added ability to set video resolution.
	
	`UC.phone.setPreferredVideoCaptureResolution(...)`
	
	`UC.phone.setPreferredVideoFrameRate(...);`
	
	See Setting Video Resolution on page 27 for more details.

	`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:
	`withAudio`
	`withVideo`

	`setLocalMediaEnabled()` supports two boolean parameters:
	`enableVideo`
	`enableAudio`
	
	`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 aqcuired, `onScreenshareError()` is the callback which is triggered
	After the browser acquires the `localMediaStream` object, the `onlocalMediaStream` `(localMediaStream)` callback is offered to the phone object.

Table 1 Browser API Changes

Creating an iOS application

RE Mobile Client SDK (CSDK) enables you to develop iOS applications offering users the following methods of communication:

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

CSDK provides you with an iOS CSDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

To develop iOS applications using CSDK, XCode 4.5 or later is required.

Information about the minimum version of iOS supported is included in the *Cisco Remote Expert Mobile—Release Notes*.

The iOS CSDK includes 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 Application Event Distribution:
  - ACBClientAED
  - `ACBTopic` and its delegate protocol `ACBTopicDelegate`.

The iOS 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 followed the steps described in Creating a session on page 6.

To set up a project including the RE Mobile Client SDK (CSDK), 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.

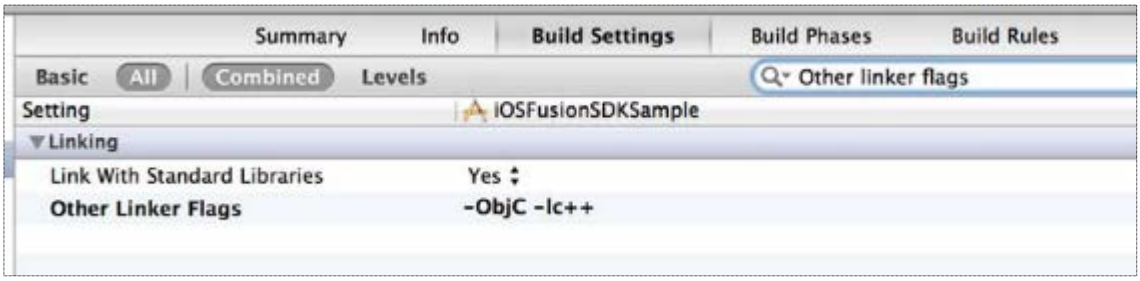
Next, add the RE Mobile Client SDK (CSDK) framework to your project.

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 **ACBClientSDK** 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++
```



4. Position Independent Executables are incompatible with some codecs in the iOS CSDK. 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 from detecting the dependency that the iOS CSDK has on the C++ Standard Library.

Take either of the following actions to work around this issue:

- Add `libstdc++.6.dylib` to the list of required libraries, or
- 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 CSDK.

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/developer.

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

1. Disable the generation of bitcode, using `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.

Threading

All method invocations on the CSDK, even to access read-only properties, must be made from the same thread. This can be any thread, and not necessarily the main thread. Internally, the CSDK can use other threads to increase responsiveness, but any delegate callbacks are made on this same thread that is used to invoke to the CSDK.

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.

The server certificate and CA root certificate can be obtained through the REAS Administration screens. 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/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 (**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 `acceptUntrustedCertificate` method of the `UC` object, although this should only be used during development.

Responding to network issues

As the iOS CSDK is network-based, it is essential that the client application's connection to the network is monitored at all times. CSDK 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 iOS CSDK automatically makes several attempts to re-establish the connection. These attempts are made at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. Each of these attempts is preceded by a `[uc:(ACBUC*)uc willRetryConnectionNumber:(NSUInteger)attemptNumber in:(NSTimeInterval)delay]` call to the `ACBUCDelegate`.

When all reconnection attempts are exhausted, the `ACBUCDelegate` receives the `[ucDidLoseConnection:(ACBUC*)uc]` callback, and the retries stop.

If any of the reconnection attempts are successful, the `ACBUCDelegate` receives the `[ucDidReestablishConnection:(ACBUC*)uc]` callback.

Note: Both the `[uc:(ACBUC*)uc willRetryConnectionNumber:(NSUInteger)attemptNumber in:(NSTimeInterval)delay]` and `[ucDidReestablishConnection:(ACBUC*)uc]` are optional, so the developer may choose to not implement them. The connection retries are attempted regardless.

Note: The retry intervals, and the number of retries attempted by the CSDK may change in future releases. Do not rely on the exact values quoted above.

At this point the client application should assume that the session is now invalid. The client application should end the session and reconnect using the web app to get a new session, as described in Creating a session on page 6.

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 end and recreate the session (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 the other way round, the client application needs to call `ACBUC setNetworkReachable:NO` followed by `ACBUC setNetworkReachable:YES` to disconnect from the first network and reregister on the second.

Network quality callbacks

During a call the application can receive callbacks on the quality of the network.

The quality callback is an optional method on the call delegate:

```
/**
A callback notification to indicate that the current perceived quality of the incoming streams has
changed.
@param call The call.
@param inboundQuality The quality of the stream between 0 and 100, where 100 is best quality.
*/
- (void) call:(ACBClientCall*)call didReportInboundQualityChange:(NSUInteger)inboundQuality;
```

The CSDK starts collecting metrics as soon as the remote media stream is received—when the `callDidAddRemoteMediaStream:` delegate callback is first fired for the call. The metrics are collected every 5s, so the first quality callback fires roughly 5s after this remote media stream callback has fired.

The callback is then be fired 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.

Adding Voice and Video

The API is accessed initially using a single object, `ACBUC`, from which the `ACBClientPhone` object can be retrieved. The phone object is then used 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.

To initialize the `ACBUC` object, the client application needs to log in using the web application, passing the capabilities to which the user has access.

Initializing the ACBUC object and making a call

The following sample shows an example of a string received by the web application:

```
- (void) exampleCode
{
ACBUC* uc = [ACBUC UCWithConfiguration:configuration delegate:aUcDelegate];
[uc startSession]
// uc.phone is available, but wait for udDidStartSession before making a call.
}

- (void) ucDidStartSession:(ACBUC *)uc
{
ACBClientPhone* phone = uc.phone;
phone.delegate = aPhoneDelegate;
phone.previewView = previewView;

ACBClientCall* call = [phone createCallToAddress:calleeAddress audio:YES video:YES
delegate:aCallDelegate];
call.videoView = aVideoView;
}
```

The `ACBUCDelegate` is invoked by the `ucDidStartSession` to indicate whether registration with the server succeeded or failed.

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 saved by iOS until your application is uninstalled or the permissions are reset in the iOS Settings.

Note: The end user can also change the microphone and camera permission for your application in iOS Settings.

The iOS CSDK provides a helper method to request access to the microphone and camera—see the `ACBClientPhone requestMicrophoneAndCameraPermission`. This method delegates to the iOS permission APIs. This method should typically be called before making or receiving calls. An individual alert is displayed for each requested permission. The alert is displayed the first time you call this method. 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 CSDK 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 optional `ACBClientCallDelegate didReceiveCallRecordingPermissionFailure` method is called, and the call ends.

Video views and preview views

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 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 performed at any time. If there are calls in progress when the properties are set, the changes take effect when the next video call is made.

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

Handling device rotation

By default, when making video calls, the device sends a portrait video stream—the client assumes that the iOS device is held in portrait mode. This can be changed by calling the `setVideoOrientation` method of `ACBClientPhone`.

