Fusion Client SDK

Developer's Guide

Version 3.3

CAFÉ

Updated: 2018-06-18

Document version: 3.3/1

Contact Information

For technical support or other queries, contact CaféX Communications Support at:

support@cafex.com

For our worldwide corporate office address, see:

http://www.cafex.com

Documentation Set

- *FCSDK Overview Guide*
- *FCSDK Architecture Guide*
- *FCSDK Installation Guide*
- *FCSDK Administration Guide*
- *FCSDK Developer Guide*

Related Documentation

Fusion Application Server

- *FAS Architecture Guide*
- *FAS Installation Guide*
- *FAS Administration Guide*

Contents

Creating the Web Application

See the ***FCSDK Architecture Guide*** for details on how the Web Application fits into the **Fusion Client SDK** architecture.

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

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

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

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

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

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

Communication with the Client

While the **Fusion Client SDK** defines the way in which both the Web Application and client communicate with the Web Gateway, it does not restrict how the Web Application communicates with the client.

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

Authenticating the User

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

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

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

Creating the Session

After it has authenticated the user, the Web Application sends a `POST` request to the Web Gateway. The request describes the requested capabilities for the session, such as:

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

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

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

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

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

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


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

```
    },
    "uuiData":"0123456789ABCDEF"
  }
```

where the members are:

Member	Description
`timeout`	The timeout period for the session, defined in minutes. If omitted, this is set to 1 by default. **Note:** The valid timeout range is 1-15 minutes, setting it to any other value outside of this range will cause errors.
`webappid`	The unique ID that the web app passes to the Gateway to identify itself. The ID must be a minimum of 16 characters in length, and must also have been configured on the Gateway itself.
`allowedOrigins`	This represents the origins from which cross realm JavaScript calls are permitted. If null or empty, there is no restriction. This is a comma separated list.
`urlSchemeDetails`	The connection details the **Fusion Client SDK** client library is configured to use to the Web Gateway. This is an object with three other settings. If these details are not provided, the default setting for each option is used: ▪ `secure` If `true`, connects using secure WebSockets (`wss`). The default value is `false`, for non-secure (`ws`). ▪ `host` Specifies the host name or IP address for the WebSocket to connect to. If not provided, the client uses the `<web_gateway_address>` that the Web Application used to issue the HTTP `POST` request. Typically, this value is set when a NAT firewall is placed between the clients and the gateway. This value should be set to the external host name or IP address. ▪ `port` Specifies the port that the WebSocket connects to. The default is set to

Member	Description
	8443 if `secure` is `true` or 8080 if `secure` is `false`.
`voice`	The details regarding voice and video calling. If omitted, voice and video calling are disabled. It is an object with the following members: ▪ `username` The SIP user name, as would appear in the `From` header ▪ `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. ▪ `inboundCallingEnabled` Set inbound calling parameters to disable inbound calling. If omitted, inbound calling is enabled by default. **Note:** If `inboundCallingEnabled` is set to `true`, a SIP `REGISTER` request is sent to the SIP network; therefore, a corresponding user must exist on the SIP network. This user's credentials should be entered in the `auth` section (see below). 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. You can omit this section if the gateway is a trusted entity in the SIP infrastructure; however, you omit it and the SIP is challenged, the registration fails.

Member	Description
	▪ `username` The user name you would 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 omitted, AED functionality is disabled. It is an object with the following members: ▪ `accessibleSessionIdRegex` A Java regular expression which defines the AED topic names which this session can subscribe to. The user will not be able to subscribe to any AED topic which does not match this expression. ▪ `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, on all topics they are subscribed to.
`uuiData`	If provided, this string is used to populate SIP `INVITE` and `BYE` messages sent by the user with a `User-to-User` header. As an example, suppose the value of this parameter is `ABCD`. The FCSDK adds the header `User-to-User:`

Member	Description

	ABCD. If omitted, no User-to-User header is added to SIP messages.
	Examples of valid uuiData values are:
	`abcdef;encoding=hex`
	`abcdefghijk;encoding=blah;paramname=paramvalue`
	`"abcdefghijk";encoding=blah`

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

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

Examples

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

Voice and Video Calling

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

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

Voice and Video Calling with URL

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

```
{
   "webAppId":"WEBAPPCSDK-A8C1D",
   "allowedOrigins":["example.com"],
   "urlSchemeDetails":
   {
      "secure":true,
```

```
      "host":"wg.example.com",
      "port":"8443"
    },
    "voice":
    {
      "username":"jbloggs",
      "displayName":"Joseph",
      "domain":"example.com",
      "inboundCallingEnabled":true,
      "allowedOutboundDestination":"sip:user@example.com",
      "auth":
      {
        "username":"1234",
        "password":"123456",
        "realm":"example.com"
      }
    }
  }
```

AED Only

For a client application with AED capabilities only:

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

Using the UUI

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

```
  {
    "webAppId":"WEBAPPCSDK-A8C1D",
    ...
    "uuiData":"53656e73697469766520446174 61;encoding=hex"
  }
```

You might use this technique to put some sensitive data in the `User-to-User` header which the consumer application does not know (or at least, does not transmit). When the consumer application requests a session token, the Web Application puts the sensitive data, known only to itself, in the `uuiData` element of the JSON which it sends to the session token servlet. The Web Gateway associates that data with the session it creates. When the consumer application makes a call to a SIP device using that session, the Web gateway populates the `User-to-User` header of the `INVITE` which it sends to the SIP device with the sensitive data. How the SIP device uses the data is a matter for the device itself, but it could be an authentication token which allows it to set up the call.

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

JSON Response

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

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

Ending the Session

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

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

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

Note:

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

Creating a Browser Client Application

Fusion Client SDK enables you to develop browser-based applications offering users the following methods of communication:

- Voice and video calling
- Application Event Distribution

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

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

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

For more detailed discussion of the **Fusion Client SDK** solution, refer to ***FCSDK Overview Guide***.

Note: Fusion Client SDK is delivered with a sample application. This is available at `<installation_directory>/Core_SDK/Sample_Source`. All samples featured in this guide can be located there.

Setting up a Project

Note: Before setting up a project for a client application, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See the Creating the Web Application section on page 14.

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

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

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

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

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

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

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

Initializing the SDK

To set up all the functionality to which the user has access, you must obtain a session ID from your Web Application (see the Creating the Web Application section on page 14), and initialize the `UC` object by calling the `start` method on the `UC` object with it.

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

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

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

UC.onInitialisedFailed = function() {
   ...
};

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

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

Note:

- STUN servers are not necessary if the Gateway is not behind a firewall, so that *Network Address Translation* is not needed; in this case the `stunServers` array can be empty. You can provide your own STUN server instead of the public Google one above; and you can provide more than one in the array, in which case FCSDK tries them in sequence until it finds a working one.
- Your Web Application may fail to return a session ID (for instance, if it cannot authenticate the user). In these situations the user should be logged out and needs to log in again to start a new session (see **the Creating the Session section on page 15**).

Checking Browser Compatibility

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

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

Member	Values
`pluginRequired`	▪ `true` If the browser needs a plugin to operate correctly ▪ `false` Otherwise
`status`	▪ zero length string If `pluginRequired` is `false` ▪ `installRequired` If the plugin is missing

Member	Values
	■ `upgradeRequired` If the plugin is present but a new version is needed ■ `upgradeOptional` If the plugin is present and will work, but a newer version is available ■ `upToDate` If the plugin is present and is the latest available version
`restartRequired`	■ `true` If a plugin is installed or upgraded, the browser will need to be restarted ■ `false` The browser will not need to be restarted, or no plugin is needed
`installedVersion`	■ `none` When `pluginRequired` is false or the plugin is missing ■ string in form `x.y.z` Where `x`, `y`, and `z` are integers
`minimumRequired`	■ `none` When `pluginRequired` is false or the plugin is missing ■ string in form `x.y.z` Where `x`, `y`, and `z` are integers. This is the minimum version of the plugin which will work correctly with the version of FCSDK in use.
`latestAvailable`	■ `none` When `pluginRequired` is false or the plugin is missing ■ string in form `x.y.z` Where `x`, `y`, and `z` are integers. This is the latest version of the plugin available on the server.

Member	Values
`pluginUrl`	▪ zero length string If `pluginRequired` is `false` ▪ URL string The URL points to the location of the latest version of the plugin on the server (the version indicated by `latestAvailable`).

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

Note: The user may not have permission to install plugins. In this case, it is the responsibility of their IT administrator to install the correct plugin, and the application should inform the user of the problem.

Adding Voice and Video

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

Adding a Preview Window

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

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

Alternatively, you can wait until you have a call (see the Making a Call section on the next page and the Receiving a Call section on page 30) before setting the preview element:

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

Making a Call

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

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

For both parameters, the possible values are:

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

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

Note: call.dial() must only be called after the application has initialized the SDK and received the UC.onInitialised callback (see the Initializing the SDK section on page 25).

