Seajei Developer Guide for Linux

Version 3.2

Introduction

The Seajei SDK enables split-second connection and highly reliable video/audio streaming with ultra-low lag between phone apps and embedded systems such as Smart Home video cameras.

It can establish direct connections over the same Wifi, or connections across the internet using Wifi or cellular data (it will use peer-to-peer when possible, or server relay otherwise). The only requirement is that both devices are connected to the network.

The SDK has been designed to be easy to integrate and has a simple API that only requires a few lines of code and then it just works!

The SDK contains libraries for Linux / embedded systems, and frameworks for iOS (see SeajeiDeveloperGuideiOS).

Libraries have been built with the following toolchains:

- armv6-linux-gnueabi
- armv7-linux-gnueabihf
- armv8-linux-gnueabihf
- aarch64-linux-gnu
- mips-linux-gnu
- powerpc-linux-gnu
- x86_64-linux-gnu

The SDK was written entirely in C code (C99), and therefore can be compiled and run on any embedded system / microcontroller. It is very easy on CPU usage and requires little memory to run. Contact us at support@seajei.com if you need it built for your own hardware.

The SDK for Linux currently contains 4 libraries:

- CjConnectivity
- CjAudioPlayer
- CjAudioCapture
- CjRaspiVideoCapture

The communication library is called CjConnectivity. It is responsible for establishing the connection and streaming video, audio and any other type of data. It will resend lost packets, and indicate to the host device whether to lower or increase bitrate depending on network conditions.

The CjAudioPlayer library can play uncompressed audio frames (PCM).

The CjAudioCapture library can capture audio frames from the device's microphone (PCM).

And finally, the CjRaspiVideoCapture library enables a Raspberry Pi to capture H.264 video frames. Resolution currently can be set between 240p and 1080p, and can be updated on the fly. Other parameters include bitrate, framerate, and more.

Sample programs

See RaspberryPiSetupInstructions.pdf to build a RaspberryPi with video camera, USB microphone, and audio out connected to a headset or speakers.

Sample programs for Linux and Raspberry Pi are found in *SamplePrograms/Linux* and *SamplePrograms/RaspberryPi*.

In order to be able to build any of the sample programs, run the following command:

```
> sudo apt update; sudo apt install libopus-dev libssl-dev
portaudio19-dev
```

BasicReceiverSender

The basic_receiver_sender.c sample is a very basic example on how to use the Seajei CjConnectivity library to establish a data connection between two devices and send data.

It can be run on either Linux system, or Raspberry Pi (Pi 0, Pi 3 or Pi 4).

To build the Linux version, run the build_x86-64.sh script.

To build Raspberry Pi, run the build_pi_0.sh on Pi 0, or build_pi_3_4.sh on Pi 3 or Pi 4.

Then run the built executable and follow directions.

Doorbell

See doorbell.c for a complete doorbell working example on Raspberry Pi on how to listen to new connections, make a new connection and stream video and audio.

Build the sample program by running build_pi_0.sh or build_pi_3_4.sh depending on which Pi you are using.

Run the built executable on your Raspberry Pi, and then run the iOS demo app (SamplePrograms/iOS/SeajeiDemoApp) on a phone or simulator to connect the 2 devices together and see live video and 2-way audio.

Getting started

This tutorial shows how to use the CjConnectivity library in a typical situation for doorbells, security cameras, or baby monitors. It walks through the steps to initialize the library to listen to new connections, and stream video and audio both ways once a connection is established.

Then it shows how to send a push notification when the doorbell button gets pressed.

Initialization

First include the library header file:
`#include "cj_connectivity.h"`

Next initialize the library by calling the `cj_co_init` function.

The first argument of the function is a unique token. You can use the free trial token (*free-trial-64-4234-89c8-e1732f71059e*). It offers full features, with these exceptions:

- network bandwidth is limited to 3 Mbps when relayed via server
- session length is limited to 3 minutes
- no push notifications

If you want to remove those limitations by getting your own token, contact us at support@seajei.com.

The next argument is a unique ID for your device. This is the ID you would exchange with a phone app during the pairing process. Often the MAC address of the device is used. For the purpose of this simple test you can use a simple ID such as "123".

The next arguments are buffer sizes for audio and video. In a situation where your device is a doorbell for example, the situation would be that it needs to send video but not receive it. As a result the videoReceivingBufferSize would be set to 0, but the videoResendBufferSize would need a value because we would want to keep video data sent for some time so if losses occurred we can resend the missing packets. See `cj_co_init` reference for an explanation of a good size buffer to use.

For audio the device would need to both send and receive data, so both `nbrReceivingAudioFramesToBuffer` and `audioResendBufferSize` would need to be set. See `cj_co_init` reference for an explanation of good values to use.

Finally a callback function must be given. It will receive all the connection events, as well as incoming data.

Main loop

The library requires the `cj_co_process_main_loop` to get called repetitively, on the same thread as all the other functions. Suggested interval is every 4 milliseconds.

