Building Native WebRTC Applications against the Pexip client API

Ian Mortimer - ian@pexip.com

Contents

1 Introduction . . . **3**
- 1.1 Who this guide is for . . . 3
- 1.2 Background reading . . . 3
- 1.3 A word on security . . . 3

2 An introduction to the call flow . . . **3**
- 2.1 Getting access to the conference . . . 4
  - 2.1.1 Requesting a token . . . 6
  - 2.1.2 Understanding your token . . . 6
  - 2.1.3 Subscribing to the event stream . . . 6
  - 2.1.4 Refreshing a token . . . 6
  - 2.1.5 Releasing a token . . . 6
- 2.2 Initial SSE flow when first connecting . . . 6
- 2.3 Dealing with subsequent SSEs . . . 7
- 2.4 Sending and receiving presentations . . . 7
  - 2.4.1 Receiving a presentation . . . 8
  - 2.4.2 Sending a presentation . . . 9
- 2.5 Sending and receiving chat messages . . . 9
  - 2.5.1 Sending chat messages . . . 9
  - 2.5.2 Receiving chat messages . . . 10
- 2.6 Establishing Media . . . 10
  - 2.6.1 High level call flow . . . 11
  - 2.6.2 Initializing the RTCPeerConnection . . . 11
  - 2.6.3 Building media streams from tracks . . . 12
  - 2.6.4 Creating the offer with ICE candidates . . . 12
  - 2.6.5 Mangling the SDP to set bandwidths and resolutions . . . 13
  - 2.6.6 Sending the SDP offer . . . 13
  - 2.6.7 Receiving the SDP answer . . . 13
  - 2.6.8 Connecting streams . . . 13
  - 2.6.9 Starting media flow . . . 14
  - 2.6.10 Switching streams . . . 14
  - 2.6.11 Crossing the streams . . . 14
  - 2.6.12 Disconnecting media flow . . . 14

3 Platform Specific Considerations . . . **14**
- 3.1 iOS . . . 14
  - 3.1.1 WebRTC binaries . . . 14
  - 3.1.2 Bitcode . . . 15
  - 3.1.3 Hardware Acceleration . . . 15
- 3.2 Android . . . 15

4 Building WebRTC . . . **15**
- 4.1 Building WebRTC for iOS . . . 15
  - 4.1.1 Custom patches for Pexip . . . 15
  - 4.1.2 Building . . . 15
  - 4.1.3 Other settings . . . 17
- 4.2 Building WebRTC for Android . . . 17

5 Example Code . . . **18**
- 5.1 License . . . 18

5.2 iOS 18
5.2.1 Design Considerations 18
5.2.2 DNS SRV 18
5.3 Android 19
5.3.1 Design Considerations 19
5.3.2 PexRTC wrapper 19
5.3.3 Native 21
5.4 Cordova 21

1 Introduction

In the following article, we'll be covering the steps required to build a complete working WebRTC based video and chat application using the Pexip client REST API and native builds of WebRTC for both Android and iOS platforms.

1.1 Who this guide is for

This guide is intended for developers of iOS or Android applications that want to create a custom work flow around the powerful video interoperability and conferencing capabilities of the Pexip platform. This guide assumes you have a good understanding of how to develop applications for your chosen platform and are familiar with making HTTP queries to REST API's from your platform of choice. You should also have basic familiarity with normal developer tools such as building and compiling applications from a command line.

This guide will not cover general application design or user experience but there are some recommendations in Sections 5.2.1 and 5.3.1

1.2 Background reading

The examples here assume you are familiar with the basic operation of the Pexip REST API for clients as documented here https://docs.pexip.com/api_client/api_rest.htm You should also be familiar the basic concepts of WebRTC as outlined here https://webrtc.org/faq/ Along with this you should also be familiar with the basic concepts of the Pexip platform here https://docs.pexip.com/admin/admin_intro.htm and the services offered by the platform here https://docs.pexip.com/admin/admin_services.htm

1.3 A word on security

As there is potentially sensitive information traveling over the client API, all client connections are performed over an HTTPS connection. Valid certificates should be installed for your platform (see https://docs.pexip.com/admin/certificate_management.htm?#overview) and you should not use self signed certificates or make attempts to bypass any OS level checks (e.g. Application Transport Security on the iOS platform). As you are in complete control of each HTTPS connection you could even perform things like certificate pinning but this is beyond the scope of this document.

You should also heed industry standard practices when parsing information received either from user input or from any incoming connection and also when storing / logging information e.g. passwords or tokens to ensure your application does not compromise the users security in any way.

2 An introduction to the call flow

The following sequence diagrams outline the high level call flow involved in setting up a connection to the Pexip service. The 'Client' is your application making HTTP requests and the 'MCU' is a Pexip worker node in your deployment. Worker nodes are normally discovered using DNS SRV for a specific domain.

Full documentation of the client API can be found here https://docs.pexip.com/api_client/api_rest.htm