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

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

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

```
    };

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

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

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

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

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

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

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

Receiving a Call

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

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

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

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

To answer the call, your client application should call the `Call` object's `answer` method.

The `answer` method takes two string parameters:

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

For both parameters, the possible values are:

- `enabled` – for 2 way media
- `onlyreceive` – for 1 way media

- `disabled` – for no media

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

To reject the call, your client application should call the `Call` object's `end` method.

Enabling Local Media

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

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

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

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

Adding a Stream

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

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

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

The Size of the Video Window

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

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

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

Ending a Call

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

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

Muting the Local Audio and Video Streams

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

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

Holding and Resuming a Call

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

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

Sending DTMF Tones

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

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

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

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

Handling Multiple Calls

Applications developed with **Fusion Client SDK** JavaScript SDK support multiple simultaneous calls:

- To make additional calls while another call is in progress, the client application would use the `UC.phone.createCall(numberToDial)` method (see **the Making a Call section on page 29**).
- To receive incoming calls while another call is in progress, the `UC.phone.onIncomingCall` method should be triggered (see **the Receiving a Call section on page 30**).

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

Setting Video Resolution

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

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

Enumerating Possible Resolutions

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

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

These values are an enumeration which list all supported resolutions:

Enumeration Value	Width	Height
`videoCaptureResolution174x144`	174	144
`videoCaptureResolution352x288`	352	288
`videoCaptureResolution320x240`	320	240
`videoCaptureResolution640x480`	640	480
`videoCaptureResolution1280x720`	1280	720
`videoCaptureResolution1920x1080`	1920	1080

Note: When you set the resolution, the device's camera may not support that resolution. In that case the browser provides a different resolution, but exactly what resolution will depend on the browser being used.

Setting the Resolution

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

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

Setting an Arbitrary Video Resolution

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

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

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

Setting the Frame Rate

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

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

Note: If the hardware cannot manage the preferred frame rate, the browser may interpolate to get as close to the desired frame rate as possible, or it may choose the closest frame rate which the hardware supports. Achieving the preferred frame rate cannot be guaranteed.

Input Device Switching

Often, a user has more than one input device attached. This is commonly the case with audio input devices (microphones), and is becoming increasingly common with video input devices (front and back cameras on tablets, for example). An application can set its preferred audio and video devices using the functions:

```
UC.phone.setPreferredAudioInputId(id);
```

and

```
UC.phone.setPreferredVideoInputId(id);
```

before a call starts. The `id` parameter is a string which uniquely identifies the input device; it may also be `'default'`, which allows the browser to choose the input device itself. To find the ID of a particular device, you must list all the input devices, and use the ID of one of them; see the Getting the Input Devices section on the next page.

Note: You can set the preferred audio or video ID to a value which does not correspond to an input device.

- If you call `Call.dial` (see the Making a Call section on page 29) while the preferred media ID is invalid in this way, FCSDK calls `onGetUserMediaError`, followed by `onCallFailed`.
- If you call `Call.answer` on a received call (see the Receiving a Call section on page 30) while the preferred media ID is invalid, FCSDK also calls `onGetUserMediaError`, followed by `onCallFailed`.
- If you make or receive a call as audio-only or video-only using the input parameters `withAudio` and `withVideo` of `dial` and `answer`, the preferred input ID of the inactive media may be invalid without affecting the call.

The application can get the current preferred audio and video input devices using:

```
var id = UC.phone.getPreferredAudioInputId();
```

and

```
var id = UC.phone.getPreferredVideoInputId();
```

Initially, the browser prefers the default device, and the ID returned by these functions is `'default'`; after local media has been established, the functions return the actual ID of the currently preferred device.

Getting the Input Devices

An application can receive a list of available input devices by implementing the `onGetMediaDevices` callback on the `Phone` object:

```
UC.phone.setOnGetMediaDevices(function(devices) {
   ...
});
```

The `devices` object contains two arrays, `videoinputs` and `audioinputs`, either of which may be empty (if there are no local media devices of that type); each element of each array is an object which contains a `label` and an `id`:

```
{
   "videoinputs" : [
      {
         "label" : "Microsoft® LifeCam HD-3000 (045e:0779)",
         "id" :
         "c5cfe4b705510e08e43346e262e81bc26bb1207e5ca0f12e0d45750099740c37"
      },
       ...
   ],
   "audioinputs" : [
      {
         "label" : "Default",
         "id" : "default"
      },
      {
         "label" : "Built-in Microphone",
         "id" :
         "77c7f2e78a889bd84a83e0df6cf76699338dde84246d342469891e91e3711cb4"
      },
       ...
   ]
}
```

The `label` element is a moderately informative description of the device; `id` is the value to be used as a parameter for the `setPreferredAudioId` and `setPreferredVideoId` functions. The callback function will be called more than once because the device labels change after local media becomes established. Before that, the labels are uninformative, such as `Mic 1` and `Cam 2`.

Handling User Media Issues

When a problem occurs obtaining the user media (microphone or camera) the **Fusion Client SDK** provides three call backs on the `Call` object:

- `onOutboundAudioFailure` is called if there is a problem obtaining audio media (such as if the microphone is disabled or unplugged), but the call can continue without it. The application can decide whether to continue or end the call when it receives this callback.
- `onOutboundVideoFailure` is called if there is a problem obtaining video media (such as if the camera is disabled or unplugged), but the call can continue without it. The application can decide whether to continue or end the call when it receives this callback.
- `getUserMediaError` is called when there is a terminal user media problem which has resulted in the call ending (for instance, if the user has denied permission to both the microphone and camera).

Note: FCSDK will only call the `onOutboundAudioFailure` and `onOutboundVideoFailure` methods if the relevant media type was requested when making the call; the timing of these method callbacks will vary depending on the browser.

Monitoring Call State

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

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

Callback	Meaning
`Phone.onIncomingCall`	An incoming call is alerting (ringing). The callback provides the `Call` object as a parameter. See the Receiving a Call section on page 30.
`Call.onRinging`	An outgoing call is ringing at the remote end
`Call.onPending`	The call is connected, and waiting for media
`Call.onInCall`	The call is fully set up, including media
`Call.onBusy`	The dialed number is busy

Callback	Meaning
`Call.onNotFound`	The dialed number is unreachable or does not exist
`Call.onTimeout`	There was no response from the dialed number within the network's timeout
`Call.onDialFailed`	Dialing the number failed. This may be due to several reasons, such as no media broker being available, or the capacity of the network being reached. The callback has an error code parameter which may give more information.
`Call.onCallFailed`	The call has errored. This may be due to no media, or a network failure, or some other reason. The callback supplies an error code as a parameter, which may give more information.
`Call.onEnded`	The call has ended

Adding Application Event Distribution

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

Creating a Topic

To create a topic, the client application can call:

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

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

Subscribing to a Topic

After it has created the topic, the client application subscribes to it by calling:

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

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

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

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

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

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

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

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

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

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

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

  topic.onConnectSuccess = function(data) {
```

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

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

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

   topic.connect();
};
```

Note: The application can compare the `version` of a data object with the `version` of a stored version of the same data to ensure that an older version does not replace a newer one.

Unsubscribing from a Topic

The client can un-subscribe to a topic by calling

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

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

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

- `onDeleteTopicSuccess`
- `onDeleteTopicFailed`

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

Publishing Data to a Topic

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

```
topic.submitData(key, value);
```

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

- `onSubmitDataSuccess`
- `onSubmitDataFailed`

All clients which are subscribed to the topic receive an `onTopicUpdate` event.

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

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

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

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

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

   topic.onSubmitDataFailed = function(key, value, message) {
      // Notify user of failure
   }

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

   topic.connect();
};
```

Deleting Data from a Topic

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

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

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

- `onDeleteSuccess`
- `onNotFound`.

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

Sending a Message to a Topic

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

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

The `msg` parameter is free-form text.

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

- `onSendMessageSuccess`
- `onSendMessageFailed`

All clients subscribed to the topic receive an `onMessageReceived` event.