```
- (void) deviceRotatedToLandscape
{
[self.phone setVideoOrientation:UIInterfaceOrientationLandscapeLeft];
}
```

The following parameters can be passed to this method:

- `UIInterfaceOrientationPortrait`
- `UIInterfaceOrientationPortraitUpsideDown`
- `UIInterfaceOrientationLandscapeLeft`
- `UIInterfaceOrientationLandscapeRight`

The application can be set to change this value as the device rotates by registering for the `UIApplicationWillChangeStatusBarOrientationNotification` notification and calling `setVideoOrientation` every time a notification is received, for example:

```
- (void) viewDidLoad
{
[super viewDidLoad];
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(willRotate:)
name:UIApplicationWillChangeStatusBarOrientationNotification object:nil];
}

- (void) viewDidUnload
{
[ super viewDidUnload];
[[NSNotificationCenter defaultCenter] removeObserver:self
name:UIApplicationWillChangeStatusBarOrientationNotification object:nil];
}

- (void) willRotate:(NSNotification*)notification
{
NSInteger value = [[notification.userInfo
objectForKey:UIApplicationStatusBarOrientationUserInfoKey] integerValue];
[self.phone setVideoOrientation:(UIInterfaceOrientation)value];
}
```

- The orientation persists between calls; for example, if the orientation is set to `UIInterfaceOrientationPortraitUpsideDown` 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 takes effect when a video call is next in progress.
- The `setVideoOrientation` method only affects the orientation of the stream being sent. If a preview window is being used, it also affects the orientation of 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 camera

During video calls, the front camera is used by default. This can be changed by calling the `setCamera` method of `ACBClientPhone`.

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

The following parameters can be passed to this method:

- AVCapturePositionBack
- AVCapturePositionFront

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

The camera setting persists between calls, so 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.

Receiving a call

The `ACBClientPhoneDelegate` `didReceiveCall:` delegate method is invoked when an incoming call is received. To answer the incoming call, its `answerWithAudio:video:` method should be called:

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

You can change the boolean values to answer the call as audio-only or video-only.

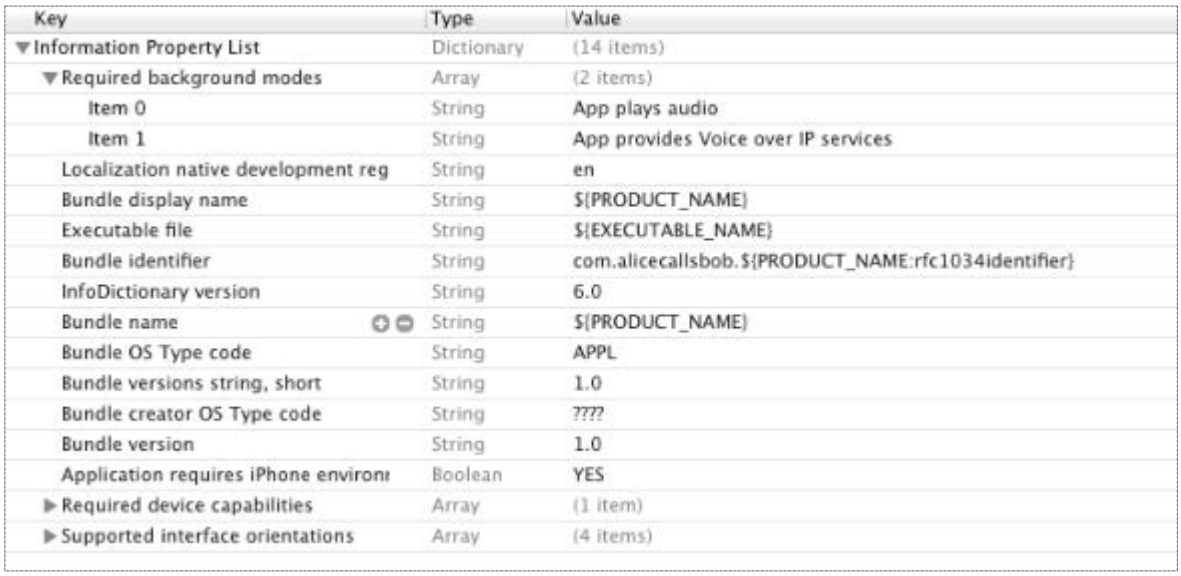
Note:

- 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 `ACBClientPhoneDelegate didChangeStatus`) before calling `AVAudioSession sharedInstance` methods.

Receiving calls when the client is in background or suspended modes

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



Monitoring the state of a call

The call transitions through several states, and these can be monitored by assigning a delegate to the call:

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
call.delegate = self;
[call answerWithAudio:YES video:YES];
}
```

Note: For a full list of possible responses, see https://en.wikipedia.org/wiki/List_of_SIP_response_codes or https://www.ietf.org/rfc/rfc3261.txt section 21, *Response Codes*.

Each state change fires the `call:didChangeStatus:` delegate method.

```
@optional
/** A callback notification indicates the call has changed
state. */
- (void) call:(ACBClientCall*)call
didChangeStatus:(ACBClientCallStatus)status;
/** A callback notification indicates the remote party
display name has changed. */
- (void) call:(ACBClientCall*)call
didChangeRemoteDisplayName:(NSString*)name;
/** A callback notification indicating that the local media
stream was added. */
- (void) callDidAddLocalMediaStream:(ACBClientCall *)call;
/** A callback notification indicating that the remote media
stream was added. */
- (void) callDidAddRemoteMediaStream:(ACBClientCall *)call;
```

As the outgoing call progresses towards being fully established, the application receives a number of calls to `didChangeStatus`, each time being passed one of the `ACBClientCallStatus` enumeration. Switching on the value of the enum provides your application with the current context of the call, which allows you to properly drive the user experience, for example by playing a local audio file for ringing or alerting.