Push notifications

When for example the doorbell button is pressed, the device needs to call the `cj_co_send_push_notification` function.

When the phone app receives the push notification, it will initiate a connection.

Connection

Once the init function has been called, the library is ready to receive new connections. As soon as an app or other device tries to connect with our device's unique ID, the callback function will be called with CJ_CO_CONNECTION_EVENT_CONNECTING as connection event.

A successful connection will result in the CJ_CO_CONNECTION_EVENT_CONNECT_SUCCESS connection event being received. If there was a problem one of the failed connection events will be received.

After CJ_CO_CONNECTION_EVENT_CONNECT_SUCCESS, the callback may receive one or two CJ_CO_CONNECTION_EVENT_CONNECT_SUCCESS_OTHER_CONNECTION_TYPE connection events. Each is received along with a connection type, which indicates the type of connection, such as local (same WiFi), peer-to-peer or server relay.

When sending data, the library will use connection types in the following order of priority if connected: WiFi, P2P, Server.

Sending data

As soon as a connection has been established, the device is ready to stream audio, video, or any other type of data.

Video

Any time a new H.264 frame has been captured from the video camera module (I-frame or P-frame), check whether we have an active connection with the `cj_co_is_connected` function. If that is the case, simply send the frame with the `cj_co_send_video_h264_frame` function.

In case the device does not get fully formed frames from the camera module but somewhat random chunks of data, use the `cj_co_send_video_h264_data_chunk` function.

Audio

Any time audio frames have been captured from the microphone, check whether we have an active connection with the `cj_co_is_connected` function. If that is the case, compress the audio with a codec such as Opus, and simply send the compressed audio data with the `cj_co_send_audio_frames` function.

Other data

If you want to send other data, use the `cj_co_send_other_data` function.

Note: unlike the functions used for audio and video streaming, this function will not automatically resend lost packets.

Receiving data

When data is received, the callback function will be called with the CJ_CO_CONNECTION_EVENT_DATA_RECEIVED event.

The `CjCoDataType` parameter of the received event will indicate whether the data is video, audio or other.

If it's audio, decompress it and send it to the audio player with the `cj_ap_queue_audio_frames` function.

Variable bitrate

The CjConnectivity library tracks the state of the network connection while streaming. In case the network does not enable the timely streaming of the video because the bandwidth is too low or the network suffers significant packet losses, a CJ_CO_CONNECTION_EVENT_REDUCE_VIDEO_BITRATE event will be received through the callback function.

Similarly, if the network has no problems streaming the current video bitrate, a CJ_CO_CONNECTION_EVENT_INCREASE_VIDEO_BITRATE event will be received.

Dealing with network changes

In order for the device to deal with the network getting lost and coming back, call the `cj_co_network_changed_hint` function any time you know there has been a change in the network conditions.

If there is no easy way to know the network condition has changed, call the `cj_co_network_changed_hint` function every few seconds.

Compressing audio data

For audio the assumption is that the device gets a chunk of frames of raw PCM audio from a microphone.

Compressing with the Opus codec

Before transmission we recommend to first compress with the open-source Opus codec. It is widely used (including by WebRTC) and is considered better quality than AAC and others, all the while minimizing the complexity, bandwidth, and lag.

For the codec to work it needs chunks of audio frames of 10ms, 20ms, 40ms or 60ms. We recommend 20ms (or 40 if 20 not possible).

To compress a mono 20ms 48kHz audio chunk:

```
#include "opus/opus.h"

int err;
OpusEncoder *encoder = opus_encoder_create(48000, 1,
OPUS_APPLICATION_VOIP
, &err);
...
// Our chunk of PCM audio data
int16_t pcmChunk[960]; // 48000 Hz * 0.02 seconds = 960
…
unsigned char cbits[MAX_PACKET_SIZE]; // MAX_PACKET_SIZE should be
big enough to contain compressed audio chunk

int nbBytes = opus_encode(encoder, pcmChunk, 960, cbits,
MAX_PACKET_SIZE); // nBytes is the length of the encoded data chunk
…
```

Decompressing audio data

When receiving an audio chunk, it is typically compressed with a codec such as Opus, and so it first needs to be decoded.

Decompressing with the Opus codec

To decompress a mono 20ms 48kHz audio chunk:

```
#include "opus/opus.h"

int err;
OpusDecoder decoder = opus_decoder_create(48000, 1, &err);
…

// Our chunk of Opus audio data received from network
uint8_t audioData[AUDIO_DATA_SIZE];

opus_int16 outData[MAX_FRAME_SIZE];
int frame_size = opus_decode(decoder, audioData,
(opus_int32)AUDIO_DATA_SIZE, outData, MAX_FRAME_SIZE, 0);
```

At this point, the PCM audio is contained in outData and can be passed on to the audio player's `cj_ap_queue_audio_frames` function.

Building and linking

Before you can build, run the following command:

```
> sudo apt update; sudo apt install libopus-dev libssl-dev
portaudio19-dev
```

See build scripts in sample programs included in SDK for details on header and library search paths, as well as libraries to link to.