2.1 Getting access to the conference

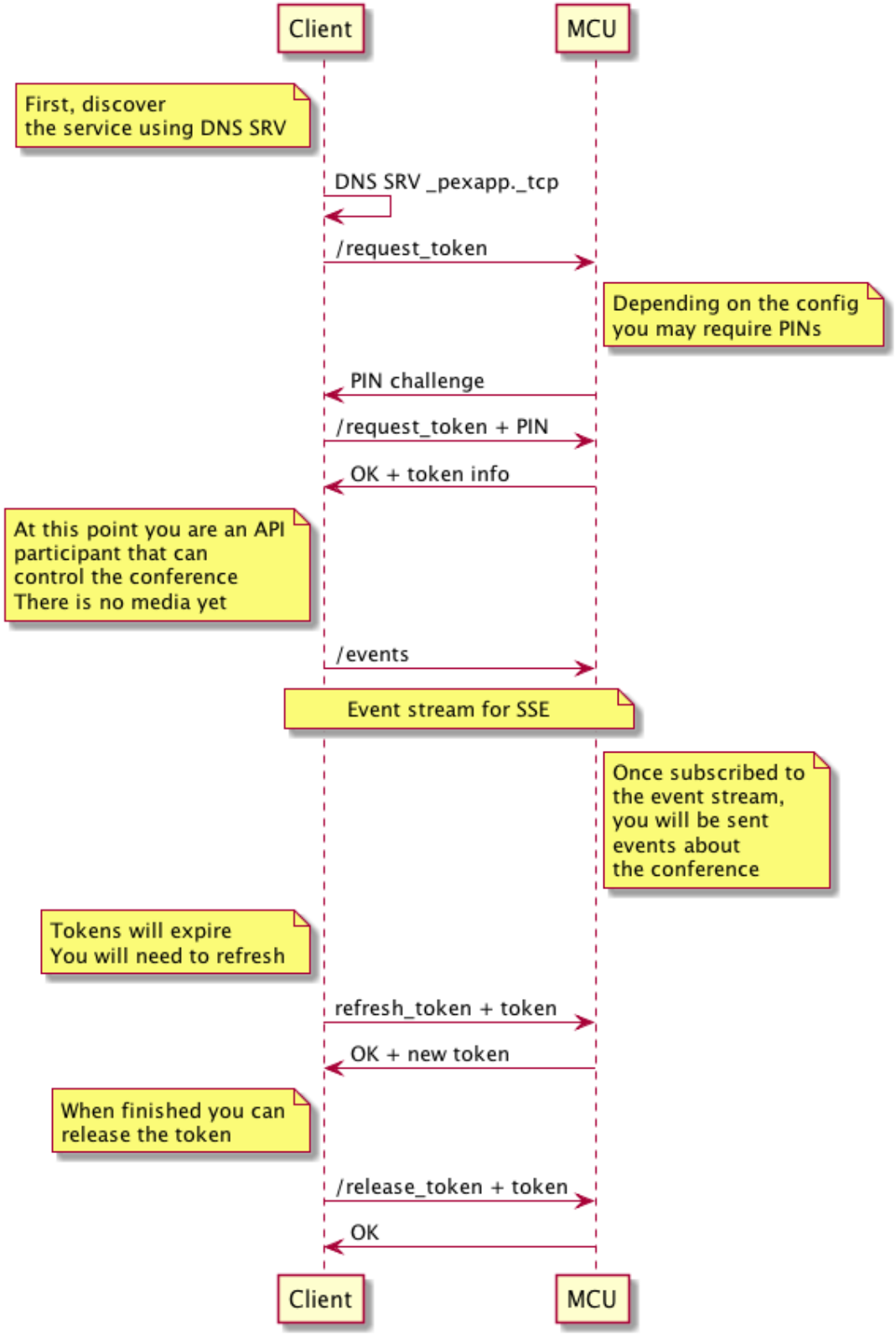
This call flow shows just the basic API participant flow. An API participant has no media associated with it but is a fully fledged participant that can control the conference, receive events, view the roster list, send and receive chat messages and also send / receive presentations. When developing your application it is important to get this call flow working reliably as it forms the basis for all communications with the MCU.

The token you receive must be used for all subsequent transactions with the MCU and it must be refreshed (the expiry time will be given to you). If you fail to provide a token or provide a token that is invalid or expired that request will fail and once the original request expires, your participant will be ejected from the conference.

If your conferencing worker nodes are behind some form of proxy e.g. a reverse proxy for load balancing you may need to deal with HTTP authentication challenges and or SSL certificate challenges.

Documentation for the client control requests can be found here `https://docs.pexip.com/api_client/api_rest.htm#client_summary`

To discover the service via DNS SRV (see `https://docs.pexip.com/admin/dns_records.htm#connect` for setup details), you should take the conference URI from the user in the form of `conference@domain.org` and extract the domain and conference parts. Perform a look up for `_pexapp._tcp.domain.org` to see if there are any SRV records available for `domain.org`. If there are none, you should just use the domains A record entry. See Section 5.2.2 for details on sample DNS SRV implementations.



Once finished with your connection, you must perform the `release_token` request so the MCU can clear down any resources taken by your client and potentially end the conference. If you do not release your token, the MCU will maintain the participant in the roster list until the token expires.

2.1.1 Requesting a token

The token request is the very first contact you make with the client API and determines all further actions. See https://docs.pexip.com/api_client/api_rest.htm#request_token for full details. You will need to make sure that any services (VMRs, Gateway rules) configured on your deployment have aliases that match what you are dialing or you will receive a not found response. The `request_token` exchange is also where you will deal with any PINs that may have been configured on the services or supply conference extensions for Virtual Reception Rooms. See the **PIN protected conferences** and **Virtual Receptions** section of the above `request_token` documentation.

2.1.2 Understanding your token

The token provides a large amount of information about the service you are connecting to. The response to a successful `request_token` should be parsed out to provide the correct feedback to the user. At a minimum you should store the `token` string to use in headers for all subsequent requests, the `participant_uuid` as this is needed in later operations, your `role` as this will determine what you can and can't do in a conference and your `service_type` as this determines what type of conference you are in or if you are in a waiting room.

2.1.3 Subscribing to the event stream

Once you have a token you must subscribe to the event stream in order to receive further updates about the conference. See https://docs.pexip.com/api_client/api_rest.htm#server_sent. Once subscribed, you will receive an initial burst of events detailed in Section 2.2. You can use your own implementation of the W3C SSE spec (https://www.w3.org/TR/2011/WD-eventsource-20111020/) or use a readily available off-the-shelf version - see Section 5 for examples.

2.1.4 Refreshing a token

The `expires` field of the original `request_token` in Section 2.1.1 will give you a time in seconds before the token expires. We recommend refreshing your token at an interval of `expires/2` seconds. See https://docs.pexip.com/api_client/api_rest.htm#refresh_token for full information.