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

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

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

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

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

   topic.connect();
};
```

Adding Floor Control

Note: The following section applies only to the Chrome browser.

When dialing into a multi-party conference on a conference server which supports **Binary Floor Control Protocol** (BFCP), BFCP operations are available from the `bfcp` element of the `Call` object. The application can request and release the floor, and can receive various callbacks that provide information about the state of the conference. Other operations (including all BFCP Chair operations) are not supported.

Clearly, requesting the floor is not something which an application should do automatically, but it could do it in response to a user action which indicates that the user wants to request the floor.

Requesting the Floor

A client requests the floor of the conference by calling `call.bfcp.requestFloor()`. Requesting the floor does not mean that the floor is automatically granted, so the client application must wait until it receives the `onFloorRequestAccepted` callback to indicate that it has been granted the floor. To do this, it must first set the `call.bfcp.onFloorRequestAccepted` element to a function which is called when the client receives the floor.

The Floor Request is Accepted

Once the floor request has been accepted, the client can stream video to the conference. Note that floor requests are not necessarily granted immediately, or even very quickly - in a conference with several participants, others may have requested the floor first, and the client may be put in a queue. The client must be prepared to receive `onFloorRequestAccepted` at any time after it has made the floor request.

The Floor Request is Rejected

A floor request can fail, in which case the client receives one of the callbacks:

- `onBFCPUnavailable`

  Indicates that no floor control is available for this call (either because the conference is not under the control of a BFCP conference server, or because the client is not allowed to make BFCP requests).

- `onNoContentStreamAvailable`

  Indicates that the request was made before the client had any content stream to send to the conference. In order to avoid this, a real client application would wait until it received a callback indicating that a media stream was available before allowing the user to request the floor.

- `onFloorControlEnded`

  This can mean a number of things, but if received instead of an `onFloorRequestAccepted`, it indicates that the floor request has been denied by the conference server.

Releasing the Floor

Once the floor has been granted, the client application can stream video to the conference, and it can eventually to release the floor by calling `call.bfcp.releaseFloor()`. When it has released the floor, the client receives an `onFloorControlEnded` callback.

Canceling a Floor Request

The client may also call `releaseFloor` to cancel a floor request which has not yet been granted. In this case, the client receives an `onFloorControlEnded` callback instead of `onFloorRequestAccepted`. However, since the floor control server may have already granted the floor to the client when it receives the `releaseFloor` message canceling the floor request, the client may receive the `onFloorRequestedAccepted` callback as well.

The Floor is Taken from the Client

The conference server may take the floor away from the client at any time after it has granted it the floor (for example, to give the floor to another conference participant, or to close the conference). If this happens, the client receives an `onFloorControlEnded` callback. This is the client's opportunity to adjust its internal state and its user interface to indicate that it is no longer streaming media to the conference.

Another User is Granted the Floor

When the conference server grants the floor to another user, the client receives an `onFloorTaken` callback. The client should never receive this while it has the floor, but may receive it at any time when it does not, including *after* making a floor request, but *before* receiving the `onFloorRequestAccepted` callback; it is especially likely immediately after receiving the `onFloorControlEnded` callback.

The `onFloorTaken` callback may contain information about the user who has been granted the floor, in the form of a `UserInfo` object containing a `name` and a `uri`. The client can update its user interface to indicate who has the conference floor, but should not rely on this information being available, and should check that the `UserInfo` parameter is not `undefined`.

Another User Releases the Floor

If another user releases the floor, or is removed from the floor by the conference server, the client receives an `onFloorReleased` callback. This is an opportunity for the client to update its UI to indicate that the floor is not taken by any user. If the conference server removes one participant from the floor and immediately grants the floor to another participant (other than the client), then the client may receive only the `onFloorTaken` callback for the new floor owner, or it may receive both `onFloorReleased` followed immediately by `onFloorTaken`. The client should therefore not rely on receiving this callback, and should be ready to transition directly from one participant having the floor to another participant having it, without necessarily going through an intermediate state in which no one has the floor.

A Simple Floor Control Client Program

```
call.bfcp.onFloorRequestAccepted = function() {
  // Update UI to show client has the floor
  // Stream video-only, then
  // Stop streaming video, then
  call.bfcp.releaseFloor();
};

call.bfcp.onBFCPUnavailable = function() {
  alert("No floor control possible");
};

call.bfcp.onNoContentStreamAvailable = function() {
  alert("Tried to request floor before content ready");
};

call.bfcp.onFloorTaken = function(UserInfo user) {
  if (user != undefined) {
    // Display user.info and/or user.uri for user who now has the floor
  }
};

call.bfcp.onFloorReleased = function() {
```

```
    // Remove floor information from the UI
  };

  call.bfcp.onFloorControlEnded = function() {
    // Update UI to show client no longer has the floor
  };
  call.bfcp.requestFloor();
```

The above skeleton program releases the floor as soon as it is granted it. A real program would use the `onFloorRequestAccepted` callback to start streaming content, and would call `releaseFloor` in response to a user command to do so.

Floor Control and HA Cluster Failover

In the case of a multi-node cluster, when failover occurs, and control of an existing call moves from one node in the cluster to another, floor control requests and messages may be lost; because of that, the floor control state on the client is resynchronized with that known to the server, after a call has moved from one node to another. When this happens, the client may receive an unexpected `onFloorControlEnded` or `onFloorRequestAccepted` notification. Clients do not normally deal with this explicitly (their normal processing of these messages should be enough), but developers should be aware that these messages may be received.

Ending the Session

To end the session, the client application needs to call in to the Web Application, which can terminate the session as described in the Ending the Session section on page 22.

Responding to Network Issues

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

Reacting to Network Loss

There are two sightly different scenarios:

- FCSDK loses the network connection to the web.

  First, the `UC` object receives a `onNetworkUnavailable` callback.

  - If the network connection cannot be re-established within 20 seconds, `onNetworkUnavailable` is followed by `onConnectivityLost`.
  - If the connection to the network *is* re-established within 20 seconds, FCSDK attempts to re-establish the connection to the Gateway.

    During this process it will make fifteen attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s, 5s, 5s, 5s, 5s, 5s, 5s, 6s, 6s. A call to the `onConnectionRetry(attempt, delayUntilNextRetry)` method of the `UC` object precedes each of these attempts.

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

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

- FCSDK loses connection to the Gateway without losing its network connection

  In this case, FCSDK attempts to automatically re-establish its connection to the Gateway.

  During this process it will make fifteen attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s, 5s, 5s, 5s, 5s, 5s, 5s, 6s, 6s. A call to the `onConnectionRetry(attempt, delayUntilNextRetry)` method of the `UC` object precedes each of these attempts.

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

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

In either case, if the application receives `onConnectivityLost`, it means that FCSDK is unable to re-establish a connection to the Gateway, and the application itself must take some action; at the very least it must inform the user that they are no longer connected.

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

Network Quality Callbacks

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

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

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

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

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

Creating an iOS Client Application

Fusion Client SDK enables you to develop iOS applications offering users the following methods of communication:

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

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

Developing iOS applications using **Fusion Client SDK** requires Xcode 4.5 or later.

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

The **Fusion Client SDK for iOS** is made up of the following classes:

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

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

Setting up a Project

Note: Before setting up a project for a client application, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See the Creating the Web Application section on page 14.

To set up a project including the **Fusion Client SDK**, 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 `iOSFusionSDKSample`.
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`
   - `VideoToolbox.framework`
   - `Metal.framework`
   - `libsqlite3.tbd` (or equivalent alternative)
   - `libc++.tbd` (or equivalent alternative)

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

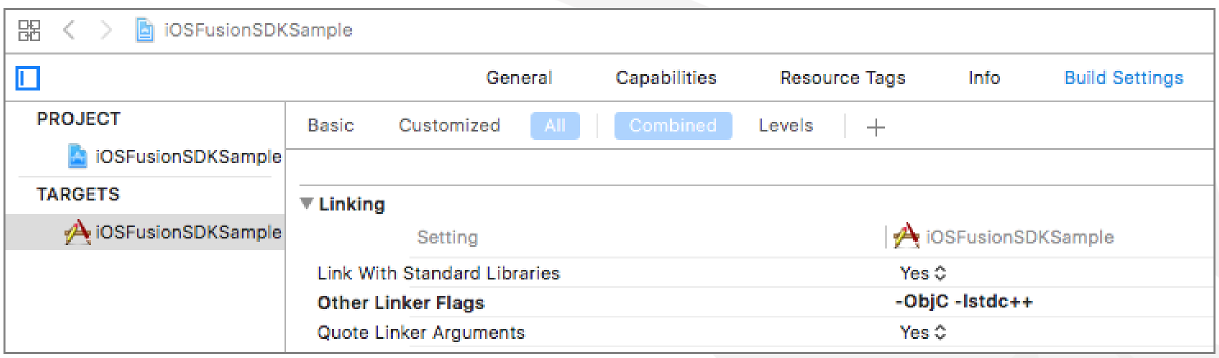
Now, you need to add the **Fusion Client SDK** framework to your project.

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

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

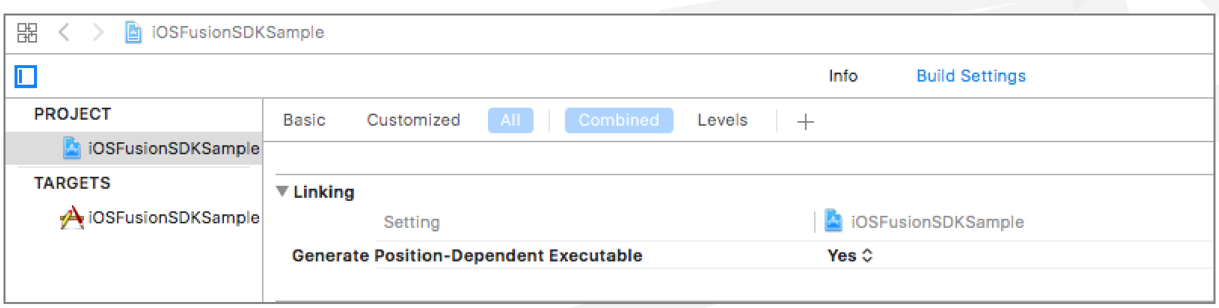
1. Select your project and click **Build Settings**.
2. Enter an appropriate term in the search field to find the **Other Linker Flags** setting, for example `Linker`. Click **Search**. **Other Linker Flags** display in the *Linking* section.
3. Configure **Other Linker Flags** with the following setting:

   `-ObjC -lc++`



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

4. **Position Independent Executables** are incompatible with some codecs in the **Fusion Client SDK for iOS**. In the *Linking* section, set **Generate Position Dependent Executable** to `Yes`.



Important: When building a project created with Xcode 5, you may get linkage errors related to the Standard C++ library. This is caused by an Xcode 5 bug, which prevents it detecting the dependency the iOS SDK has on the C++ Standard Library. To work-around this issue, add `libstdc++.6.dylib` to the list of required libraries.

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

iOS 9 and Xcode 7

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

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

1. Disable the generation of bitcode

   `Enable Bitcode = NO`.

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

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

Note: The iOS sample application has been modified accordingly.

Initializing the ACBUC Object

The application accesses the API initially via a single object, ACBUC. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see the Creating the Web Application section on page 14), and initialize the ACBUC object using it. Once it has received the Session ID, the client application must call the `ucWithConfiguration` method on the ACBUC:

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

