Kurento-iOS Documentation

Release latest

Jul 26, 2017

Contents

1 Overview **1**
- 1.1 System Requirements . . . 1

2 Installation guide **3**
- 2.1 Installation with CocoaPods . . . 3
- 2.2 Manual installation . . . 4

3 Developers Guide **7**
- 3.1 Usage . . . 7
- 3.2 Documentation . . . 13

4 Architecture **15**

CHAPTER 1

Overview

Kurento Toolbox for iOS provides a set of basic components that have been found useful during the native development of the WebRTC applications with Kurento.

It contains:

- a JSON-RPC 2.0 client over websocket; it draws inspiration from javascript implementation found in kurento-jsonrpc-js.
- a WebRTCPeer manager aimed to simplify WebRTC interactions (i.e. handle multiple peer connections, manage SDP offer-answer dance, retrieve local or remote streams ...). This class is implemented based on the native WebRTC library libjingle_peerconnection packaged as a CocoaPod.

System Requirements

The KurentoToolbox supports the next:

- iOS 8+
- Armv7, Arm64 architectures

No support for iOS 32/64 bit simulator (no access to camera yet)

CHAPTER 2

Installation guide

Installation with CocoaPods

CocoaPods is a dependency manager for Objective-C, which automates and simplifies the process of using 3rd-party frameworks or libraries like KurentoToolbox.framework in your projects.

Step 1: Downloading CocoaPods

CocoaPods is distributed as a Ruby gem and is installable with the default Ruby available on OSX by running the following command:

```
$ sudo gem install cocoapods
$ pod setup
```

Step 2: Creating a Podfile

Project dependencies to be managed by CocoaPods are specified in a file called `Podfile`. Create this file in the same directory as your Xcode project (`.xcodeproj`) file:

```
$ touch Podfile
$ open -a Xcode Podfile
```

You just created the pod file and opened it using Xcode, to add some content to the empty pod file copy and paste the following lines:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '8.0'
pod 'KurentoToolbox', '~> 0.2.4'
```

Step 3: Installing Dependencies

Now you can install the dependencies in your project:

```
$ pod install
```

From now on, be sure to always open the generated Xcode workspace (`.xcworkspace`) instead of the project file when building your project:

```
$ open <YourProjectName>.xcworkspace
```

At this point, everything is in place for you to start using KurentoToolbox. Just **:obj-c:'#import'** the umbrella header wherever you need to use it, usually in <YourProjectName-Prefix>.pch file:

```
#import <KurentoToolbox/KurentoToolbox.h>
```

Manual installation

Step 1: Download & unzip the framework

Download the latest KurentoToolbox.framework (v0.2.4) below:

KurentoToolbox.framework

Step 2: Add the framework to your Xcode project

Drag the KurentoToolbox.framework into your Xcode project. Make sure the "Copy items to destination's group folder" checkbox is checked.

Step 3: Link Binary With Library Frameworks

Click on Project → Select Target of interest → Choose Build Phases tab → Link Binary With Libraries → At the bottom of this list hit + to add libraries.

Here is the list of required Apple library frameworks/dylibs:

- `libicucore.dylib`
- `libstdc++.dylib`
- `libc.dylib`
- `libsqlite3.dylib`
- `AVFoundation.framework`
- `AudioToolbox.framework`
- `CoreGraphics.framework`
- `CoreMedia.framework`
- `GLKit.framework`
- `UIKit.framework`
- `VideoToolbox.framework`

- `CFNetwork.framework`
- `Security.framework`

Step 4: Add -ObjC linker flag

Click on Project → Select Target of interest → Choose Build Settings tab → Other Linker Flags and add the `-ObjC` linker flag.

Step 5: Importing Header(s)

Just **:obj-c:'#import'** the umbrella header wherever you need to use KurentoToolbox, usually in <YourProjectName-Prefix>.pch file

```
#import <KurentoToolbox/KurentoToolbox.h>
```

CHAPTER 3

Developers Guide

Usage

WebRTC

Initialization

`NBMWebRTCPeer` is the main component used to setup a WebRTC media session, it must be initialized with a delegate (`NBMWebRTCPeerDelegate`) and a media configuration object (`NBMMediaConfiguration`):