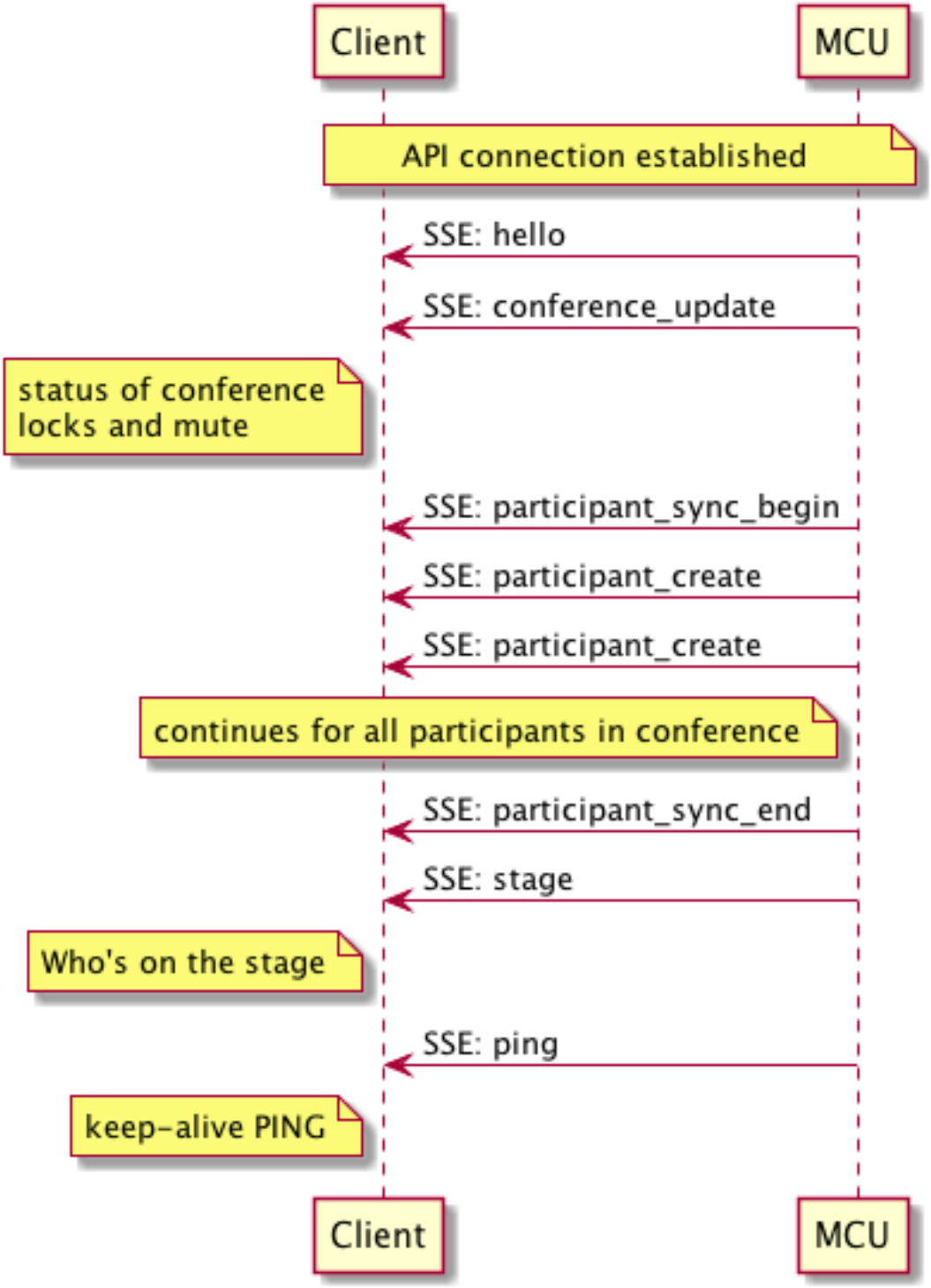
2.1.5 Releasing a token

This will disconnect you from the conference and clear any resources used by your participant. See https://docs.pexip.com/api_client/api_rest.htm#release_token

2.2 Initial SSE flow when first connecting

When you first connect the event source, the MCU will send you a set of events so you can correctly display the initial list of participants in the roster and prepare yourself. The full list of events can be seen here https://docs.pexip.com/api_client/api_rest.htm#server_summary

The initial participant sync is useful for creating the roster list of participants in your application and only happens on connection. Stage updates show you who is currently "on the stage" i.e. visible in the main window. You can find out who is talking by looking at the information in the participant update messages.



2.3 Dealing with subsequent SSEs

A full list of SSEs located here `https://docs.pexip.com/api_client/api_rest.htm#server_summary` and should be dealt with accordingly e.g. on a `participant_create` message, you should add a participant into the roster and potentially display this information to the user.

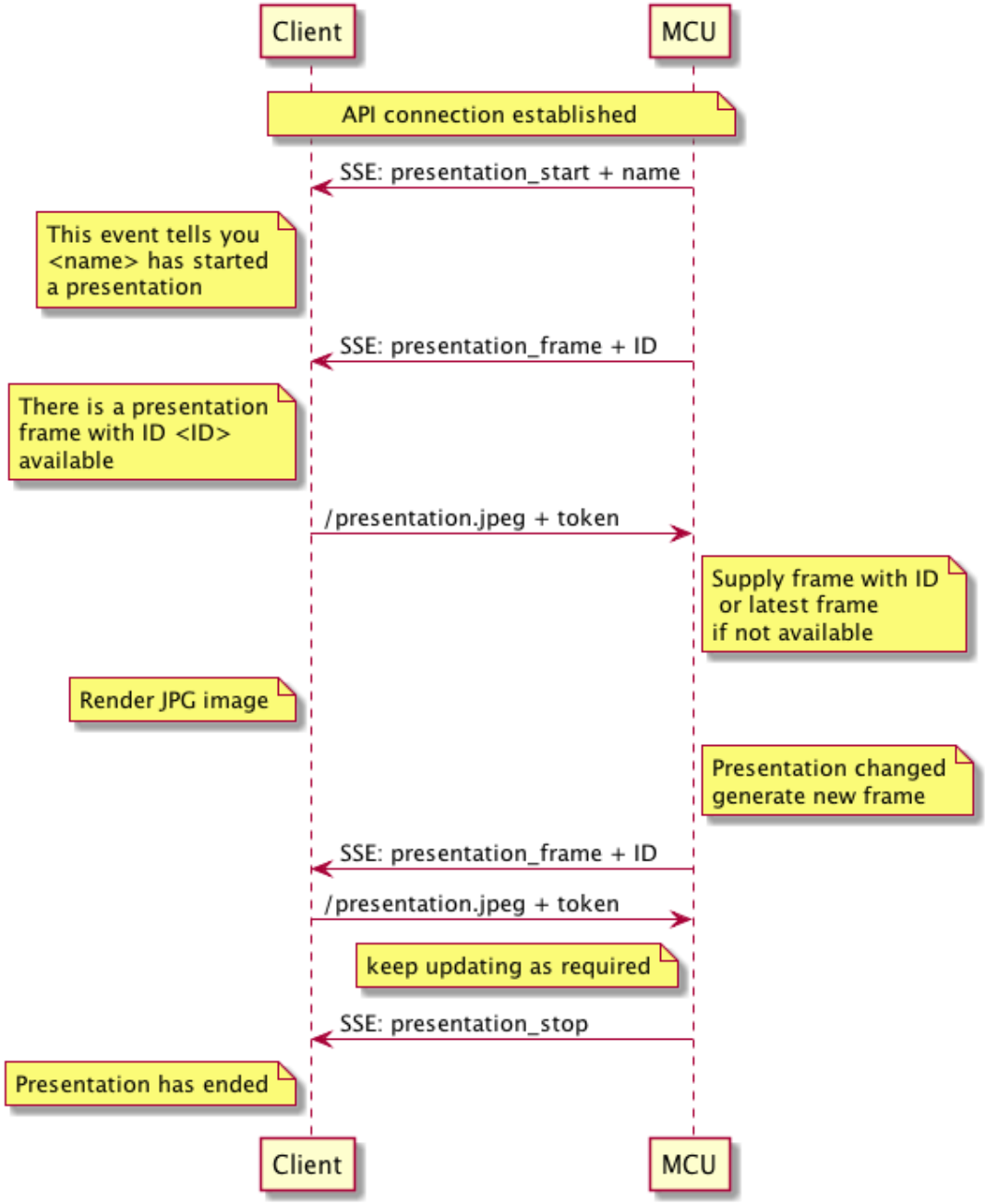
2.4 Sending and receiving presentations

There are two methods for sending / receiving presentations. The first is to respond to a `presentation_started` and `presentation_frame` event by requesting a `presentation.jpg`