```
}

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

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

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

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

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

Adding Voice and Video

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

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

Requesting Permission to use the Microphone and Camera

On iOS 7.0 and higher, your application needs to ask the end user for permission to use the microphone and camera before they can make or receive calls. 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. iOS remembers the answer they provide until your application is uninstalled or the permissions are reset in the *iOS Settings*. The end user can also change the microphone and camera permissions for your application in *iOS Settings*.

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

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

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

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

Making a Call

In the following example, the application makes a call (using `createCallToAddress` on the `ACBClientPhone` object) as soon as the session has started (see the Initializing the ACBUC Object section on page 53):

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

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

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

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

Receiving a Call

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

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

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

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

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

Note:

- The older form, `answerWithAudio:video:`, which took two boolean values, is now deprecated.

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

Receiving Calls when the Client is in Background or Suspended Mode

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

- App plays audio
- App provides Voice over IP services

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

Video Views and Preview Views

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

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

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

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

Ending a Call

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

To receive notification that the remote party has terminated the call, the application must monitor the state of the call (see the Monitoring the State of a Call section on page 103) for the `ACBClientCallStatusEnded` state.

Muting the Local Audio and Video Streams

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

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

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

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

Holding and Resuming a Call

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


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

DTMF Tones

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

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

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

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

- The second parameter is a boolean which indicates whether the application should play the tone back locally so that the user can hear it.

Handling Multiple Calls

Applications developed with **Fusion Client SDK for iOS** do not support multiple simultaneous calls.

Setting Video Resolution

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

Enumerating the Possible Resolutions

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

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

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

The supported resolutions are:

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

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

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

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

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

Setting the Resolution

The application can set the captured video resolution using the `preferredCaptureResolution` property of the `ACBClientPhone`object. The value supplied must be one of the resolutions presented in `recommendedCaptureSettings`, as described the Enumerating the Possible Resolutions section on page 59

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

Alternatively, one of the values from the enumeration:

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

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

Setting the Frame Rate

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

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

Alternatively, it can try to set a custom frame rate:

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

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

Dial Failures

FCSDK does not call the `ACBClientCallDelegate` failure methods (`didReceiveDialFailure` and so on) for failures caused by a timeout. This results in the client seeing the **Trying to call...** dialog, despite the call being inactive. To avoid this, handle these timeout errors using the status delegate methods; examples can be found in the Monitoring the State of a Call section on page 103 and in particular to the callback:

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

Handling Device Rotation

The SDK automatically handles control of the video orientation.

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

Switching between the Front and Back cameras

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

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

Two parameters can be passed to this method:

- `AVCaptureDevicePositionBack`
- `AVCaptureDevicePositionFront`.

These enumeration values are in `<AVFoundation/AVCaptureDevice.h>`.

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

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

Application Background Mode

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

It is an application developer's responsibility to consider both functional and privacy implications, and decide whether their application should mute audio and video when transitioning to background mode (see the Muting the Local Audio and Video Streams section on page 58).

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

Note: The behavior of iOS is different to Android.

Monitoring the State of a Call

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

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

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

The application can adjust the UI by switching on the value of the `status` parameter, to give the user suitable feedback, for example by playing a local audio file for ringing or alerting:

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

```
}
```

The following table gives the possible status codes:

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

Adding Application Event Distribution

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

To create an AED application, you need to:

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

Creating and Connecting to a Topic

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

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

or the `createTopicWithName:expiryTime:delegate:` method:

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

The `name` of the topic is an `NSString`, and the `expiryTime` parameter is a time in minutes. A topic created with an expiry time will be automatically removed from the server after the topic has been inactive for that time. When created without an expiry time, the topic exists indefinitely, and the application must delete it explicitly (see the Disconnecting from a Topic section on the next page). The `delegate` is an object conforming to the `ACBTopicDelegate` protocol.

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

didConnectWithData

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

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

```
- (void)topic:(ACBTopic*)topic didConnectWithData:(NSDictionary*)data
{
  //topic data is an array containing all our key/value pairs
  NSArray *topicData = [data objectForKey:@"data"];
  if([topicData count] > 0)
  {
    //we can show our users the data in the topic as follows
    for(int i = 0; i < [topicData count] ; i++)
```

```
        {
          NSString* keyField = [[topicData objectAtIndex:i]
          valueForKey:@"key"];
          NSString* valueField = [[topicData objectAtIndex:i]
          valueForKey:@"value"];
          // Display key and value
        }
      }
    }
```

Disconnecting from a Topic

You can either disconnect from the topic without destroying it:

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

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

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

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

topicDidDelete

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

Publishing Data to a Topic

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

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

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

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

didUpdateWithKey

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

Deleting Data from a Topic

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

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

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

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

Sending a Message to a Topic

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

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

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

didReceiveMessage

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

Threading

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

Self-Signed Certificates

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

You can obtain the server certificate and CA root certificate through the FAS Administration screens. The ***FAS Administration Guide*** explains how to view and export certificates. You need to extract the HTTPS Identity Certificate (server certificate) and the Trust Certificate (CA root certificate) that has signed your server certificate.

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

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

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

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

Note: Since iOS 9, you also need to add a setting to your application's `plist` file to allow connection to a server using self-signed certificates. Set **Allow Arbitrary Loads** under **App Transport Security Settings** to `YES`.

Testing IPv6

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

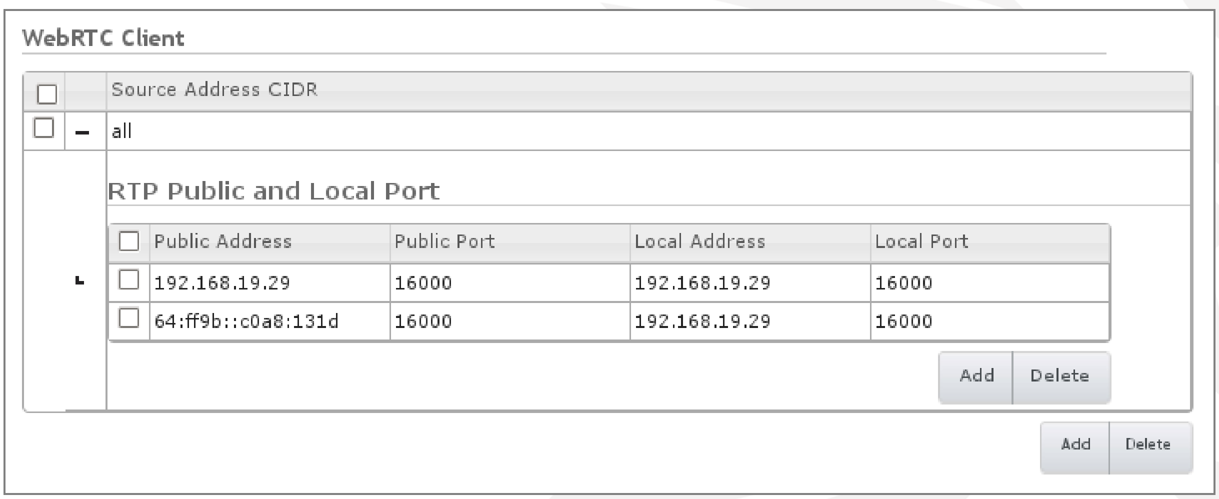
https://developer.apple.com/library/ios/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/UnderstandingandPreparingfortheIPv6Transition/UnderstandingandPreparingfortheIPv6Transition.html

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

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

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

   If using an Apple laptop hotspot, then the IPv6 address equivalent starts with
   `64::ff9b::`
   and is followed by the hexadecimal version of the IPv4 address. For example `c0a8:131d` is the equivalent of `192.168.19.29`



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

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

Bluetooth Support

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

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

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

Starting and Stopping ACBAudioDeviceManager

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

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

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

After the `ACBAudioDeviceManager` starts, the application can call its methods to set the audio devices which the phone should use for calls made or received by the application. Calls which are not handled by the FCSDK application will be unaffected, and will use the phone's default behavior.