Handling multiple calls

Applications developed with RE Mobile Client SDK (CSDK) support multiple simultaneous calls:

- To make additional calls while another call is in progress, the client application uses the ACBClientCall* call method (see Initializing the ACBUC object and making a call on page 22).

  Once a call is established, an application can send DTMF tones on that call. To do this, call the following method:

  ```
  - (void) playDTMFCode:(NSString*)code localPlayback:(BOOL)localPlayback;
  ```

  This call has two parameters:

- The first can either be a single tone, (for example, 6), or a sequence of tones (for example, #123,*456)

  Note: A comma indicates a two second pause between the `3` and the `*` tone.

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

  During a call it is possible to mute or unmute one, or both, of the local audio and video streams.

- To receive incoming calls while another call is in progress, the didReceiveCall: method should be triggered (see Receiving a call on page 24).

Muting the local audio and video streams

Muting the stream stops that stream being sent by the user 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) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
self.call = call;
[call answerWithAudio:YES video:YES];
}
- (void) muteButtonPressed:(UIButton*)button
{
[self.call enableLocalAudio:NO];
[self.call enableLocalVideo:NO];
}
```

Holding and resuming a call

During a call it is possible to 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) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
{
[call answerWithAudio:YES video:YES]
}
- (void) holdButtonPressed:(UIButton*)button
{
[call hold];
}
- (void) resumeButtonPressed:(UIButton*)button
{
[call resume];
}
```

Setting Video Resolution

The RE Mobile Client SDK (CSDK) supports configuring the captured, and hence sent, video resolution for video calls. One of a set of video resolutions can be selected and applied to the capture device. Additionally the frame rate for capture can be configured. When a resolution and frame rate are specified, the CSDK makes every effort to match those values where hardware allows.

Enumerating the Possible Resolutions

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

Device Type	Maximum Resolution	MaximumFrame Rate
iPhone 4 and below iPad 1	No video support	
iPhone 4s iPad 2 iPad (3rd generation) iPad mini	352 x 288	20
iPhone 5 iPhone 5c iPhone 5s iPhone 6 iPhone 6s iPhone 6 Plus iPhone 6s Plus iPad (4th generation)/iPad Retina iPad mini 2/iPad mini Retina iPad mini 3 iPad mini 4	640 x 480	30
iPad Air iPad Air 2 iPad Pro	1280 x 720	30

Note: The values shown here are determined by the processing capability of the device to perform the encoding and decoding required, as well as the screen resolution.

Setting the Resolution

The captured video resolution can be set using the `preferredCaptureResolution` property of the phone object. The value supplied can be obtained using `recommendedCaptureSettings`, as described above, for example:

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

Alternatively the value can be set directly from the enumeration, for example:

```
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 captured video frame rate can be set using the `preferredCaptureFrameRate` property of the phone object;

```
uc.phone.preferredCaptureFrameRate = chosenSetting.frameRate;
```

or alternatively, a custom frame rate can be attempted:

```
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.

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. The video streaming is automatically resumed 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 to decide whether their application should mute audio and video when transitioning to background mode.

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

Note: The behaviour of iOS is different from Android.

Adding Application Event Distribution

The API is accessed initially using a single object, `ACBUC`, from which other objects can be retrieved. `ACBUC` has an attribute named `aed`, which is the starting point for all AED operations.

To create an AED application, you need to do the following:

1. Create an instance of `ACBTopicDelegate` and implement the callback methods.
2. Access the `aed` attribute to create and 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 notifications.

Assigning the delegate

It is essential that you do this before you can create a topic.

1. Make the class that implements the callback methods conform to the `ACBTopicDelegate` protocol:

```
@interface AEDViewController : UIViewController<UCConsumer, ACBTopicDelegate, UITableViewDelegate,
UITableViewDataSource>
```

2. Assign the delegate inside the implementation:

```
currentTopic.delegate = self
```

Creating, and connecting to, a topic

You can create a topic using the `createTopicWithName: delegate:` method on the AED object.

```
currentTopic = [uc.aed createTopicWithName:_topicName.text delegate:aTopicDelegate];
```

Important: This creates a client-side representation of a topic and automatically connects to it.

You know that this has succeeded if the following callback is fired:

```
- (void)topic:(ACBTopic *)topic didConnectWithData:(NSDictionary *)data
```

You can deal with the `NSDictionary` data object by writing something such as the following inside the `didConnectWithData` callback implementation:

```
//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"];
[self logMessage:[NSString stringWithFormat:@"Key:'%@' Value:'%@'",keyField,valueField]];
}
}
```

Calling methods on the topic

Once you have connected to a topic, the client can run the following methods on the topic and you can expect to get either success or error callbacks in response:

Publishing data

```
- (IBAction)publishData:(id)sender {
[currentTopic submitDataWithKey:dataKey.text value:dataValue.text];
}
```

Deleting data

```
- (IBAction)deleteData:(id)sender {
[currentTopic deleteDataWithKey:dataKey.text];
}
```

Sending a message

```
- (IBAction)sendMessage:(id)sender {
[currentTopic sendAedMessage:message.text];
}
```

You only get callbacks if the topic is connected. You can check whether the topic is connected by checking the `connected` property of the `ACBTopic`. You should be aware that you may not be notified using a callback if a topic is disconnected, as there is no disconnected callback.

Disconnecting from the topic

You can either disconnect from the topic without destroying it, or delete the whole topic, which disconnects any other subscribers.

```
- (IBAction)disconnectTopic:(id)sender {
if(currentTopic.connected)
{
if( [deleteTopic.text isEqualToString:tickedBoxStr])
{
[currentTopic disconnectWithDeleteFlag:TRUE];
}
else
{
[currentTopic disconnectWithDeleteFlag:FALSE];
}
NSString *msg = [NSString stringWithFormat:@"Topic '%@' disconnected.", currentTopic.name];
[self logMessage:msg];
}
else
{
NSString *msg = [NSString stringWithFormat:@"Topic '%@' already disconnected.", currentTopic.name];
[self logMessage:msg];
}
}
```

If the topic deletes successfully, you are notified by the following callbacks:

- When you delete the topic by calling `disconnectWithDeleteFlag:TRUE` the following callback is called:

  ```
  - (void)topic:(ACBTopic *)topic didDeleteWithMessage:(NSString *)message;
  ```

- When you delete the topic, or another subscriber to the topic deletes it:

  ```
  - (void)topicDidDelete:(ACBTopic *)topic;
  ```