from the MCU and displaying it to the user and every time you get a new presentation frame event, requesting a new `presentation.jpg`. The second is to establish a presentation call and the MCU will send you a media stream (often referred to as an HD presentation) that you can render. For mobile clients, rendering two simultaneous video streams is a lot of work and is not recommended so only the image based presentation is shown here but there is documentation for the REST calls required if you wish to do this.

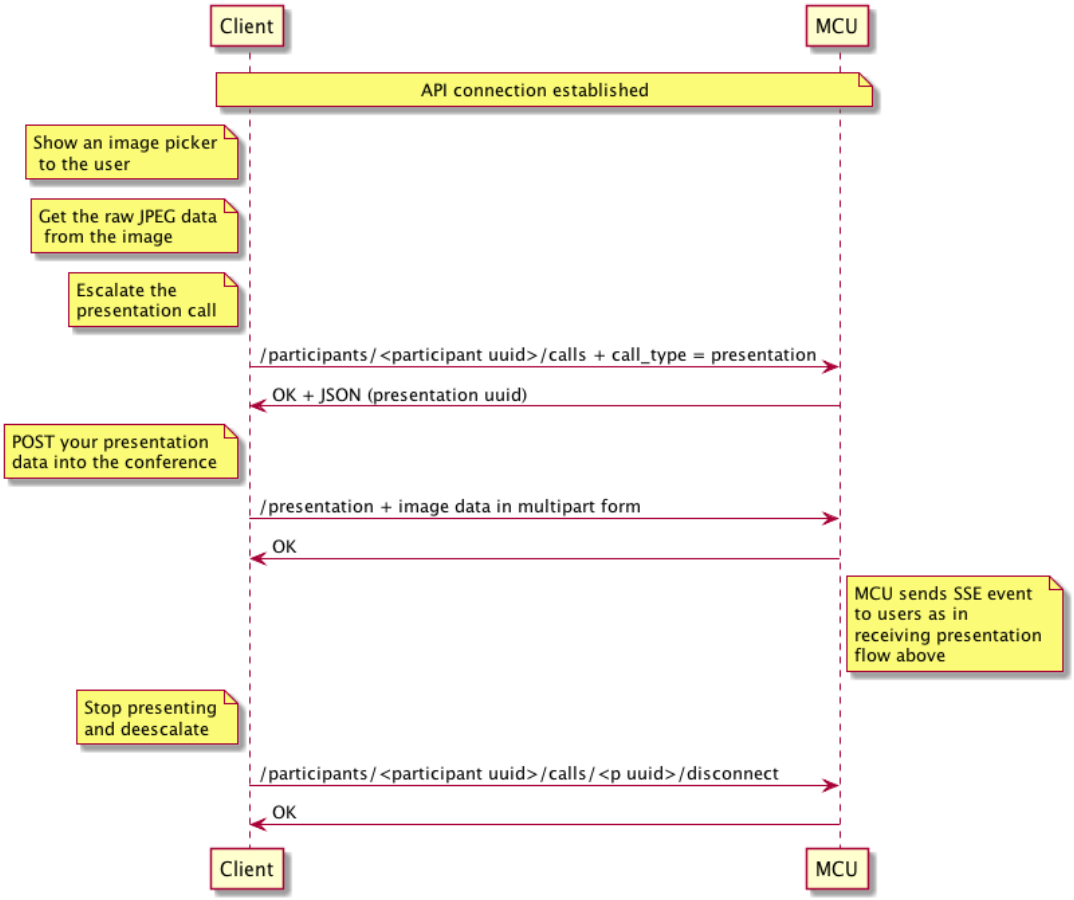
2.4.1 Receiving a presentation

This example flow shows the reception of a presentation using images from the MCU. See `https://docs.pexip.com/api_client/api_rest.htm#presentation_start` for more information



2.4.2 Sending a presentation

This example flow shows the flow for presenting images from a client device into the conference.

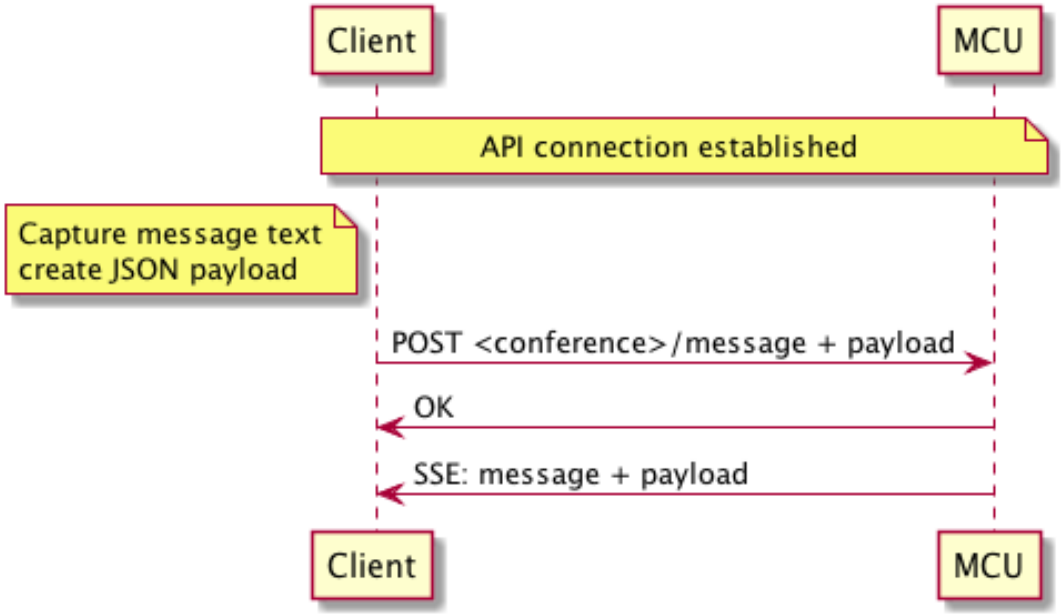


2.5 Sending and receiving chat messages

Chat messages are broadcast to the entire conference. There is no one-to-one or private chat.