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

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

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

Setting the Preferred Device

The application can set the preferred device for the call:

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

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

- `ACBAudioDeviceSpeakerphone`

  Audio goes to the loudspeaker in the phone, and is audible to others in the vicinity. Audio input is from the phone's internal microphone.

- `ACBAudioDeviceWiredHeadset`

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

- `ACBAudioEarpiece`

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

- `ACBAudioDeviceBluetooth`

  Audio is sent to and received from a paired Bluetooth device.

- `ACBAudioDeviceNone`

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

If the preferred device is available when the application calls `setAudioDevice`, the call starts using that device; otherwise, there is no immediate change, but if it later becomes available (the Bluetooth device is switched on or is otherwise recognized), then the audio switches to this device.

Setting the Default Device

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

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

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

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

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

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

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

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

Listing Available Devices

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

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

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

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

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

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

Responding to Network Issues

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

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

Reacting to Network Loss

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

When all reconnection attempts are exhausted, the `ACBUCDelegate` receives the `ucDidLoseConnection` callback, and the retries stop. At this point the client application should assume that the session is invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in the Creating the Session section on page 15.

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

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

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

Reacting to Network Changes

If the issues with the network are caused by a temporary loss of connectivity (for example, when moving between two Wi-Fi networks, or from a Wi-Fi network to a cellular data connection), the client application should not log out from the session and log back in (as described in the Reacting to Network Loss section on page 105), as all session state will be lost.

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

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

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

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

Network Quality Callbacks

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

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

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

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

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

Creating an Android Client Application

Fusion Client SDK enables you to develop Android applications offering users the following methods of communication:

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

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

To develop Android applications using **Fusion Client SDK**, your system will need to conform to the system requirements listed at `http://developer.android.com/sdk/index.html`.

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

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

Note: The structure of an Android application revolves around different `Activity` objects, and the sample code included with the FCSDK Android SDK shows a typical structure. In the sample code, there is a `LoginActivity`, which gets the session token from the server (see **the Creating the Session section on page 15**); a `Main` Activity, which creates the `UC` object and connects to the session (see **the Creating the UC Object section on the next page**); and an `InCallActivity`, which makes and receives calls, and works with those calls while they are in progress. AED operations (see **the Adding Application Event Distribution section on page 86**) are centralized in the `AEDFragment` and `AEDTopicManager` classes. We recommend that Android applications which make use of the FCSDK should separate their operations similarly; however, for simplicity this separation is not shown in the code snippets in the following section - see the sample code.

Setting up a Project

Note: Before setting up a project for a client application, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **the Creating the Web Application section on page 14**.

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

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

   - `libacbjnglpeerconnection.jar`
   - `org.apache.http.legacy.jar`
3. Add the following to your `libs` folder:
   - `android-support-v4.jar`
   - `libacbjnglpeerconnection_so.so`

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

Creating the UC Object

To set up all the functionality which the user has access to, the client application must obtain a session ID from the Web Application (see the Creating the Web Application section on page 14), and create a `UC` object using it. You create a `UC` object from a single object, `UCFactory`:

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

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

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

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

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

Note: STUN servers are not necessary if the Gateway is not behind a firewall, so that *Network Address Translation* is not needed. You can provide your own STUN server instead of the public Google one above. You can provide more than one in the array, in which case FCSDK tries them in sequence until it finds a working one.

Adding Voice and Video

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

The application uses the `Phone` object to make or receive calls, represented by `Call` objects. Objects in the API implement the listener pattern which enables an application to be informed of the outcome of operations and other events.

The application can use the `AED` object to create AED Topics (see **the Adding Application Event Distribution section on page 86**).

Making a Call

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

```
Call call;

public void onSessionStarted()
```

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

The `callee` parameter is a `String` containing the address to make the call to.

You can use the values of the `audio` and `video` parameters to make an audio-only or video-only call. Valid values are members of the `MediaDirection` enumeration:

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

The `listener` parameter is an object implementing the `CallListener` interface, which receives notifications of events of interest on the call (such as when it is completely set up).

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

Note: The older form in which `audio` and `video` were boolean parameters is now deprecated.

Receiving a Call

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

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

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

```
        ...
       case IN_CALL:
         // Adjust UI
          ...
         break;
        ...
    }
  }
```

You can use the values of the `audio` and `video` parameters to answer the call as audio-only or video-only. Valid values are members of the `MediaDirection` enumeration:

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

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

Note: The older form in which `audio` and `video` were boolean parameters is now deprecated.

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

Video Views and Preview Views

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

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

```
void onCreate(Bundle savedInstanceState)
{
```

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

  void onStatusChanged(Call call, CallStatus status)
  {
    switch (status)
    {
       ...
      case IN_CALL:
        // Show video
        call.setVideoView(videoSurface);
         ...
        break;
       ...
    }
  }
```

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

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

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

The application can set the camera to use as the local video source on the `Phone` object. See **the Switching between the Front and Back Cameras section on page 84**.

Ending a Call

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

To detect that the remote party has ended the call, the client application must implement the `CallListener` interface in order to receive `onStatusChanged` notifications (see the Monitoring the State of a Call section on page 84).

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

Muting the Local Audio and Video Streams

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

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

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

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

Note: This will affect all calls and persists to subsequent calls. When a call starts, the streams will be muted as per the current `Phone` setting.

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

Holding and Resuming a Call

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

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

Sending DTMF Tones

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

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

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

Handling Multiple Calls

Applications developed with **Fusion Client SDK for Android** do not support multiple simultaneous calls:

Setting Video Resolution

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

Enumerating the Possible Resolutions

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

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

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

The supported resolutions are:

Enumeration Value	Width	Height
RESOLUTION_176x144	176	144
RESOLUTION_352x288	352	288

Enumeration Value	Width	Height
RESOLUTION_640x480	640	480
RESOLUTION_960x720	960	720
RESOLUTION_1280x720	1280	720

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

Setting the Resolution

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

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

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

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

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

Setting the Frame Rate

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

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

Note: The video capture frame rate only applies to the next call made with the `Phone` object; it does not affect calls currently in progress.

Handling Device Rotation

The SDK automatically handles control of the video orientation.

Note: The `setVideoOrientation` method of the `Phone` object is now deprecated.

Switching between the Front and Back Cameras

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

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

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

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

Application Background Mode

When the user presses the **Home** button, presses the power button, or the system launches another application, the foreground application transitions to the inactive state and then to the background state. If you are currently streaming video from your application, this 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 behavior of Android is different to that of iOS.

Monitoring the State of a Call

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

The application can adjust its UI by switching on the value of the CallStatus enumeration, to give the user suitable feedback; for example by playing a local audio file for ringing or alerting:

```
public void onStatusChanged(Call call, CallStatus status)
{
   switch (status)
   {
      case RINGING:
         playRingtone();
         break;
      case IN_CALL:
         stopRinging();
         break;
      case ENDED:
      case BUSY:
      case ERROR:
      case NOT_FOUND:
      case TIMED_OUT:
         updateUIForEndedCall();
         break;
      default:
         break;
   }
}
```

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

The following table gives the possible status codes:

Status Code	Meaning
UNINITIALIZED	The Call object has been created, but not initialized
SETUP	Call is in process of being set up
ALERTING	The call is an incoming one which is alerting (ringing)
RINGING	An outgoing call is ringing at the remote end
MEDIA_PENDING	The call is connected, and waiting for media
IN_CALL	The call is fully set up, including media
BUSY	Dialed number is busy

Status Code	Meaning
NOT_FOUND	Dialed number is unreachable or does not exist
TIMED_OUT	The dialing operation timed out without a response from the dialed number
NO_MB_CAPACITY	The media broker has reached its full capacity
REQUEST_TERMINATED	The request was terminated by the network
TEMPORARILY_UNAVAILABLE	Something necessary to set up the call was unavailable on the network
MEDIA_UNAVAILABLE	Unable to access media
ERROR	The call has errored
ENDED	The call has ended

Adding Application Event Distribution

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

An AED application will:

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

Creating and Connecting to a Topic

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

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

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

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

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

Topic Expiry

There is another version of `createTopic` which takes an expiry time in addition to the topic name and the listener. This allows you to set an expiry time for the topic (in minutes); if the topic is inactive for that amount of time, the server automatically deletes it. A topic created without an expiry time remains on the server indefinitely, until explicitly deleted by a subscriber (see **the Unsubscribing from a Topic section on the next page**).

onTopicConnected

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

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