```
/*
Default media constraints:

Audio codec: Opus audio codec (higher quality)
Audio bandiwidth limit: none
Video codec: Software (VP8)
Video renderer: OpenGLES 2.0
Video bandwidth limit: none
Video format: 640 x 480 @ 30fps
*/
NBMMediaConfiguration *mediaConfig = [NBMMediaConfiguration defaultConfiguration];
//If necessary modify media configuration and use it to setup WebRTCPeer, 'nil' configuration means default values
NBMWebRTCPeer *webRTCPeer = [[NBMWebrTCPeer alloc] initWithDelegate:self configuration:mediaConfig];
```

SDP negotiation #1 : Generate Offer

An Offer SDP (Session Description Protocol) is metadata that describes to the other peer the format to expect (video, formats, codecs, encryption, resolution, size, etc). An exchange requires an offer from a peer, then the other peer must receive the offer and provide back an answer:

```
//Create a new offer for connection with specified identifier
[webRTCPeer generateOffer:@"connectionId"];
```

When the offer is generated, the `[webRTCPeer:didGenerateOffer:forConnection:]` message is sent to webRTCPeer's delegate:

```
#pragma mark -
#pragma mark NBMWebRTCPeerDelegate

//Handle SDP offer generation
- (void)webRTCPeer:(NBMWebRTCPeer *)peer didGenerateOffer:(RTCSessionDescription
↪*)sdpOffer forConnection:(NBMPeerConnection*)connection {
//TODO: Signal SDP offer for connection
}
```

SDP negotiation #2 : Process Answer

An Answer SDP is just like an offer but a response, we can only process an answer once we have generated an offer:

```
//Process an answer from connection with specified identifier
[webRTCPeer processAnswer:@"sdpAnswer" connectionId:@"connectionId"];
```

Tricke ICE #1 : Gathering local candidates

After creating the peer connection an event will be fired each time the ICE framework has found some local candidates, a `[webRTCPeer:hasICECandidate:forConnection:]` message is sent to webRTCPeer's delegate:

```
#pragma mark -
#pragma mark NBMWebRTCPeerDelegate

//Handle the gathering of ICE candidate for connection with specified identifier
- (void)webRTCPeer:(NBMWebRTCPeer *)peer hasICECandidate:(RTCICECandidate *)candidate
↪forConnection:(NBMPeerConnection *)connection {
//TODO: Signal ICE candidate for connection
}
```

Tricke ICE #2 : Set remote candidates

When a new remote ICE candidate is received it is necessary to process it properly to achieve a successful media negotiation:

```
//Set remote candidate for connection with specified identifier
[webRTCPeer addICECandidate:candidate connectionId:@"connectionId"];
```

JSON-RPC

Initialization

The `NBMJSONRPCClient` is responsible to establish client sessions to JSON-RPC 2.0 servers, it sends JSON-RPC 2.0 requests and receives the corresponding responses. Sending (and receiving) of JSON-RPC 2.0 notifications is also

supported. Messages are transported using WebSocket connection. It must be initialized with a server URL, a delegate (`NBMJSONRPCDelegate`) and, optionally, a configuration object (`NBMJSONRPCClientConfiguration`):

```
/*
Default client configuration:

Request timeout: 5 sec.
Timeout request retries: 1
Connect after initialization: YES
*/
NBMJSONRPCClientConfiguration *clientConfig = [NBMJSONRPCClientConfiguration␣
↪defaultConfiguration];
//WebSocket URI
NSURL *wsURI = [NSURL URLWithString:@"http://localhost:8080/json-rpc"];
//If necessary modify client configuration, 'nil' configuration means default values
NBMJSONRPCClient *jsonRpcClient = [[NBMJSONRPCClient alloc] initWithURL:wsURI␣
↪configuration:clientConfig delegate:self];
```

Using the default configuration (`autoConnect` property is set to YES) the client connects automatically after initialization, otherwise, before starting to send requests is necessary to call `[connect]` and wait the delegate `[clientDidConnect:]` message sent when the WebSocket connection was established successfully or key-value observe `connected` property. If the WebSocket initialization failed or, in any case, when a connection error occurred, the `[client:didFailWithError:]` message is sent to the client's delegate.

Send Request message

Call `[sendRequestWithMethod:parameters:completion]` to send a JSON-RPC 2.0 request and obtain the corresponding response:

```
//Use NSArray for positional params
NSArray *positionalParams = @[@"param1", @"param2", @(1234), @YES];
//Use NSDictionary for named params
NSDictionary *namedParams = @{@"param1": @"value1", @"param2": @"value2", @"number":␣
↪@(1234), @"boolean": @YES};

NSString *method = @"methodName";

[jsonRpcClient sendRequestWithMethod:method parameters:namedParams completion:^
↪(NBMResponse *response) {
    //If no response is returned, request is gone on timeout
    if (!response) {
        NSLog(@"Request with method %@ is gone on timeout!", method);
    }

    //Evaluate the response

    //If has a response error
    NBMResponseError *responseError = response.error;
    if (responseError) {
        NSLog(@"Response error: %@", responseError);
    } else {
        //Response has a result
        NSLog(@"Response result: %@", response.result);
    }
}];
```

Send Notification message

Call `[sendNotificationWithMethod:parameters:nil]` to send a JSON-RPC 2.0 notification (no response is returned):

```
NSString *method = @"methodName";

//Send a notification with no parameters
[jsonRpcClient sendNotificationWithMethod:method parameters:nil];
```

Receive Requests/Notification

When a request or, usually, a notification is received by the client, the `[client:didReceiveRequest:]` method is called on delegate:

```
#pragma mark -
#pragma mark NBMJSONRPCClientDelegate

- (void)client:(NBMJSONRPCClient *)client didReceiveRequest:(NBMRequest *)request {
    // Handle request/notification
    NSLog(@"Request received: %@", request);
}
```

Kurento Room

Initialization

`NBMRoomClient` is the main class that communicates with Kurento Room server using WebSocket API, the exchanged messages between server and client are JSON-RPC 2.0 requests and responses. To use it you must first create a `NBMRoom` object (providing a username, a room name and the server's URI for listening JSON-RPC requests) and a `NBMRoomClientDelegate` object to be informed about room's events. At the moment the client supports only one room:

```
//NBMRoom

//Local peer's identifier
NSString *username = ...
//Room name
NSString *roomName = ...
//WebSocket URI
NSURL *wsURI = [NSURL URLWithString:@"http://localhost:8080/room"];
NBMRoom *room = [[NBMRoom alloc] initWithUsername:username roomName:roomName␣
↪roomURL:wsURI];

//NBMRoomClient

//NBMRoomClient with default timeout (5 sec)
NBMRoomClient *roomClient = [[NBMRoomClient alloc] initWithRoom:room delegate:self];

//Or

//NBMRoomClient with custom timeout (10 sec)
NSTimeInterval clientTimeout = 10;
NBMRoomClient *roomClient = [[NBMRoomClient alloc] initWithRoom:room␣
↪timeout:clientTimeout delegate:self];
```

Once intialized, before start calling client APIs we have to call `[connect]` and wait the delegate `[client:isConnected:]` message sent when the WebSocket connection was established successfully or key-value observe `connected` property:

```
[roomClient connect];

#pragma mark -
#pragma mark NBMRoomClientDelegate

- (void)client:(NBMRoomClient *)client isConnected:(BOOL)connected {
    if (connected) {
        //TODO: Start using APIs, eg. "joinRoom"
        [client joinRoom];
    } else {
        //TODO: Handle client disconnection, eg. try to reconnect
        [client connect];
    }
}
```

If the WebSocket initialization failed or, in any case, when a connection error occurred, the `[client:didFailWithError:]` message is sent to the client's delegate.

Room APIs

The client provides two different types of signatures of its asynchronous API, these differ in the way they handle callbacks:

- the first type implements callbacks sending messages to the client's delegate
- the second type uses blocks as method parameters (no message is sent to delegate)

Join Room

Call `[joinRoom]` or `[joinRoom:]` method to join the room, once joined you can see the list of other `NBMPeers` partecipants:

```
[roomClient joinRoom]