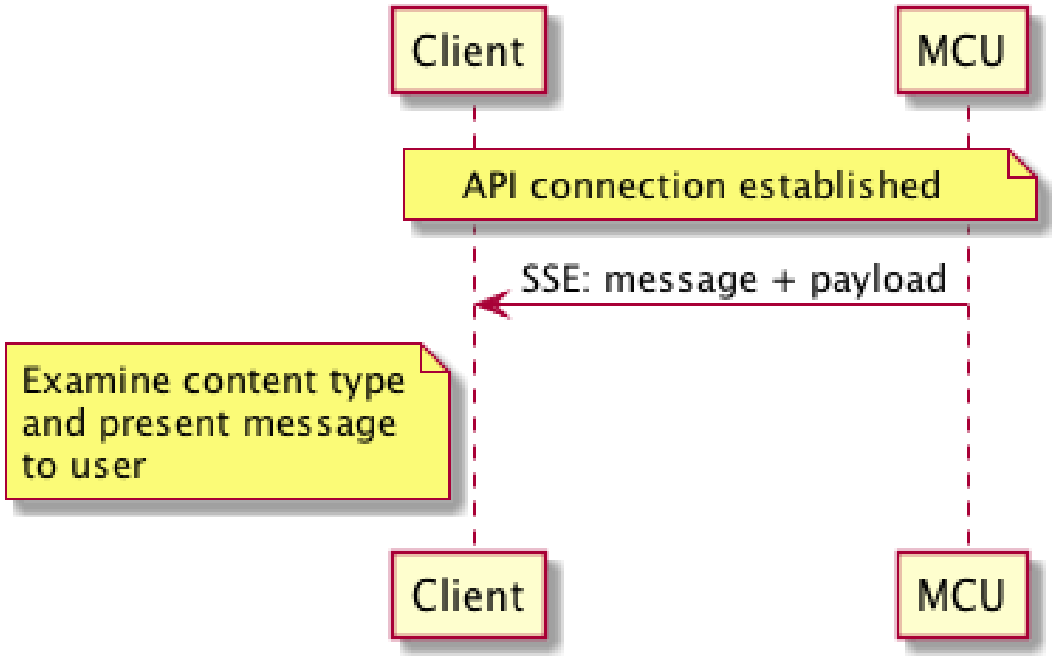
2.5.1 Sending chat messages

See https://docs.pexip.com/api_client/api_rest.htm#message



2.5.2 Receiving chat messages

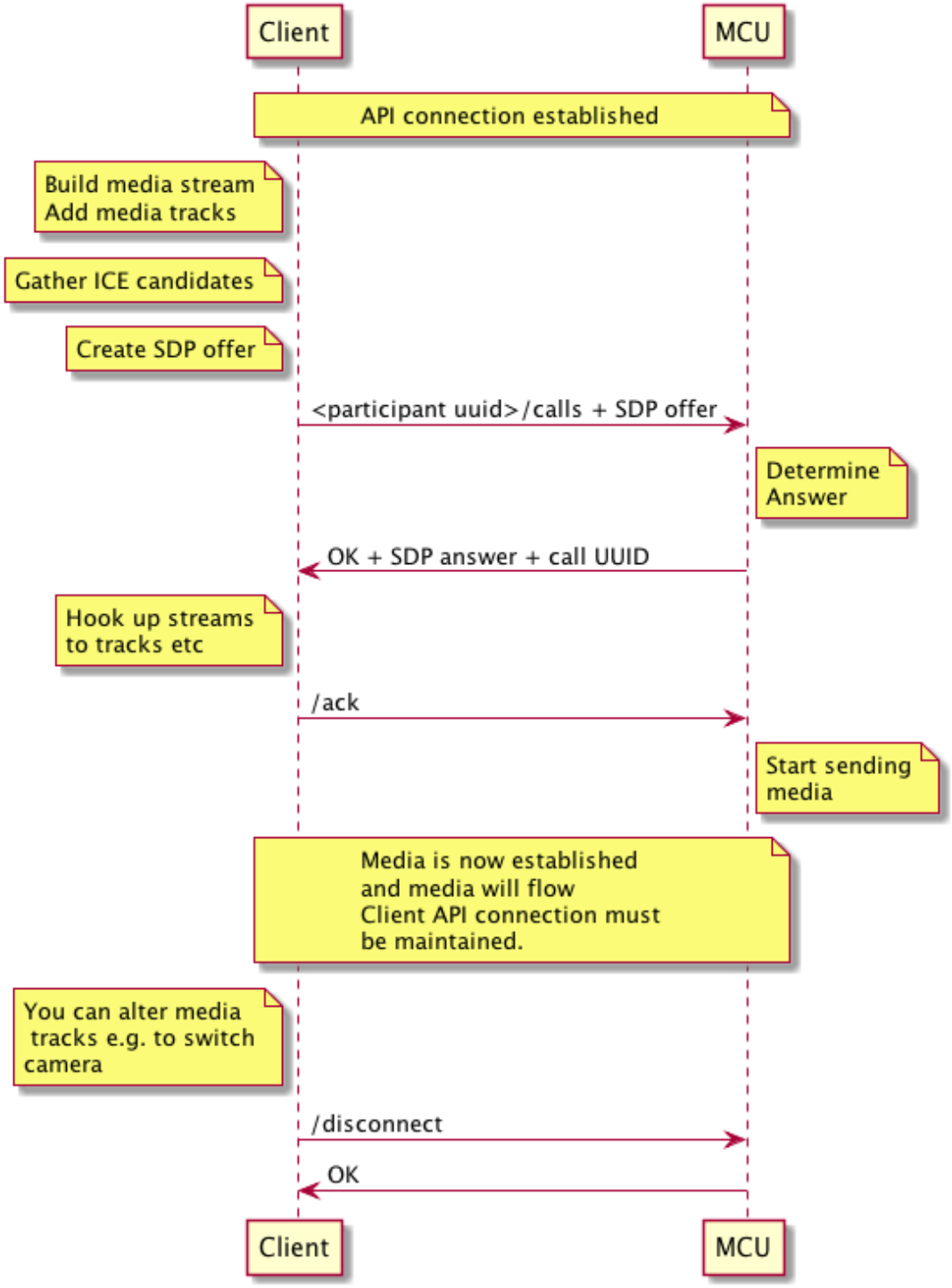
See https://docs.pexip.com/api_client/api_rest.htm#message_event



2.6 Establishing Media

Once connected to the conference as an API participant you can then escalate media. Media escalation can be audio only or a full audio/video session. This part of the call flow is more involved as it requires you to setup media devices (cameras and microphones) and also perform what is called the offer/answer dance whereby the two participants in the conversation (your application and the MCU media engine) decide what codecs to use and how best to route media to each other using ICE negotiation. It is here where you can set things like bandwidth of the call and video resolution.