```
        value = (String)pair.get("value");
        // Display data
      }
    }
```

Unsubscribing from a Topic

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

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

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

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

onTopicDeletedRemotely

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

Publishing Data to a Topic

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

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

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

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

onTopicUpdated

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

Deleting Data from a Topic

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

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

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

Sending a Message to a Topic

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

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

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

onMessageReceived

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

Self-Signed Certificates

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

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

You can obtain the server certificate and CA root certificate through the FAS Administration screens. The ***FAS Administration Guide*** explains how to view and export certificates. You need to obtain the HTTPS Identity Certificate (server certificate) and the Trust Certificate (CA root certificate) that has signed your server certificate.

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

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

Bluetooth Support

An FCSDK application can support Bluetooth devices using the `AudioDeviceManager` class; a single instance of this class is available on the `Phone` object which controls the call. While this is in use, FCSDK will:

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

Using the `AudioDeviceManager`, the application can:

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

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

Starting and Stopping AudioDeviceManager

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

```
AudioDeviceManager adm;

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

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

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

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

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

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

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

Using the Listener

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

1. Bluetooth device
2. Wired headset

3. The device selected by the application (see the Setting the Audio Device section below)
4. The default device (see the Setting the Default Device section on the next page)

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

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

```
AudioDeviceManager adm;

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

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

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

Note: The list of available devices which `AudioDeviceManager` passes to `onDeviceListChanged` will not necessarily contain all the devices which exist on the phone; see the Listing Available Devices section on page 95.

Setting the Audio Device

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

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

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

- `NONE`

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

- `SPEAKER_PHONE`

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

- `WIRED_HEADSET`

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

- `EARPIECE`

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

- `BLUETOOTH`

  Audio is sent to and received from a paired Bluetooth device.

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

Setting the Default Device

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

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

The argument is one of the values from the `AudioDevice` enumeration (see the Setting the Audio Device section on the previous page).

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

Listing Available Devices

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

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

The resulting set contains members of the `AudioDevice` enumeration, taken from the connected devices known to the phone. The list of available devices will not necessarily contain all the devices which exist on the phone. In particular, if it includes `WIRED_HEADSET`, it will not include `EARPIECE` or `SPEAKER_PHONE`, because it considers these to be mutually exclusive. For a phone, the list of available devices may be:

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

A tablet is not likely to have an earpiece.

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

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

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

Responding to Network Issues

As **Fusion Client SDK** is network-based, it is essential that the client application is made aware of any loss of network connection. When it loses a network connection, 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 **Fusion Client SDK** API, and any other available technologies, to handle network failure scenarios.

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

Reacting to Network Loss

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

When all reconnection attempts are exhausted, the `UCListener` receives the `onConnectivityLost` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in the Creating the Session section on page 15.

If any of the reconnection attempts succeeds, the `UCListener` receives the `onConnectionReestablished` callback.

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

Reacting to Network Changes

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

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

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

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

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

Network Quality Callbacks

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

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

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

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

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

Creating an OSX Client Application

Fusion Client SDK enables you to develop OSX applications offering users the following methods of communication:

- Voice calling

Fusion Client SDK provides you with an OSX SDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

To develop OSX applications using **Fusion Client SDK** requires Xcode 4.5 or later.

The **Fusion Client SDK for OSX** is made up of the following classes:

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

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

Setting up a Project

Note: Before setting up a project for a client application, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See the Creating the Web Application section on page 14.

To set up a project including the **Fusion Client SDK**, first create a new project and add OSX native frameworks to it:

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

4. Select the following OSX native dependencies from the OSX folder:

    - `Cocoa.framework`
    - `QuartzCore.framework`

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

Add the **Fusion Client SDK** framework to your project:

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

Initializing the ACBUC Object

The application accesses the API initially via a single object, `ACBUC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see the Creating the Web Application section on page 14), and initialize the `ACBUC` object using it. Once it has received the Session ID, the client application must call the `ucWithCofiguration` method on the `ACBUC`:

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

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

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

Adding Voice

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

Making a Call

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

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

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

- ACBMediaDirectionNone
- ACBMediaDirectionSendOnly
- ACBMediaDirectionReceiveOnly
- ACBMediaDirectionSendAndReceive

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

Receiving a Call

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

```
- (void) phone:(ACBClientPhone*)phone didReceiveCall:(ACBClientCall*)call
```

```
{
  [call answerWithAudio:ACBMediaDirectionSendAndReceive
  video:ACBMediaDirectionNone]
}
```

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

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

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

Note:

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

Video Views and Preview Views

Video is not supported in this release of the OSX **Fusion Client SDK**.

Muting the Local Audio Stream

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

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

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

Holding and Resuming a Call

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

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

DTMF Tones

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

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

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

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

- The second parameter is a boolean which indicates whether the application should play the tone back locally so that the user can hear it.

Handling Multiple Calls

Applications developed with **Fusion Client SDK for OSX** do not support multiple simultaneous calls:

Monitoring the State of a Call

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

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

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

The application can adjust the UI by switching on the value of the `status` parameter, to give the user suitable feedback, for example by playing a local audio file for ringing or alerting:

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

```
            break;
        }
    }
```

The following table gives the possible status codes:

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

Threading

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

Self-Signed Certificates

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

You can obtain the server certificate and CA root certificate through the FAS Administration screens. The ***FAS Administration Guide*** explains how to view and export certificates. You need to extract the HTTPS Identity Certificate (server certificate) and the Trust Certificate (CA root certificate) that has signed your server certificate.

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

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

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

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

Note: Since iOS 9, you also need to add a setting to your application's `plist` file to allow connection to a server using self-signed certificates. Set **Allow Arbitrary Loads** under **App Transport Security Settings** to `YES`.

Responding to Network Issues

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

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

Reacting to Network Loss

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

When all reconnection attempts are exhausted, the ACBUCDelegate receives the ucDidLoseConnection callback, and the retries stop. At this point the client application should assume that the session is invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in the Creating the Session section on page 15.

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

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

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

Reacting to Network Changes

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

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

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

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

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

Creating a Windows Client Application

Fusion Client SDK enables you to develop Windows applications that offer users the ability to make voice and video calls.

Fusion Client SDK provides you with a Windows SDK and a network infrastructure which integrate seamlessly with your existing SIP infrastructure.

The **Fusion Client SDK for Windows** is made up of the following main classes:

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

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

Setting up a Project

Note: Before setting up a project for a client application, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See [the Creating the Web Application section on page 14].

To set up a project including the **Fusion Client SDK**, you first need to create a new project and add SDK native libraries and headers:

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

(from step 1) to the **Additional Include Directories**,

7. In the *Linker->Input* section, add the `csdk-sample\target\lib\fcsdk\Release\ClientSDKWin.lib` library (from step 1) to the **Additional Dependencies**.

Initializing the UC and Starting the Session

The application initially accesses the API via a single object, `UC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see **the Creating the Web Application section on page 14**), and initialize the `UC` object using it. Once it has received the Session ID, the client application must create the `UC` object, then call its `StartSession` method:

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

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

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

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

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

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

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

The failure notification is `OnSessionNotStarted()` .

Adding Voice and Video

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

Adding a Preview Window before a Call is made

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

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

Making a Call

Once you have created the session, you can start making calls. Create a `Call` object from the `Phone` object :

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

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

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

The `CreateCall` method returns a `Call` object immediately, but the application should not use it until the `CallListener` receives the asynchronous notification that the call creation has succeeded.

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

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

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

Receiving a Call

FCSDK invokes the `PhoneListener` object that you registered with the `Phone` object when it receives an incoming call:

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

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

The two parameters to the `Answer()` method indicate whether to answer the call with audio and video respectively. They are members of the `MediaDirection` enumeration:

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

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

Displaying Video

Your application can display both video received from the remote peer, and a preview of the locally captured video that it sends to the remote peer. You display video in your application using two classes provided by the Windows SDK: `VideoView` and `ImagePipeServer`.

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

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

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

The first line creates a `VideoView` with a unique name for the remote view. The second line sets a refresh function for the view; this refresh function is invoked every time there is a new video frame to render. The application should implement it to invalidate the user interface element that displays the video. The third line creates an instance of the subclass of `ImagePipeServer`, giving it the name and reference of the `VideoView`. The fourth line tells the `Call` object the name of the `VideoView` to use for displaying the remote video.

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

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

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

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

Muting Local Audio and Video Streams

Muting the local streams 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);
}
```

You can also mute the audio and video streams as soon as the call is answered by supplying `RECEIVE_ONLY` as the media direction:

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

Holding and Resuming a Call

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

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

Sending DTMF Tones

Your application can send DTMF tones on a call:

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

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

The boolean parameter indicates whether or not you want the tones to also be played back locally, so that the user of your application can hear them.

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

Handling Multiple Calls

Applications developed with the **Fusion Client SDK for Windows** do not support multiple simultaneous calls.

Setting Video Resolution

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

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

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

You can also set the frame rate:

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

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

Monitoring the State of a Call

During call setup, the call transitions through several states, from the initial setup to being connected with media available (or failure). You can monitor these states using the `onStatusChanged` method of `CallListener`. Switching on the value of the `CallStatus` enumeration allows the application to adjust the UI to give the user suitable feedback, such as by playing a local audio file for ringing or alerting:

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

```
}
```

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

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

The `CallListener` interface defines other notification functions that allow your application to detect call quality changes and dial and call failures (see the Network Quality Callbacks section on page 119).