API changes

API Release	Description
11.5	Added `UC.checkBrowserCompatibility(pluginInfoCallback)`
10.6	Sample modified to build and run on Xcode 7 and iOS 9.
	Added `ACBDevice` class to provide access to recommended resolutions without requiring an `ACBClientPhone` instance.
	Support for arm64. Extra libraries are now required at runtime – see iOS Self-Signed Certificates on page 20.
	Added two new optional callbacks—`didReceiveDialFailureWithError` and `didReceiveCallFailureWithError` – to the `ACBClientCallDelegate` that pass a `NSError` object (containing error code, reason etc.) to the application. These supersede the existing `didReceiveDialFailure` and `didReceiveCallFailure` callback, 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 20.
	Extra callbacks during network loss. See Reacting to network loss on page 21: `(ACBUC*)uc willRetryConnectionNumber:(NSUInteger)attemptNumber in:(NSTimeInterval)delay]`
	Added ability to determine inbound video quality. Information about the quality of inbound video is available through the Call delegate: - (void) call:(ACBClientCall*)call didReportInboundQualityChange:(NSUInteger)inboundQuality;

	ACBClientCall:
	`(void) hold;`
	`(void) resume;`
	
	The `hold` and `resume` methods on the `call` object only attempt to hold or resume the call once the call has been established, that is: `ACBClientCallStatusInCall`.

	Added ability to set the video capture resolution. The list of possible resolutions is available from the ACBClientPhone object using the "`recommendedCaptureSettings`" method: NSArray* recommendedSettings = [uc.phone recommendedCaptureSettings]; Set the resolution using `uc.phone.preferredCaptureResolution` Set the frame rate using `uc.phone.preferredCaptureFrameRate`

API Release	Description
	Added ability to set the video capture resolution. The list of possible resolutions is available from the ACBClientPhone object using the "`recommendedCaptureSettings`" method: `NSArray* recommendedSettings = [uc.phone recommendedCaptureSettings];` Set the resolution using `uc.phone.preferredCaptureResolution` Set the frame rate using `uc.phone.preferredCaptureFrameRate`
	The API now offers hold and resume methods on the call object.

	ACBClientAED
	`(ACBTopic *)createTopic:(NSString *)topicName`
	
	Now replaced with two methods:
	`(ACBTopic *)createTopicWithName:(NSString *)topicName delegate:(id<ACBTopicDelegate>)delegate;`
	
	`(ACBTopic *)createTopicWithName:(NSString *)topicName expiryTime:(int)expiryTime delegate:(id<ACBTopicDelegate>)delegate;`

	ACBClientConversation:
	`@property (readonly) NSString* contact;`
	
	Now replaced with:
	`@property (readonly) NSString* contactAddress;`

	ACBClientContact:
	`@property (readonly) NSString* contactId;`
	
	Now replaced with:
	`@property (readonly) NSString* name`
	`@property (readonly) NSString* address`

	ACBTopic:
	`(void)topic:(ACBTopic *)topic didNotSendMessageWithError:(NSString *)messageError errorMessage:(NSString *)errorMessage;`
	
	Now replaced with:
	`(void)topic:(ACBTopic *)topic didNotSendMessage:(NSString *)originalMessage message:(NSString *)message;`

	ACBTopic:
	`(void) connectWithTimeout:(int)timeout;`
	`(void) connect;`
	
	Both methods have been removed, the functionality is now within ACBClientAED.

Table 2 iOS API Changes

Creating an Android application

RE Mobile Client SDK (CSDK) enables you to develop applications for Android devices which offer users the following methods of communication:

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

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

To develop Android applications using CSDK, your system needs to conform to the system requirements listed at the following URL:

http://developer.android.com/sdk/index.html

Information about the minimum version of Android supported is in *Cisco Remote Expert Mobile—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.

Setting up a project

Note: Before following this process, make sure that you have followed the steps described in Creating a session on page 6.

1. Create an Android Application project using either Android Studio or the Eclipse Android Developer Tools (ADT) plugin.
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:

- libjingle_peerconnection.jar
- autobahn-0.5.0.jar
- jackson-core-asl-1.9.7.jar
- jackson-mapper-asl-1.9.7.jar

3. Add the following to your `libs` folder. 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.

- android-support-v4.jar
- libjingle_peerconnection_so.so

4. In order to allow your project to access the required features on Android devices, include the following in your `AndroidManifest.xml` file:

- android.permission.INTERNET
- android.permission.RECORD_AUDIO
- android.permission.CAMERA
- android.permission.MODIFY_AUDIO_SETTINGS

Self-Signed Certificates

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

- Make use of the `setTrustManager` and `setHostNameVerifier` methods on the `UC` object to perform your own 'validation' of the SSL connection, or
- Add the server certificate and the associated CA root certificate to the Credential Storage on your client.

The server certificate and CA root certificate can be obtained through the REAS Administration screens. 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, 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 connect to the server.

Responding to network issues

Reacting to network loss

In the event of network connection problems, the Android CSDK automatically makes several attempts to re-establish the connection. These attempts are made at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. Each of these attempts is preceded by a `onConnectionRetry(int attemptNumber, long delayUntilNextAttempt` call to the `UCListener.`

When all reconnection attempts are exhausted, the UCListener receives the `onConnectionLost()` callback, and the retries stop.

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 CSDK are subject to change in future releases. Do not rely on the exact values quoted above.

Reacting to network changes

The network loss described above could have been caused by network connectivity to the device being temporarily lost, for example when moving between two Wi-Fi networks, or from a Wi-Fi network to a cellular data connection. In this scenario, it is not desirable to end and create the session again, 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 reachability is available, call `UC.setNetworkReachable(true)`. Until this call is made, no session connection to the server is attempted.

Once a session connection is established, if network reachability drops, 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.

The quality callback is a new method on the Call listener:

```
/**
* A callback notification indicating that the perceived quality of the
* incoming audio and video streams has changed.
*
* @param call The call that changed.
* @param inboundQuality The quality of the stream(s) between 0 and
* 100, where 100 is best quality.
*/
void onInboundQualityChanged(Call call, int inboundQuality);
```

The CSDK starts collecting metrics as soon as the remote media stream is received. The metrics are collected every 5s.

The callback is then fired 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.

Adding Voice and Video

The API is accessed initially using a single object, `UCFactory`, from which a `UC` object can be used to obtain `Phone` and `AED` instances.

The `Phone` object is then used 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 `AED` object can be used to create AED Topics.