2.6.1 High level call flow



The generation of SDP and ICE candidates is handled for you by the WebRTC library and is explained below.

2.6.2 Initializing the RTCPeerConnection

The `RTCPeerConnection` object is the main interface to the WebRTC library and is created using a factory. First, we must first initialize the SSL libraries underneath everything by running `RTCInitializeSSL()`. Once we have that done, we can then create a `RTCPeerConnectionFactory` object and build the component parts to create our `RTCPeerConnection` object.

First we need an `RTCBundlePolicy` set with `maxCompat` so as not to bundle all media over a single port and we'll also need to fill in any ICE server configuration at this point e.g. if your TURN/ICE servers need any authentication setup (see https://docs.pexip.com/admin/about_turn_server.htm for more information) We can then set our media constraints to show if we're offering video or audio or both and also set the `DtlsSrtpKeyAgreement` to true. Once we have those, we can pass them into the factory to produce our RTCPeerConnection object. For iOS, this could look like:

```
self.factory = RTCPeerConnectionFactory()

self.resolution = resolution

self.videoView = videoView
self.videoSelfView = videoSelfview

self.videoEnabled = videoEnabled

self.rtcConfg = RTCConfiguration()

// Set this to maxCompat for correct operation with Pexip MCU
self.rtcConfg?.bundlePolicy = RTCBundlePolicy.maxCompat

// If your ICE servers need credentials, create them in here as RTCIceServer objects
self.rtcConfg?.iceServers = []

self.rtcConst = RTCMediaConstraints(
    mandatoryConstraints: [
        "OfferToReceiveAudio" : "true",
        "OfferToReceiveVideo" : self.videoEnabled ? "true" : "false"
    ],
    optionalConstraints: [
        "DtlsSrtpKeyAgreement": "true"
    ]
)
```

2.6.3 Building media streams from tracks

Once we have our `RTCPeerConnectionFactory`, we can also use it to create our media streams and assign our audio and video tracks to them. For iOS we'll be using `AVFoundation` sources and using the `avFoundationVideoSource`, `videoTrack` and `audioTrack` methods of the `RTCPeerConnectionFactory`. For iOS, this could look like:

```
self.peerConnection = self.factory?.peerConnection(with: rtcConfg!, constraints: rtcConst!, delegate: self

self.mediaStream = self.factory?.mediaStream(withStreamId: "PEXIP")

if self.videoEnabled {
    print("Call adding video")
    let mandatory = ["minWidth":"\(self.resolution!.width())", "maxWidth":"\(self.resolution!.width())"]
    let constraints = RTCMediaConstraints.init(mandatoryConstraints: mandatory, optionalConstraints: [:])
    let videoSource = self.factory?.avFoundationVideoSource(with: constraints)
    self.videoTrack = self.factory?.videoTrack(with: videoSource!, trackId: "PEXIPv0")
    self.videoTrack?.add(self.videoSelfView!)
    self.mediaStream?.addVideoTrack(self.videoTrack!)
}
self.audioTrack = self.factory?.audioTrack(withTrackId: "PEXIPa0")
```

2.6.4 Creating the offer with ICE candidates

Creating the offer using the `RTCPeerConnection` object will trigger a bunch of calls that must be handled by your delegate in particular:

RTCPeerConnection didChange newState RTCICEGatheringState

As this is what will return the final SDP back up to the application so we can POST it to the MCU as our offer. For iOS this could look like:

```
self.peerConnection?.offer(for: self.rtcConst!, completionHandler: offerCompletionHandler)
```

2.6.5 Mangling the SDP to set bandwidths and resolutions

Before we POST the SDP to the MCU we must manipulate our SDP to set the supported bandwidth and resolutions for the call e.g. make this a wCIF call at 384kbps. You could make this decision on behalf of your user by looking at the connectivity of the device e.g. WiFi or Cellular or through selection from user input e.g. "High Quality" might convert to a 2Mbps call @720p. Bear in mind that this is only what is offered to the MCU, it might not actually end up being negotiated and honored. For more information, the reader is pointed to https://tools.ietf.org/html/rfc4566. We can mangle the SDP once we reach the `RTCICEGatheringState.complete` i.e. all the ICE candidates have been discovered and added to the description and we only really need to set the `AS` and `TIAS` setting for the total session bandwidth and the RTC constraints for capture device to set the out bound resolutions (`minWidth`). For iOS, this could look like:

```
private func mutateSdpToBandwidth(sdp: RTCSessionDescription) -> RTCSessionDescription {
    let h264BitsPerPixel = 0.06
    let fps = 30.0
    // if using AAC-LD --> 128.0
    let audioBw = 64.0
    let videoBw = Double(self.resolution!.width()) * Double(self.resolution!.height()) * fps * h264BitsPerPixel
    let asValue = Int((videoBw / 1000) + audioBw)
    let tiasValue = Int(videoBw + audioBw)
    let range = sdp.sdp.range(of: "m=video.*\\r\\nc=IN.*\\r\\n", options: .regularExpression)
    var origSdp = sdp.sdp
    let bwLine = "b=AS:\(asValue)\r\nb=TIAS:\(tiasValue)\r\n"
    if range != nil {
        origSdp.insert(contentsOf: bwLine.characters, at: range!.upperBound)
        return RTCSessionDescription(type: sdp.type, sdp: origSdp)
    } else {
        return sdp
    }

}
```

2.6.6 Sending the SDP offer

Once we have a complete offer with all candidates and we have manipulated the SDP to what we want we can now POST this to the MCU. See https://docs.pexip.com/api_client/api_rest.htm#calls

2.6.7 Receiving the SDP answer

Once the MCU has calculated an answer for for our offer, it will send back its response and we can then manipulate this further e.g. to limit the out bound bandwidth from our device and then pass this into our `RTCPeerConnection` objects `remoteDescription`. For iOS, this could look like:

```
func setRemoteSdp(sdp: RTCSessionDescription, completion: @escaping (_ error: Error?) -> Void) {
    print("mutating remote SDP bandwidth")
    let mutated = self.mutateSdpToBandwidth(sdp: sdp)
    self.peerConnection?.setRemoteDescription(mutated) { error in
        print("Setting remote SDP on connection, status: \(error)")
        self.remoteSdpCompletion = completion
        completion(nil)
    }
}
```