Adding Application Event Distribution

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

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

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

Initialization of the `UC` object is exactly the same as for voice and video calling (see the Initializing the UC and Starting the Session section on page 108).

Creating a Topic

Once you have obtained an `AED` object, you can create a topic.

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

or

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

The `name` is a `std::string` which uniquely identifies the topic on the server, and the `expiry` is a time in minutes. If you create the topic with an expiry time, it will be automatically removed from the server after it has been inactive for that time. When created without an expiry time (by the first method), the topic exists indefinitely, and needs to be deleted explicitly (see the Disconnecting from a Topic section on page 117).

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

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

OnTopicConnected

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

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

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

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

```
    } while (it.Next());
  }
```

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

Publishing Data to a Topic

Once the client application has 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 will also be an `OnTopicUpdated` callback when the data is sent to all clients connected to the topic. The `OnTopicSubmitted` callback includes a `version` parameter, enabling clients to know whether any `OnTopicUpdate` callback they receive 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 client receives the `OnTopicUpdated` callback when any client makes a change to a data item on a topic (adding, deleting, or changing the value associated with it), and contains information about the change:

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

The `topic`, `name`, and `value` parameters are as detailed previously (the `value` parameter is the new value); `deleted` will be `true` if the data item has been removed from the topic (see **the Deleting Data from a Topic section on the next page** for details about deleting data). The `version` parameter is an integer which increases with every change made to the value of the data item. The client application can 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 client successfully sends the message, the listener receives an `OnTopicSent` followed by an `OnMessageReceived` callback, both containing the `topic` and the `message`; if it is not successful, the client will receive an `OnTopicNotSent` callback containing the `topic`, the `message` which failed, and an error message.

OnMessageReceived

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

Disconnecting from a Topic

The client application can disconnect from a topic:

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

or delete it altogether:

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

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

OnTopicDeletedRemotely

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

Responding to Network Issues

As **Fusion Client SDK** 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 **Fusion Client SDK** API, and any other available technologies, to handle network failure scenarios.

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

Reacting to Network Loss

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

When all reconnection attempts are exhausted, the `UCListener` receives the `OnConnectionLost` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in the Creating the Session section on page 15.

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

Of these three notifications, the application only needs to implement `OnConnectionLost()`. The other two are optional.

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

Network Quality Callbacks

The application can implement the `OnInboundQualityChanged` method of the `CallListener` to receive notification of changes in the quality of the network during a call:

```
void MyCallListener::OnInboundQualityChanged(int inboundQuality)
{
  // Show indication of quality
}
```

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

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

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

Note: The `CallListener` class provides a skeleton implementation (which does nothing) of `OnInboundQualityChanged`, so there is no need for the application to implement it.

Creating a Windows .NET Client Application

Fusion Client SDK includes a wrapper for the native Windows library that allows you to develop Windows applications in languages that use the .NET Framework, such as C# or VB.NET.

The wrapper generally follows the native Windows SDK library, provides similar functionality, and works with similar objects. It departs from the FCSDK for Windows where doing so works better with the .NET Framework. Managed versions of the classes provided by the wrapper are contained within the namespace `FCSDKCLR` (FCSDK Common Language Runtime) and are prefixed by CLI_ to indicate that they use the Common Language Infrastructure.

Interfaces are prefixed by ICLI_ in keeping with conventional naming of interfaces used by the .NET framework.

The examples are expressed in the C# language.

Setting up a Project

Note: Before setting up a project for a client application, make sure you have created a Web Application to authenticate and authorize users, and to create and destroy sessions for them. See **the Creating the Web Application section on page 14**.

This section provides guidance on setting up a project using Visual Studio 2013 that imports the CSDK-CLR wrapper library.

1. Create a new .NET project (typically a Visual C# Windows Forms application or a Visual Basic Windows Forms application).
2. Within Solution Explorer (use the View menu to display it if it is not shown), open the project node, then open its **References** node.
3. Right-click on the **References** node and click on **Add reference...**.
4. The SDK contains a file `CSDK-CLR.dll`. Click on the **Browse...** button, browse to the file and click on **Add**.

Initializing the CLI_UC and Starting the Session

The application initially accesses the API through a single object, `CLI_UC`. To set up all the functionality to which the user has access, the application needs to obtain a session ID from the Web Application (see the Creating the Web Application section on page 14), and initialize the `CLI_UC` object using it. Once it has received the Session ID, the client application must create the `CLI_UC` object, then call its `StartSession` method:

```
public sealed class UCOwner : FCSDKCLR.ICLI_UCListener
{
  private FCSDKCLR.CLI_UC mUC;
  public void MakeTheCliUcObject(String sessionID, String stunServers)
  {
    mUC = new FCSDKCLR.CLI_UC(sessionID, stunServers, this);
    mUC.StartSession();
  }
  /// The class needs to implement ICLI_UCListener
  /// so that it can act upon its callbacks.
}
```

If you use a different object that implements `ICLI_UCListener` to provide the callbacks, then you should substitute a reference to that object for `this` in the call to the `CLI_UC` constructor.

Note that you must dispose of the `CLI_UC` object, in common with many of the objects provided by the CLI wrapper, when it is no longer useful. Do this by calling its `Dispose` method:

```
mUC.Dispose();
```

The object that implements the `ICLI_UCListener` interface must provide implementations for the call-backs:

```
void OnSessionStarted();
void OnSessionNotStarted();
```

The SDK calls one of these two functions to indicate whether starting the session has succeeded or failed. The other function callbacks in the interface are self-explanatory (for their use, see the Responding to Network Issues section on page 131):

```
void OnConnectionRetry(byte attemptNumber, ushort delayInSeconds);
void onConnectivityLost();
void OnConnectionReestablished();
void OnUnknownWebsocketMessage(string s);
```

Adding Voice and Video

Once the application has created the `CLI_UC` object, it can retrieve the `CLI_Phone` object from it, and use it to make or receive calls, which are represented by `CLI_Call` objects. Each of the `CLI_UC`, `CLI_Phone` and `CLI_Call` objects have corresponding listener interfaces that the application can implement in order to receive error notifications and other events.

Adding a Preview Window before a Call is made

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

```
public void OnSessionStarted()
{
  mUC.GetPhone().SetPreviewViewName("local");
};
```

Making a Call

Once you have created the session, you can start making calls. Create a `CLI_Call` object from the `CLI_Phone` object:

```
CLI_Call call = mUC.GetPhone().CreateCall("1234",
  CLI_MediaDirection.SEND_AND_RECEIVE,
  CLI_MediaDirection.SEND_AND_RECEIVE, this);
```

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

Note: There is an alternative version of `CreateCall`, which does not take the final `ICLI_CallListener` parameter. If you use this version, you will need to call `SetListener` on the resulting `CLI_Call` object in order to receive notifications about the call.

The `CreateCall` function returns the newly created call as a `CLI_Call` object, but the application should not use it until the `ICLI_CallListener` receives an asynchronous notification indicating that the call creation has succeeded.

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

- NONE
- SEND_ONLY
- RECEIVE_ONLY
- SEND_AND_RECEIVE

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

Receiving a Call

FCSDK will invoke the `ICLI_PhoneListener` object that you registered with the `CLI_Phone` object when an incoming call is received:

```
public void OnIncomingCall(CLI_Call call)
{
   call.Answer(MediaDirection.SEND_AND_RECEIVE, MediaDirection.SEND_AND_
   RECEIVE);
}
```

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

The two parameters to the `Answer` method indicate whether to answer the call with audio and video respectively. They are members of the `CLI_MediaDirection` enumeration:

- NONE
- SEND_ONLY
- RECEIVE_ONLY
- SEND_AND_RECEIVE

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

Displaying Video

Your application can display both video received from the remote peer, and a preview of the locally-captured video that it sends to the remote peer.

The SDK uses named pipes internally to route frame information from a receiving (producer) thread to a consuming thread, These activities are separated within the SDK, and you need to connect them to allow video to be displayed.

The `CLI_Phone` object has a method, `SetPreviewViewName` (associated with local video), and the `CLI_Call` object has a method, `SetVideoViewName` (associated with remote video). Each method takes a string which is used to create the name of the pipe. File naming conventions should be followed when assigning this name. 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 must override the method `SendImageToView`, which returns `void` and takes no arguments. FCSDK calls it whenever it receives a new frame from the pipe and makes it available for painting. The overridden method must obtain the frame and render it.

To obtain the frame, call the method `GetImageData`, which returns a `CLI_VideoViewImageData` object that contains the image data. You can render the frame onto a `CLI_VideoView` object. The `CLI_VideoView` has a method, `SetImageData`, that takes the `CLI_VideoViewImageData` object that came from `GetImageData` as its argument; it associates the image data with the `CLIVideoView` object. It also has a method, `Paint`, which renders the associated frame data - the window's `onPaint` method should call it, passing in its `PaintEventArgs` object.

When each frame is ready to display, the `CLI_VideoView` calls an `ICLI_ViewRefresher` object's `RefreshViewFunc`. You can associate an `ICLI_ViewRefresher` with the `CLI_VideoView` by calling its `SetRefreshViewFunc` method. The usual behavior of the `RefreshViewFunc` method is to invalidate the region of the client area which renders the video.