#pragma mark -
#pragma mark NBMRoomClientDelegate

- (void)client:(NBMRoomClient *)client didJoinRoom:(NSError *)error {
    if (!error) {
        NSLog(@"Partecipants %@", client.peers);
    } else {
        NSLog(@"Join room error: %@", error);
    }
}

//Or

[roomClient joinRoom:^(NSDictionary *peers, NSError *error) {
    if (!error) {
        NSLog(@"Partecipants %@", [roomClient.peers]);
    } else {
        NSLog(@"Join room error: %@", error);
```

```
    }
}];
```

Publish video

Call `[publishVideo:loopback:]` or `[publishVideo:loopback:completion]` method to start streaming local media to anyone inside the room. The user should pass the SDP offer generated by `NBMWebRTCPeer` in response of `[generateOffer:]` call, obtaining the SDP answer required to display local media after having passed through the KMS server:

```
//Retrieve local peer identifier (username)
NBMRoom *room = roomClient.room;
NSString *localPeerId = room.localPeer.username;

//Generate SDP offer for local peer connection
[webRTCPeer generateOffer:localPeerId];

#pragma mark -
#pragma mark NBMWebRTCDelegate

- (void)webRTCPeer:(NBMWebRTCPeer *)peer didGenerateOffer:(RTCSessionDescription␣
↪*)sdpOffer forConnection:(NBMPeerConnection*)connection {
    //Connection is related to local peer
    if ([localPeerId isEqualToString:connection.connectionId]) {
            //Publish video with looback enabled
            [roomClient publishVideo:sdpOffer loopback:YES completion:^(NSString␣
↪*sdpAnswer, NSError *error) {
                if (sdpAnswer) {
                    [webRTCPeer processAnswer:sdpAnswer connectionId:connection.
↪connectionId];
                }
            }];
        }
}
```

Receive video

Call `[receiveVideoFromPeer:offer:]` or `[receiveVideoFromPeer:offer:completion]` method to receive media from peers in the room that published their media:

```
//Retrieve a partecipant
NBMPeer *aPeer = [roomClient peerWithIdentifier:@"peerId"];

//Verify that peer has joined and has already published its media
if (aPeer.streams.count > 0) {
    //Receive video from peer
    NSString *sdpOffer = ... //A generated SDP Offer by NBMWebRTCPeer
    [roomClient receiveVideoFromPeer:aPeer offer:sdpOffer];
}

#pragma mark -
#pragma mark NBMRoomClientDelegate

- (void)client:(NBMRoomClient *)client didReceiveVideoFrom:(NBMPeer *)peer␣
↪sdpAnswer:(NSString *)sdpAnswer error:(NSError *)error {
    if (sdpAnswer) {
            [webRTCPeer processAnswer:sdpAnswer connectionId:connection.connectionId];
        }
```

```
}
```

Send ICE candidate

Call `[sendICECandidate:forPeer]` or `[sendICECandidate:forPeer:completion]` method when an ICE candidate is gathered on the client side by a peer connection:

```
#pragma mark -
#pragma mark NBMWebRTCDelegate

- (void)webRTCPeer:(NBMWebRTCPeer *)peer hasICECandidate:(RTCICECandidate *)candidate
↪forConnection:(NBMPeerConnection *)connection    {
    //Find peer using connection's identifier
    NBMPeer *peer = [roomClient peerWithIdentifier:connection.connectionId];

    //Send ICE candidate
    [roomClient sendICECandidate:candidate forPeer:peer completion:^(NSError *error) {
        if (!error) {NSLog(@"ICE candidate sent for peer: %@", peer.identifier);
        }
    }]
}
```

Kurento Tree

Initialization

`NBMTreeClient` is the main class that communicates with Kurento Tree server using WebSocket API, the exchanged messages between server and client are JSON-RPC 2.0 requests and responses.

...

Documentation

Link to API Reference (Apple style).

CHAPTER 4

Architecture

- WebRTC
  - NBMWebRTCPeer
  - NBMPeerConnection
- JSON-RPC

  Objects
  - <NBMMessage>
  - NBMRequest
  - NBMResponse

  Client
  - NBMJSONRPCClient