2.6.8 Connecting streams

Once our `RTCPeerConnection` objects `remoteDescription` has been set and accepted you can now connect up the incoming streams to your views to display the video and play the sound when the delegate function fires:

RTCPeerConnection didAdd stream RTCMediaStream

In this event, you can pick out the audio and video streams and attach them to your `RTCEAGLView` renderers i.e. the views you have setup to show video in your app. For iOS, this could look like:

```
func peerConnection(_ peerConnection: RTCPeerConnection, didAdd stream: RTCMediaStream) {
    print("peer connection didAddStream")
    if stream.videoTracks.count > 0 && self.videoEnabled {
        print("Got video track")
        self.videoTrack = stream.videoTracks[0]
        self.videoTrack?.add(self.videoView!)
    }
    if stream.audioTracks.count > 0 {
        print("Got audio track")
        self.audioTrack = stream.audioTracks[0]
    }
}
```

2.6.9 Starting media flow

Once the offer/answer dance has completed and you have wired up your streams, you can know trigger the MCU into sending media by sending an `ack` message: See `https://docs.pexip.com/api_client/api_rest.htm#ack`. Once this completes, you should start to see and hear media in your application.

2.6.10 Switching streams

You can switch tracks in the media stream to allow you to do things like swapping between front and rear cameras. You can do this by selecting the source (e.g. for iOS, selecting an `AVFoundation` source with the correct `AVCaptureDevicePosition`), removing the old media stream, and then re-adding the new media stream created from your new tracks as in Section 2.6.3

2.6.11 Crossing the streams

Never cross the streams, it would be bad.

2.6.12 Disconnecting media flow

Once you have finished you can disconnect the media and drop back down to an API participant. See `https://docs.pexip.com/api_client/api_rest.htm#call_disconnect`

3 Platform Specific Considerations

3.1 iOS

3.1.1 WebRTC binaries

At the time of writing, there is no native support for WebRTC in the safari view controller so the only way to work with WebRTC is via a binary install of the WebRTC library and rendering the results using RTCEAGLViews in your UI. You will need to build the WebRTC library yourself (see Section 4) and then include the header files in your project. If you are using Swift, make sure that a bridging header is in place to expose the bindings. All examples for this document will use Swift but the concepts transfer directly for Objective-C based applications.

The library can be compiled with the full instruction set for use on all ARM platforms and also with 386 and X86_64 instructions for use in the simulator. You can control which

platforms to target and hence control the size of the included library. When submitting to the App Store, you **must** remove these non-ARM architectures from the library or your app will be rejected.

3.1.2 Bitcode

At the time of writing, there is also no support for Bitcode in the library so you will need to disable this for your project.

3.1.3 Hardware Acceleration

Hardware acceleration is enabled by default for the latest builds of WebRTC and this should dramatically reduce the CPU load used when decoding H264 streams but be aware that decoding video and audio is still a very intensive workload and consideration should be taken when deciding bandwidths and resolutions to negotiate with the MCU i.e. a 2Mbps stream at 720p HD resolution will require a modern ARM processor and a lot of CPU power and older phones will struggle to decode this in a timely manner.

On iOS devices with an A4 up to A6 processor, there is support for H.264/AVC/MPEG-4 Part 10 (until profile 100 and up to level 5.1), MPEG-4 Part 2, and H.263. The A7 added support for H.264's profile 110.

3.2 Android

For Android, there are actually two ways you can interact with WebRTC; a web view with built in support for WebRTC that can be accessed via JavaScript bindings; or using the same method as iOS with a WebRTC library and a pure Java implementation. See Sections 4.2 and 5.3.1 for more information.

4 Building WebRTC

4.1 Building WebRTC for iOS

4.1.1 Custom patches for Pexip

At the time of writing, there is a custom patch to workaround support for rotation of video streams that must be applied in order for the video stream to be rotated to the correct orientation. See Section 5 for links to the patches.

The reader is advised to use a "branch head" when building rather than master as this is slightly more stable and reliable when building. At the time of writing, branch head 56 was used.

```
git checkout branch-heads/56
glclient sync
```

We recommend building the framework as this greatly simplifies addition to your project although if you want more fine grained control, you can build the static library and only include what you need.

4.1.2 Building

Building WebRTC for iOS must be performed on a Mac - these examples were performed on a Macbook Pro running macOS 10.12.1. The canonical build instructions are `https://webrtc.org/native-code/ios/` but the following process is a good summary.

1. Prerequisites

   (a) depot tools
   Clone the depot tools

   ```
   git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
   ```

   Make sure they are in your path:

   ```
   export PATH=‘pwd‘/depot_tools:$PATH
   ```

   (b) WebRTC code
   Create a working directory, enter it and fetch the code:

   ```
   mkdir ~/webrtc
   cd ~/webrtc
   fetch --nohooks webrtc_ios
   ```

   Now sync and pull down the code. This will take a long time and should not be interrupted as multiple gigabytes of data will be downloaded.

   ```
   gclient sync
   ```

2. Building Once you have the source (and have applied any necessary patches) you can either build a static binary or a framework. Building a framework is the simpler option but the static binary gives you more control.

3. Building the framework

   ```
   cd ~/webrtc/src
   ```

   Build the framework:

   ```
   ./webrtc/build/ios/build_ios_libs.sh
   ```

   The result will be a directory called `out_ios_libs` containing the framework called `WebRTC.framework`. You can now embed this directly into you project.

4. Building the static Binary

   ```
   cd ~/webrtc/src
   ```

   Build the static binary:

   ```
   ./webrtc/build/ios/build_ios_libs.sh -b static_only -o out
   ```

   The result will be a library and a set of headers in the `out` directory.