Muting Local Audio and Video Streams

Muting the local streams stops audio or video from being sent to the remote party. Audio and video received from the remote party is not affected.

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

```
public void MyUIController.MuteButtonPressed()
{
   call.EnableLocalAudio(false);
   call.EnableLocalVideo(false);
}
```

You can also mute the audio and video streams as soon as the call is answered by supplying RECEIVE_ONLY as the media direction:

```
call.Answer(CLI_MediaDirection.RECEIVE_ONLY, CLI_MediaDirection.RECEIVE_
ONLY);
```

Holding and Resuming a Call

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

```
public void MyUIController.HoldButtonPressed()
{
  call.Hold();
}
public void MyUIController.ResumeButtonPressed()
{
  call.Resume();
}
```

Sending DTMF Tones

Your application can send DTMF tones on a call by using the SendDTMF method provided by the CLI_Call object. It can be used as follows:

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

The example sends five tones sequentially. To send a single tone, use a single-character string.

The boolean parameter indicates whether you want the tones to also be played locally, so that the user of your application can hear them.

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

Handling Multiple Calls

Applications developed with the **Fusion Client SDK for .NET** do not support multiple simultaneous calls.

Setting Video Resolution

Your application can configure both the resolution and frame rate of the video it sends to the remote party in a video call. You can set the video resolution on the CLI_Phone object:

```
phone.SetPreferredCaptureResolution(CLI_VideoCaptureResolution.RESOLUTION_
1280x720);
```

Available options at the time of writing are:

- `RESOLUTION_AUTO`
- `RESOLUTION_352x288`
- `RESOLUTION_640x480`
- `RESOLUTION_1280x720`

You can also set the frame rate:

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

Note: Changes to the video resolution and frame rate apply only to new calls. Calling these APIs while a call is in progress has no effect.

Monitoring the State of a Call

During call setup, the call transitions through several states, from the initial setup to being connected with media available (or failure). You can monitor these states using the `ICLI_CallListener` object using the method `OnStatusChanged(CLI_CallStatus newStatus)`. `CLI_CallStatus` is an enumeration with the following values:

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

Status Code	Meaning
	ation is not available.
ENDED	The call has ended

There are also other callback functions that allow your application to detect call quality changes (`OnInboundQualityChange` which provides a number from 0 to 100 representing the call quality; 100 is best) and dial or call failures (`OnDialFailed` and `OnCallFailed`).

Adding Application Event Distribution

Application Event Distribution is contained in the `CLI_AED` and `CLI_Topic` objects. Asynchronous notification of events occurs by callback to any class that implements the `ICLI_TopicListener` interface.

In order to use Application Event Distribution, you must first obtain a `CLI_AED` object from the `CLI_UC` object:

```
CLI_AED aed = uc.GetAED();
```

Initialization of the `CLI_UC` object is exactly the same as for voice and video calling (see the Initializing the CLI_UC and Starting the Session section on page 121).

Creating a Topic

You can create a topic using the `CreateTopic` API provided by the `CLI_AED` object:

```
aed.CreateTopic(name, listener);
```

or

```
aed.CreateTopic(name, expiry, listener);
```

The `name` argument is a string that uniquely identifies the topic on the server. If you use the form of the API that includes an `expiry` argument, the expiry period is in minutes. If the topic has an expiry time, it will be removed from the server after it has been inactive for that period of time. When created without an expiry time (by the first method), the topic exists indefinitely, and the application must delete it explicitly (see the Disconnecting from a Topic section on page 130).

The `listener` is an object that implements `ICLI_TopicListener`, the methods of which provide notification to your application of events that concern the topic.

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

OnTopicConnected

After creating a topic, the listener should receive an `OnTopicConnected` callback (in case of failure, the listener receives an `OnTopicNotConnected` callback instead). The arguments to the callback are a `CLI_Topic` identifying the topic and a `CLI_TopicData` that contains all existing data for the topic.

The data contained within a topic consists of key-value pairs. If you know the name of a key for which you want the associated value, you can retrieve it as follows:

```
CLI_TopicDataValue value = topicData.GetValue(name);
```

where the `name` argument is a string containing the key name.

You can also iterate through the keys with a simple `foreach` loop:

```
foreach (CLI_TopicDataElement element in topicData)
{
    string key = element.key;
    CLI_TopicDataValue value = element.value;
    int version = element.version;
    bool deleted = element.deleted;
}
```

A method that is probably most useful during testing and debugging is `CLI_TopicDataElement.ToString` which in the case of `CLI_TopicDataElement` objects is overridden to provide a readable representation of the object state.

`CLI_TopicDataValue` wraps an `acb::TopicDataValue` object. It contains at most one value that is a `string`, a `double`, an `int`, or a `bool`. If it doesn't contain a value, it represents an empty value. Note that only `string` values can be set using the current API.

The type is returned in string form by using `CLI_TopicDataValue.GetType()` which returns a `string` identifying the type. You can obtain the value using `GetAsString(defaultValue)`, `GetAsBool()`, `GetAsInt(defaultValue)` or `GetAsDouble(defaultValue)` and if the value does not exist, the default value is returned (or `false` in the case of `GetAsBool()`). You can also call `GetAsInt()` or `GetAsDouble()` in which case the default value returned is 0.

Publishing Data to a Topic

Once the client application has 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 **the OnTopicUpdated section below** for details.

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

OnTopicUpdated

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

```
OnTopicUpdated(CLI_Topic topic, string key, string value, int version, bool deleted);
```

The `topic`, `name`, and `value` parameters are as detailed previously. The `version` parameter enables clients to know, when they receive this callback, whether it refers to data they have just submitted (if it does, the `version` parameters are equal). For a newly-created data item, the version is 0, and it is incremented upon every change.

Deleting Data from a Topic

The client can delete a key-value pair from a topic by calling `CLI_Topic.DeleteData(“keyName”)`. The client receives either an `OnDataDeleted` callback followed by an `OnTopicUpdated` callback, or an `OnDataNotDeleted` callback (in the case of failure).

Sending a Message to a Topic

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

```
topic.SendMessage(“message”);
```

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

OnMessageReceived

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

Disconnecting from a Topic

The client application can disconnect from a topic:

```
topic.Disconnect();
```

or delete it altogether:

```
topic.Disconnect(true);
```

If the `Disconnect` method is called with the boolean argument set to `true`, the topic is deleted from the server. Calling `Disconnect` without an argument is equivalent to calling `Disconnect(false)`, and in either case the topic is not deleted.

If the application attempts to delete the topic from the server, the listener receives an `OnTopicDeleted` callback followed by either an `OnTopicDeletedRemotely` callback if successful, or an `OnTopicNotDeleted` callback otherwise. `OnTopicDeleted` includes a message string as one of its arguments, which is not likely to be useful. `OnTopicNotDeleted` includes a message string that provides information about the error.

OnTopicDeletedRemotely

`OnTopicDeletedRemotely` is received by all clients connected to the topic when it is deleted from the server, either as a result of any client calling `Disconnect(true)`, or because it expires. It passes the `CLI_Topic` which has been deleted. Once a topic has been deleted, the client should not call any of the topic methods and should consider itself unsubscribed from that topic. If a topic with the same name is subsequently created, it is a new topic, and the client is not automatically subscribed to it.

Responding to Network Issues

As **Fusion Client SDK** 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 **Fusion Client SDK** API, and any other available technologies, to handle network failure scenarios.

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

Reacting to Network Loss

In the event of network connection problems, the SDK automatically tries to re-establish the connection. It will make seven attempts at the following intervals: 0.5s, 1s, 2s, 4s, 4s, 4s, 4s. A call to `OnConnectionRetry` on the `ICLI_UCListener` precedes each of these attempts. The callback supplies the attempt number (as a `System::Byte`) and the delay in seconds before the next attempt (as a `System::UInt16`) in its two parameters.

When all reconnection attempts are exhausted, the `ICLI_UCListener` receives the `OnConnectionLost` callback, and the retries stop. At this point the client application should assume that the session is now invalid. The client application should then log out of the server and reconnect via the web app to get a new session, as described in the Creating the Session section on page 15.

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

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

Network Quality Callbacks

The application can implement the `OnInboundQualityChange` method of the `ICLI_CallListener` interface to receive callbacks on the quality of the network during a call:

```
public void OnInboundQualityChange(int inboundQuality)
{
  // Show indication of quality
}
```

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

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

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

Appendix: Error Codes

Some of the APIs give access to an error code from the Gateway. These are:

Code	Meaning
`NOMATCH`	The `Offer`, `Answer`, or `Offer Request` is not for a new session and does not correspond to an existing session
`REFUSED`	The initial `Offer` was refused
`BUSY`	There is already an `Offer` or `Offer Request` pending, so this `Offer` cannot be handled
`TIMEOUT`	The outbound request failed because of a SIP timeout
`NOTFOUND`	The outbound request failed because the callee could not be found
`FAILED`	The request failed for some other reason.