Initializing the Phone object and making a Call

```
public class Main extends Activity implements UCListener, PhoneListener, CallListener
{
protected void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);

mUC = UCFactory.createUc(this, //context
"session-token", //session token obtained from
//Session Token REST API.
This); //listener

mUC.setNetworkReachable(true);
mUC.startSession();

mPhone = mUC.getPhone();
mPhone.addListener(this);
mPhone.createCall(“calleeAddress”,
true, true, // audio and video enabled
this); // CallListener
}

public void onConnectionLost() { Log.w(TAG, "onConnectionLost"); }
public void onSessionNotStarted() {}
public void onSessionStarted() { mUCInitialized = true; }
public void onSystemFailure() { Log.w(TAG, "onSystemFailure"); }

public void onCallFailed(Call call, String message, CallStatus callStatus) { Log.w(TAG, "onCallFailed: "
+ message); }
public void onDialFailed(Call call, String message, CallStatus callStatus) {Log.w(TAG, "onDialFailed: " +
message); }
public void onLocalMediaStream(Call call) { Log.d(TAG, "onLocalMediaStream"); }
```

```
public void onMediaChangeRequested(Call call, boolean hasAudio, boolean hasVideo) { Log.d(TAG,
"onMediaChangeRequested: audio(" + hasAudio + ") video (" + hasVideo + ")"); }
public void onRemoteDisplayNameChanged(Call call, String name) { Log.i(TAG, "onRemoteDisplayNameChanged:
" + name); }
public void onRemoteMediaStream(Call call) { Log.d(TAG, "onRemoteMediaStream"); }
public void onStatusChanged(Call call, CallStatus status) { }
}
```

Receiving a call

The `PhoneListener onIncomingCall` method is invoked when an incoming call is received. To answer the incoming call, its `call.answer(boolean, boolean)` method should be called:

```
public void onIncomingCall(Call call)
{
call.addListener(this);
call.answer(true, true); //Answer with audio & video.
}
```

Change the boolean values to answer the call as audio only or video only.

Note: 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, if the local application answers as video only, then neither party will send or receive audio.

Monitoring the state of a call

The call transitions through a number of states during its lifetime, and these can be monitored by setting the `CallListener` and implementing the `onStatusChanged` method.

```
public void onStatusChanged(Call call, CallStatus status)
{
//status could be ALERTING, BUSY, ENDED etc.
}
```

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

Video views and preview views

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

Initialising the video surface and preview surface is optional and can be done at any time. A video surface is created using the Phone object. Typically, you would create a video surface for the preview view and remote video view, and then set them using the `Phone` object or a `Call` object. If there are calls in progress when the properties are set, the changes take effect immediately and endure for future calls. If there are no active video calls, the value takes 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 renders a full frame of green. Only when streaming does video appear.

The camera to use as the local video source is set on the `Phone` object. See Switching between the front and back camera on page 39.

Handling multiple calls

Applications developed with RE Mobile Client SDK (CSDK) support multiple simultaneous calls:

- To make additional calls while another call is in progress, the client application would use the Phone `createCall` method (see Initializing the Phone object and making a Call on page 36).
- To receive incoming calls while another call is in progress, the `onIncomingCall` method of the `PhoneListener` should be triggered (see Receiving a call on page 37).
- The muting of video and audio applies to all calls, but individual calls can be put on hold and terminated.
- The sample application implements multiple call functionality for tablets only.

Sending DTMF tones

Once a call is established, an application can send DTMF tones on that call. To do this, call the `Call playDTMFCode` method. The first parameter to this call can either be a single tone, (for example, `6`), or a sequence of tones (for example, `#123*456`).

Muting the local audio and video streams

During a call, it is possible to 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. This affects all calls.

When a call starts, the streams are muted as in the current Phone setting.

Muting or unmuting a video stream in an audio-only call has no effect.

Handling device rotation

By default, when making video calls, the device sends a portrait video stream (that is, it is assumed that the Android device is held in portrait mode). This can be changed by calling the `setVideoOrientation` method of the `Phone` object.

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

- Surface.ROTATION_0
- Surface.ROTATION_90
- Surface.ROTATION_180
- Surface.ROTATION_270

The orientation persists between calls, for example if the orientation is set to `Surface.ROTATION_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 only affects the orientation of the stream being sent. If a preview window is being used, it also affects 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 camera

By default, when making video calls, the front-facing camera is used. This can be changed by calling the `setCamera` method on the `Phone` object and passing the `cameraId` that you wish to use—for selecting the `cameraId` to use, see the Android API Guide at the following URL:

http://developer.android.com/reference/android/hardware/Camera.html#open(int)

The camera setting persists between calls, that is, if the rear-facing 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.

The RE Mobile Client SDK (CSDK) Android sample app checks to see how many cameras there are on the device. If there is only one camera, it will use it, whether it is front-facing or back-facing. If there is more than one camera, it will use the first front-facing camera it can find. If there is more than one 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.

Holding and resuming a call

During a call it is possible to 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()
}
```

Setting Video Resolution

The RE Mobile Client SDK (CSDK) supports configuring the captured, and hence sent, video resolution for video calls. One of a set of video resolutions can be selected and applied to the capture device. Additionally the frame rate for capture can be configured. When a resolution and frame rate are specified, the CSDK makes every effort to match those values where hardware allows.

Enumerating the Possible Resolutions

The list of possible resolutions is available from the Phone object using the `getRecommendedCaptureSettings()` method:

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

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

The supported resolutions are:

Enumeration Value	Width	Height
`RESOLUTION_174x144`	174	144
`RESOLUTION_352x288`	352	288
`RESOLUTION_640x480`	640	480
`RESOLUTION_960x720`	960	720
`RESOLUTION_1280x720`	1280	720

Note: The behaviour on Android is different from iOS. On iOS the SDK does not allow you to set the resolution to one that

is not supported by the device. Due to the vast number of Android devices, it is not possible to know what devices can support a given resolution and so the Android SDK allows the application to choose any resolution.

Setting the Resolution

The captured video resolution can be set using the `setPreferredCaptureResolution(PhoneVideoCaptureResolution resolution)` method of the phone object.

```
phone.setPreferredCaptureResolution(PhoneVideoCaptureResolution.RESOLUTION_640x480);
```

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

Setting the Frame Rate

The captured video frame rate can be set using the `setPreferredCaptureFrameRate(int frameRate)` method of the phone object.

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

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

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 continues 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 behaviour of Android is different from that of iOS.

Adding Application Event Distribution

Also accessible from the `UC` object is the `AED` object, which is the starting point for all Application Event Distribution (AED) operations.

To create an AED application:

- Access the `AED` object to create a topic, add a `TopicListener` and connect to a `Topic`.
- Call any other methods on the `Topic` object (apart from `disconnect`) in order to change data on the topic
- Disconnect from the topic when you no longer want to receive notifications.

Creating and connecting to a topic

A topic can be created using the `createTopic` method on the `AED` object. This creates a client-side representation of a topic and automatically connects to it.