   The `out` directory will contain a single `librtc_sdk_objc.a` with all architectures combined and sub directories containing the individual architectures. The header files will be located in `./webrtc/sdk/objc/Framework/Headers/WebRTC/`

   If the build script fails you can run the compilation manually:

```
gn gen out/arm64 --args='target_os="ios" target_cpu="arm64" \
is_component_build=false is_debug=false'
gn gen out/arm --args='target_os="ios" target_cpu="arm" \
is_component_build=false is_debug=false'

ninja -C out/arm64 rtc_sdk_framework_objc
ninja -C out/arm rtc_sdk_framework_objc
```

(a) Adding WebRTC and headers to your Swift project

i. Create a new group in your project hierarchy called WebRTC and create a new header file called `<bundle id>-Bridging-Header.h`

ii. Copy in the binary lib and headers created in the build process above.

iii. Add the import lines to the bridging header

A. You can run the following command from the headers directory from Section 2 to get a listing ready to paste in:

`ls *h | awk '{print "#import \"" $NF "\""}'`

You won't need all of these headers and the macOS ones can be removed.

iv. Make sure `Build Settings -> Objective-C Bridging Header` has a path set to the new bridging header you created

v. Include all the other frameworks and libraries required for proper operation including your freshly built `librtc_sdk_objc` library, in the following order:

- libresolv.tbd
- AVFoundation.framework
- CoreMedia.framework
- GLKit.framework
- OpenGLES.framework
- CoreVideo.framework
- CoreAudio.framework
- QuartzCore.framework
- AudioToolbox.framework
- libc++.tbd
- libstdc++.tbd
- VideoToolbox.framework
- librtc_sdk_objc.a

4.1.3 Other settings

- Make sure the `Build Settings -> Other linker flags` is set to `-ObjC` or you'll get weird crashes about unknown signatures.
- turn off bit-code support in `Build Settings`

4.2 Building WebRTC for Android

First, please take into account the design considerations mentioned in Section 5.3.1 as this will determine if you even need to compile WebRTC for Android.

The canonical build instructions are here `https://webrtc.org/native-code/android/` but at the time of writing are still using the old `gyp` build system but the main repository has moved over to the new `gn` build system.

5 Example Code

The example code is intended as a rudimentary guide to show you the basic concepts for working with the API and it **should not be used as a basis for your application** as, for example, it does no validation of user input, no error checking for responses from the API, no proper layout of the video elements, no care for proper background operation and no translation of any user interface elements into other languages. The examples also do not follow any of the HIGs laid out by Apple (https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/) or Google (https://developer.android.com/design/index.html)

The example code can be found in https://github.com/pexip/pexkit-sdk.

5.1 License

All examples and code are released under the MIT license:

```
The MIT License (MIT)

Copyright (c) 2016 Pexip

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sub license, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

5.2 iOS

5.2.1 Design Considerations

It's a good idea to encapsulate the Conference functionality (e.g. token handling, event handling, chat, presentation and call handling) into a single class and the Call functionality (dealing with WebRTC) in another class. You can also setup structures for important pieces of information e.g. resolutions, service errors etc

5.2.2 DNS SRV

We have provided an example DNS SRV code snippet that you may include in your projects if you prefer not to implement your own.

5.3 Android

5.3.1 Design Considerations

As WebRTC is implemented in the Chrome web view for the Android platform, there are a couple of options open to you when working with WebRTC. Firstly, you can work with the native web view and call the REST API as the iOS client or you can use a wrapper around our PexRTC JavaScript bindings to alleviate some of the work required.

The following examples assume you are using Android Studio and an API Level of 21 or above.

1. To bundle PexRTC.js or not

   In order to use PexRTC.js you can either bundle the JavaScript with the wrapper project or fetch the JavaScript from your running deployment.

   The wrapper includes a function called `fetchPexRTCSource` that will pull in the running version of PexRTC on the targeted deployment. You should make sure that any functions you are calling are still supported in the current version.

   The supplied wrapper already includes a bundled version of PexRTC, you may need to update this every time you publish your app.

2. Building the wrapper

   Simply open the supplied project and build it. This will produce an AAR file (in `pexrtc-android-wrapper/app/build/outputs/aar`)

5.3.2 PexRTC wrapper

Import the wrapper library into your project using "File –> New –> New Module" and select "import JAR/AAR package" from the selection.

Then navigate to the AAR wrapper file and import it. Once complete, you will now have this available in your resources side bar.

Open you project structure ("File –> Project Structure") then select your existing "app" module and then select the dependencies tab and add (+) a module dependency and select the AAR submodule you just imported.

From your app, you should now be able to import `com.pexip.android.wrapper.PexView`

You can now add in your PexView (either or programmatically or via adding it to your layout.xml file).

1. fragments vs normal view destruction

   For configuration changes e.g. rotation, there are two methods of dealing with the view destruction: use fragments or disable view destruction (non-fragment way).

   The following example uses the non-fragment method so you will need to add the following line to your android manifest inside the activity:

   ```
   android:configChanges="orientation|screenSize"
   ```

   e.g.

   ```
   <activity android:name=".MainActivity"
        android:configChanges="orientation|screenSize" >
        <intent-filter>
        <action android:name="android.intent.action.MAIN" />
   ```

```
    <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

2. pro grammatic addition of PexView

   In your `onCreate` method of your main activity:

   ```
   final PexView pexView = new PexView(this);
   ```

   Once you've have this, you can then grab the ID of your layout (e.g. `myLayoutId`) from your layout XML file and then add the PexView to it.

   ```
   RelativeLayout layout = (RelativeLayout) findViewById(R.id.myLayoutId);
   layout.addView(pexView);
   ```

   If you wish to show a self view:

   ```
   WebView selfView = new WebView(this);
   layout.addView(selfView);
   pexView.setSelfView(selfView);
   ```

3. addition via layout.xml

   Using the design view, add a custom view (search for PexView) and add it where you see fit and then link them to your main activity e.g.

   ```
   PexView pexView = (PexView) findViewById(R.id.pexViewId);
   WebView selfView = (WebView) findViewById(R.id.selfViewID);
   pexView.setSelfView(selfView);
   ```

4. setting up callbacks for events

   Once you have your views setup, you can now register the callbacks for the events from PexRTC e.g.

   ```
   pexView.setEvent("onSetup", pexView.new PexEvent() {
           @Override
           public void onEvent(String[] strings) {
               pexView.setSelfViewVideo(strings[0]);
               pexView.evaluateFunction("connect");
           }
       });

   pexView.setEvent("onConnect", pexView.new PexEvent() {
           @Override
           public void onEvent(String[] strings) {
               if (strings.length > 0 && strings[0] != null)
                   pexView.setVideo(strings[0]);
           }
       });
   ```

```
pexView.addPageLoadedCallback(pexView.new PexCallback() {
        @Override
        public void callback(String args) {
            // make a call
            pexView.evaluateFunction("makeCall", "pexipdemo.com", "meet.vmr", "My Di
        }
    });

// Load the page which will then trigger the callbacks
pexView.load();
```

5. adding permissions to the android manifest

   You'll need the following permissions in your manifest file:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

5.3.3 Native

Follow iOS instructions if you wish to go the native route.

5.4 Cordova

We have created a fork of the official Cordova plugin to work with the latest WebRTC. Access to these plugins will be made available at a later date.