```
uc.getAED().createTopic("my topic", this);
.
.
.
//TopicListener method implementations.
public void onTopicConnected(Topic topic, java.util.Map<java.lang.String,java.lang.Object> data) {}
```

You know that the connect has succeeded if the `onTopicConnected` listener is fired. You can then add data to the topic. The `onTopicSubmitted` method is called locally if data is submitted successfully, and remote clients can register their own `TopicListener` and receive data and message updates through the `onTopicUpdatedRemotely` method. In the event of an error, `onTopicNotSubmitted` is called.

```
//TopicListener method implementations.
public void onTopicConnected(Topic topic, java.util.Map<java.lang.String,java.lang.Object> data)
{
topic.submitData("foo", "bar");
topic.sendAedMessage("Hello World!");
}
```

Similarly, the `onTopicSent`, `onMessageReceived` and `onTopicNotSent` methods is called to indicate success or failure when sending AED messages.

Disconnecting from the topic

You can either disconnect from the topic without destroying it, or you can choose to delete the topic. If the topic deletes successfully, you are notified using the `TopicListener` `onTopicDeleted` callback.

API changes

API Release	Description
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 WebSockets connection between the client and Web 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 WebSockets connection between the client and Web 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. The `VideoSurfaceListener` `onFrameSizeChanged` method has been changed to include an additional `VideoSurface` parameters because there may be multiple remote video surfaces that need to be differentiated. 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 WebSockets connection to the CSDK.
	VideoSurface: `VideoSurface create(Context, Point, VideoSurfaceListener);` A `VideoSurface` is now created using the `createVideoSurface` method of the `Phone` interface.

	VideoSurface;
	`abstract modifier`
	`onPause/onResume`
	`GLSurfaceView.Renderer` methods
	
	`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.
	The `onPause`/`onResume` and `GLSurfaceView.Renderer` methods have been removed to simplify the API and avoid confusion.

	Call:
	`setPreviewSurface(VideoSurface);`
	`setVideoSurface(VideoSurface);`
	
	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. The list of possible resolutions is available from the Phone object using `phone.getRecommendedCaptureSettings;` Set the resolution using `phone.setPreferredCaptureResolution` Set the frame rate using `phone.setPreferredCaptureFrameRate`
	The API now offers `hold` and `resume` methods on the `Call` object.

Table 3 Android API Changes

Creating an OSX application

RE Mobile Client SDK (CSDK) enables you to develop OSX applications offering users the following method of communication:

- Voice calling

RE Mobile Client SDK (CSDK) provides you with an OSX SDK and a network infrastructure that integrate seamlessly with your existing SIP infrastructure.

To develop OSX applications using RE Mobile Client SDK (CSDK), XCode 4.5 or later is required.

The OSX CSDK includes 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 that you have followed the steps described in Creating a session on page 6.

To set up a project including the RE Mobile Client SDK (CSDK), 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, the file explorer displays.
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 RE Mobile Client SDK (CSDK) 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 ACBClientSDK folder, select it and click **OK**.

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. Internally, the SDK can use other threads to increase responsiveness, but any delegate callbacks are made on this same thread that is used to invoke to 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.

The server certificate and CA root certificate can be obtained through the REAS Administration screens. 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, 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 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 `acceptUntrustedCertificate` method of the `UC` object, although this should only be used during development.

Responding to network issues

As the OSX RE Mobile Client SDK (CSDK) is network-based, it is essential that the client application's connection to the network is monitored at all times. CSDK 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 OSX RE Mobile Client SDK (CSDK) automatically makes several attempts to re-establish the connection. These attempts are made at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. Each of these attempts is preceded by a `[uc:(ACBUC*)uc willRetryConnectionNumber:(NSUInteger)attemptNumber in:(NSTimeInterval)delay]` call to the `ACBUCDelegate`.

When all reconnection attempts are exhausted, the `ACBUCDelegate` receives the `[ucDidLoseConnection:(ACBUC*)uc]` callback, and the retries stop.

If any of the reconnection attempts are successful, the `ACBUCDelegate` receives the `[ucDidReestablishConnection:(ACBUC*)uc]` callback.

Note:

- Both the [uc:(ACBUC*)uc willRetryConnectionNumber:(NSUInteger)attemptNumber in:(NSTimeInterval)delay] and [ucDidReestablishConnection:(ACBUC*)uc] are optional, so the application may choose to not implement them. The connection retries are attempted regardless.
- 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.

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 using the web app to get a new session, as described in in Creating a session on page 6.

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, 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 OSX to receive notification of changes in network reachability. When OSX 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 will 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 one WiFi network to another, the client application needs to call `ACBUC setNetworkReachable:NO` followed by `ACBUC setNetworkReachable:YES` to disconnect from the first network and reregister on the second.

Adding Voice

The API is accessed initially using a single object, `ACBUC`, from which the `ACBClientPhone` object can be retrieved. The phone object is then used 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.

To initialize the `ACBUC` object, the client application needs to log in using the web application, passing the capabilities to which the user has access.

Initializing the ACBUC object and making a call

The following sample shows an example of a string received by the web application:

```
- (void) exampleCode
{
ACBUC* uc = [ACBUC UCWithConfiguration:configuration delegate:aUcDelegate];
[uc startSession]
}

- (void) ucDidStartSession:(ACBUC *)uc
{
ACBClientPhone* phone = uc.phone;
phone.delegate = aPhoneDelegate;
ACBClientCall* call = [phone createCallToAddress:calleeAddress audio:YES video:NO
delegate:aCallDelegate];
}
```

The `ACBUCDelegate` is invoked by the `ucDidStartSession` to indicate whether registration with the server succeeded or failed.

Video views and preview views

Video is not supported in this release of the OSX RE Mobile Client SDK (CSDK).

Receiving a call

The `ACBClientPhoneDelegate incomingCallReceived:` delegate method is invoked when an incoming call is received. To answer the incoming call, its `answerWithAudio:video:` method should be called:

```
- (void) incomingCallReceived:(ACBClientCall*)call
{
[call answerWithAudio:YES video:NO]
}
```

You can change the boolean values to answer the call as audio-only or video-only.

Note:

- 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 will send or receive video.
- Video is not supported in this release of the OSX RE Mobile Client SDK (CSDK).

Monitoring the state of a call

The call transitions through several states, and these can be monitored by assigning a delegate to the call:

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

Each state change fires the `call:didChangeStatus:` delegate method.

```
@optional
/** A callback notification indicates the call has changed state. */
- (void) call:(ACBClientCall*)call
didChangeStatus:(ACBClientCallStatus)status;
/** A callback notification indicates the remote party display name has changed. */
- (void) call:(ACBClientCall*)call
didChangeRemoteDisplayName:(NSString*)name;
/** A callback notification indicating that the local media stream was added. */
- (void) callDidAddLocalMediaStream:(ACBClientCall *)call;
/** A callback notification indicating that the remote media stream was added. */
- (void) callDidAddRemoteMediaStream:(ACBClientCall *)call;
```

Handling multiple calls

Applications developed with RE Mobile Client SDK (CSDK) support multiple simultaneous calls:

- To make additional calls while another call is in progress, the client application uses the `ACBClientCall* call` method (see Initializing the ACBUC object and making a call on page 45).
- To receive incoming calls while another call is in progress, the `incomingCallReceived:` method should be triggered (see Receiving a call on page 45).

Sending DTMF tones

Once a call is established, an application can send DTMF tones on that call. To do this, call the following method:

```
- (void) playDTMFCode:(NSString*)code localPlayback:(BOOL)localPlayback;
```

This call has the following parameters:

- The first can either be a single tone, (for example, `6`), or a sequence of tones (for example, `#123,*456`)

  Note: A comma indicates that there should be a two second pause between the `3` and the `*` tone.
- The second is a boolean which indicates whether the tone is played back locally.

Muting the local audio stream

During a call it is possible to mute or unmute the local audio stream.

Muting the stream stops that stream being sent by the user to the remote party, however the user still receives any stream that the remote party sends.

To mute the audio stream use one 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 it is possible to 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) incomingCallReceived:(ACBClientCall*)call
{
[call answerWithAudio:YES video:NO]
}
- (void) holdButtonPressed:(UIButton*)button
{
[call hold];
}
- (void) resumeButtonPressed:(UIButton*)button
{
[call resume];
}
```

API changes

API Release	Description
2.1.11	Support for self-signed certificates—see Self-Signed Certificates on page 44.
2.1.3	Introduces OSX CSDK with support for audio-only calls.

Table 4 OSX API Changes

Creating a Windows Application

RE Mobile Client SDK (CSDK) enables you to develop Windows applications that offer users the ability to make voice and video calls. CSDK provides you with a Windows SDK and a network infrastructure that integrate seamlessly with your existing SIP infrastructure.

The Windows CSDK includes the following main classes:

- The top-level `UC` class and its corresponding `UCListener` interface.
- `Phone` and `PhoneListener` for creating and receiving calls.
- `Call` and `CallListener` for working with calls.

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 followed the steps described in Creating a session on page 6.

This section provides guidelines on setting up a RE Mobile Client SDK (CSDK) using Visual Studio 2013. You need to create a new project and add to it the SDK 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 click **File > New Project**,
3. Select your preferred project template and name, 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. From the **Configuration** combo box, select **All Configurations**,
6. In the **C/C++ > General** section, add the CSDK "`include`" directory (from step 1) to the **Additional Include Directories**,
7. In the **Linker > Input** section, add the "`lib\ClientSDKWin.lib`" library (from step 1) to the **Additional Dependencies**.

Adding Voice And Video

The API is accessed initially using the `UC` object, from which the `Phone` object can be retrieved. The `Phone` object can then 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.

Before initialising the UC object, your application needs to obtain a "session ID" string from the Web Gateway.
See Creating a session on page 6.

Initializing the UC, and starting the session

The following sample illustrates the creation of the UC, using a session ID string:

```
uc = std::make_unique<UC>(sessionID, this);
uc->StartSession();
```

The first line in this sample creates the `UC` object using the session ID string obtained from the Web 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 UC listener.

The second line 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.

The UC now attempts to start the session, and the registered UCListener object will shortly receive asynchronous notification that the session has started or failed, for example:

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

The failure notification is OnSessionNotStarted() .

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", true, true, this);
```

In this example, "1234" is the number we are dialling, the two boolean 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: A Call object is immediately returned, but it is the CallListener that is asynchronously notified as to whether the call succeeds or fails.

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: the VideoView and the 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 needs to provide a concrete implementation of in order to receive image data and pass it on to the VideoView.

The following code illustrates the creation of a VideoView and ImagePipeServer subclass to display video received from the remote peer:

```
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; it should be implemented by the application to invalidate the user interface element that displays the video. Finally we create 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`, for example:

```
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 is redrawn, you can simply pass the `HDC` to the `VideoView` to paint the new video frame:

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

Receiving a call

The `PhoneListener` object that you registered with the `Phone` object is invoked when an incoming call is received:

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

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

The two boolean parameters to the `Answer()` call indicate whether or not we want to answer the call with audio and video respectively.

Monitoring the state of a call

Each call transitions through several states, and these transitions can be monitored by the `CallListener` as follows:

```
void MyCallListener::OnStatusChanged(CallStatus newStatus)
{
//react to the state change
}
```

`CallStatus` is an enum class that enumerates all of the possible states, including `ALERTING`, `IN_CALL`, `BUSY`, `ENDED` and more. These states can be used to enhance the user experience in your application, for example, by playing audio when `RINGING` or `ALERTING`.

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

Sending DTMF tones

Your application can send DTMF tones on a call as follows:

```
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, that is, to be audible to the user of your application.

A comma character can be used to insert a two-second pause into a sequence of tones.

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:

```
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:

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

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();
}
```

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.

The video resolution is set on the Phone object:

```
phone->SetPreferredCaptureResolution(
VideoCaptureResolution::RESOLUTION_1280x720);
```

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

The frame rate can also be set as follows:

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

Note: Changes to the video resolution and frame rate apply only to new calls—existing calls are not affected.

Adding Application Event Distribution

Application Event Distribution is contained in the `AED` and `Topic` objects. There is also a TopicListener object which

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.

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 the topic is created with an expiry time, that topic is automatically removed from the server after the topic 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 callback methods through which the application can be informed about things which happen on the topic.

If the topic identified by the name parameter 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 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 is always a 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 receives 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 is also 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`) include a `version` parameter greater than 0.

OnTopicUpdated

The `OnTopicUpdated` callback is received 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` is true if the data item has been removed from the topic. 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 receives 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 the message is successfully sent, the listener receives an `OnTopicSent` 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 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, causes the topic to be deleted from the server (the default value is `false`). The listener receives 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 the topic when it is deleted from the server as a result of any client calling `Disconnect(true)`. 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

The network connection is essential to the functionality of the Windows SDK, and therefore the SDK monitors the status of the connection to the Web Gateway and notify the application of any issues.

If the SDK detects a failure in the Web Gateway connection, it automatically makes several attempts to re-establish the connection. These attempts are made at decreasing intervals (0.5s, 1s, 2s, 4s, 4s, 4s, 4s) and each is preceded by the following invocation on the `UCListener` object:

```
OnConnectionRetry(uint8_t attemptNumber, uint16_t delayInSeconds);
```

As soon as the connection is re-established, the `UCListener` receives the following notification:

```
OnConnectionReestablished();
```

If all of the reconnection attempts fail then the `UCListener` receives the following notification. At this point, the session cannot be recovered and a new session will be required for further calls.

```
OnConnectionLost();
```

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.

Creating a Windows .NET Application

RE Mobile Client SDK (CSDK) 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: The application you create must be capable of creating the session as described in in Creating a session on page 6.

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. In 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 **Add**.

Adding Voice And Audio

The API is accessed initially using the `CLI_UC` object, from which the `CLI_Phone` object can be retrieved. The `CLI_Phone` object can then be used 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.

Before initialising the `CLI_UC` object, your application needs to obtain a "session ID" string from the Web Gateway, as described in Chapter 1.

Initializing the CLI_UC and starting the session

The following sample illustrates the creation of the CLI_UC using a session ID string and a list of Stun servers:

```
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);
}

/// 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 the reference to that object should substitute `this` in the call to the `CLI_UC` constructor.

Note: The `CLI_UC` object, in common with many of the objects provided by the CLI wrapper, needs to be disposed of when it is no longer useful. This is accomplished by calling its Dispose method, e.g.

```
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 OnConnectionLost();
void OnConnectionReestablished();
void OnUnknownWebsocketMessage(string s);
```

Making and receiving calls

You can obtain a `CLI_Phone` object from the `CLI_UC` by using its `GetPhone` method, for example:

```
CLI_Phone phone = mUC.GetPhone();
```

The `CLI_Phone` object is used to make and receive calls.

To make calls, use the `CLI_Phone.CreateCall` method which is overloaded as follows:

```
CLI_Call CreateCall(string address, bool audio, bool video);
CLI_Call CreateCall(string address, bool audio, bool video, ICLI_CallListener listener);
```

The parameter address contains the phone number that should be reached; audio is set to true to allow audio to be sent; video is set to true to allow video to be sent; and listener is a reference to an object that implements `ICLI_CallListener`. If not specified immediately, it can be specified by calling the `SetListener` method on the `ICLI_CallListener` object.

When an inbound call is being connected, the `CLI_Phone` object calls an object that implements the `ICLI_PhoneListener` interface that has one method:

```
void OnIncomingCall(CLI_Call newCall);
```

The object that implements `ICLI_PhoneListener` is notified to the `CLI_Phone` object by calling its `SetListener` method. The `CLI_Call` object that is passed to the `OnIncomingCall` method has already been set up with details such as the address of the caller. If it is desired to accept the call, the `Answer` method of the `CLI_Call` object needs to be called whose signature is:

```
CLI_Call.Answer(bool withAudio, bool withVideo);
```

Of course, `withAudio` should be set to true to allow audio to be sent to the remote end, and `withVideo` works in the same way.

To reject the call, the `End` method of the `CLI_Call` object needs to be called; that takes no parameters.

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 they need to be hooked up to allow video to be displayed.

The `CLI_Phone` object has two methods, `SetPreviewViewName` (associated with local video) and `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. This method is called whenever a new frame has been received from the pipe and is available for painting.

To obtain the frame, the method `GetImageData` returns a `CLI_VideoViewImageData` object that contains the image data. To render the frame, a `CLI_VideoView` object is required. The `CLI_VideoView` has a method `SetImageData` that takes the `CLI_VideoViewImageData` object that came from `GetImageData` as its argument. It also has a method Paint that takes a `PaintEventArgs` object; such an object is passed to a window's `OnPaint` method when it is time to redraw that object. This contains a device context into which the video frame is rendered.

The `CLI_VideoView` calls an `ICLI_ViewRefresher` object's `RefreshViewFunc` when each frame is ready to display. That object is set by calling its `SetRefreshViewFunc` method. The usual behaviour of the `RefreshViewFunc` method is to invalidate the region of the client area in which the video is being rendered.

Receiving a call

When a call is received, the `OnIncomingCall` method of the `ICLI_PhoneListener` interface is called with a `CLI_Call` object. The call can be answered by using the `Answer` method of the `CLI_Call` object. It can be rejected by using the `End` method.

Monitoring the state of a call

State transitions that occur during the progress of a call are reported by the library calling back to your `ICLI_CallListener` object using the method `OnStatusChanged(CLI_CallStatus newStatus)`. `CLI_CallStatus` is an enum with the following values: `UNINITIALIZED`; `SETUP`; `ALERTING`; `RINGING`; `MEDIA_PENDING`; `IN_CALL`; `TIMED_OUT`; `BUSY`; `NOT_FOUND`; `CALL_ERROR`; `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/call failures (`OnDialFailed` and `OnCallFailed`).

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, just 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.

A comma character can be used to insert a two-second pause into a sequence of tones.

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(false, false);
```

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();
}
```

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.

The video resolution is set on the `CLI_Phone` object:

```
phone.SetPreferredCaptureResolution(
CLI_VideoCaptureResolution.RESOLUTION_1280x720);
```

Available options are `RESOLUTION_AUTO`; `RESOLUTION_352x288`; `RESOLUTION_640x480`; and `RESOLUTION_1280x720`

The frame rate can also be set as follows:

```
phone.SetPreferredCaptureFrameRate(20);
```

Note: 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.

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.

Creating a Topic

A topic can be created by 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 the form of the API that includes an expiry period is used, that expiry period is expressed in minutes. The topic is in that case removed from the server after the topic has been inactive for that period of time. If the topic is created without an expiry period, it remains until 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: 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. The value is obtained by 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()`. It is also possible to call `GetAsInt()` or `GetAsDouble()` in which case the default value used 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 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 `OnTopicUpdated` callback is called 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 the message is sent successfully, 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 it is attempted 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 called by all clients connected to the topic when it is deleted from the server as a result of any client calling `Disconnect(true)`. 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

The network connection status is monitored and the library calls back the application if any issues are detected.

The `ICLI_UCListener` object callback `OnConnectionRetry(System.Byte attemptNumber, System.UInt16 delayInSeconds)` is called if the connection is lost for a short period and the library is attempting to re-establish communications. The callback is called just prior to making each attempt. Attempts are made at increasing intervals (for example, 0.5s, 1.0s, 2.0s, and 4.0s).

If the connection is re-established, the `OnConnectionReestablished()` method is called.

If the connection is lost, the `OnConnectionLost()` method is called.

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