DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

University of Strathclyde Engineering

University of Strathclyde

WebRTC SFU server

Media server written in C over GStreamer framework
Final career project report

By

Roy Ros Cobo

Universitat de Girona

March 1, 2018

Index

0 Introduction

The project consists in building a basic **Selective Forward Unit (SFU)** server with **GStreamer** framework to serve **Web Real-Time Communications (WebRTC)** peers media streams. (Every technology will be explained ahead).

The WebRTC is meant to be used directly from one browser peer to another. The problem is when the number of peers increase, then probably browsers may not support that much media fluxes due these peers use to be mobile phones, laptops and other not powerful or specialized devices.
With a SFU this problem has less impact.

The idea of this SFU is that the peers don't have to send their streams to all of the other peers. They just communicate with the SFU server and it resend the media to all the other peers. Each peer will be sending 1 stream and receiving N-1 streams (where N is the total number of peers in the conversation).

The SFU server is built with a very new plug-in of the **GStreamer** framework; **GstWebRTC**, running on Linux OS. I have chosen GStreamer because is natively integrated on Linux and is always up to date with a good documentation and examples.

There are other SFU like Mediasoup, Licode, Janus, Kurento… but all of them are done to cover as many issues as possible, so they are not a "pure" SFU, they are in some way doing more functions than a basic SFU does.
If you want to learn how a server can handle WebRTC media by these servers its a huge coding space on it, most because of the debugging, the error control, etc. So my intention with this basic SFU is provide it without any extra capability or error control to make an understandable code, easy to learn with.
Also the real WebRTC apps use STUN and TURN servers to allow calls behind NAT networks but following with the main idea of the project I will not use this feature. This server and its peers will works only if all components are in the same network or has direct connectivity between them.
So the purpose of this project is not to be a competitive SFU but to be useful to learn how to work with this WebRTC technology on lower levels like C.

To make it work properly it needs more components than the server that I have developed.
It needs a **signalling server** working as intermediary of the server and the browser peers. It serves the **browser JavaScript app** and let the server and the peer communicate between a **WebSocket** connection where this signalling server works as a proxy.
This WebSocket connection needs a **WebSocket JavaScript Object Notation (JSON) protocol** to work with.
The last component is the **browser JavaScript app** to be the peer who provides the streaming to handle.

1 WebRTC

WebRTC is a free, open project that provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via **APIs**.

This project is being developed by Google and **World Wide Web Consortium (W3C)** to allow video and audio communication work inside web pages without the need to install plug-ins or download native apps.

This is actually supported by the most important browsers as Chrome, Firefox and Opera. WebRTC is being standardized by W3C.

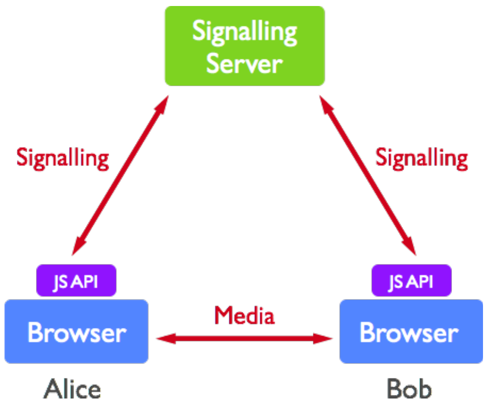


Figure 1. WebRTC environment diagram.

WebRTC API is meant to be used directly **peer to peer**.
There are many ways to handle the peer connections like rooms, broadcasts, multipartys… but a basic example would be like this:
Peer 1 turns on the JS app and connects to the signalling server and get registered in there via WebSocket connection. When peer 2 turns on and do the same, the signalling server tells to the peer 1 there is a new connected: peer 2.
When peer 1 get this information starts the negotiation through the signalling server with peer 2 to establish a direct connection between peer 1 and peer 2.

Once a connection has been established and opened, **media streams** and/or **data channels** can be added to the connection.
Media streams consists of any number of tracks of media information. This tracks can contain one type of media data for example **video, audio** or **text.**
Data channels can be used to exchange arbitrary binary data. This can be used for back-channel information, metadata exchange, text, etc.

1.1 WebRTC Architecture

WebRTC is an API to work with the **RTC** technology. This Real-Time Communications (RTC) technology works with **Real-time Transport Protocol** (**RTP**).

RTP is designed for end-to-end, real-time, transfer of streaming media. The protocol provides facilities for **jitter** compensation and detection of packet loss and out-of-order delivery, which are common during transmissions on an IP network.

The **RTP** specification describes two sub-protocols, **RTP** and **RTCP**. The data transfer protocol, **RTP**, facilitates the transfer of real-time data. Information provided by this protocol include timestamps (for synchronization), sequence numbers (for packet loss and reordering detection) and the payload format which indicates the encoded format of the data. **RTP** use the **VP8** codec to encode and decode data.

The control protocol, **RTCP**, is used for quality of service (QoS) feedback and synchronization between the media streams.

The majority of the **RTP** implementations works over **Interactive Connectivity Establishment** (**ICE**) alongside with **STUN** and **TURN** servers, this is built on the **User Datagram Protocol** (**UDP**) transport layer.

RTP sessions are typically initiated between communicating peers using a signalling protocol, such as (**Real-Time Signalling Protocol**) **RTSP**. These protocols may use the **Session Description Protocol** (**SDP**) to negotiate the parameters for the sessions. In this project **RTSP** is encapsulated on the **WebSocket JSON protocol** described on section 2. The JSON protocol of this project works over **WebSocket**, it works over **HTTPS** what is built on **TLS/SSL** and this one over **TCP** transport layer.

In the below figure 2 we can see the whole layer architecture for WebRTC on this project. On the left side of the figure there is the signalling protocol layers. On the right the media stream protocol layers.

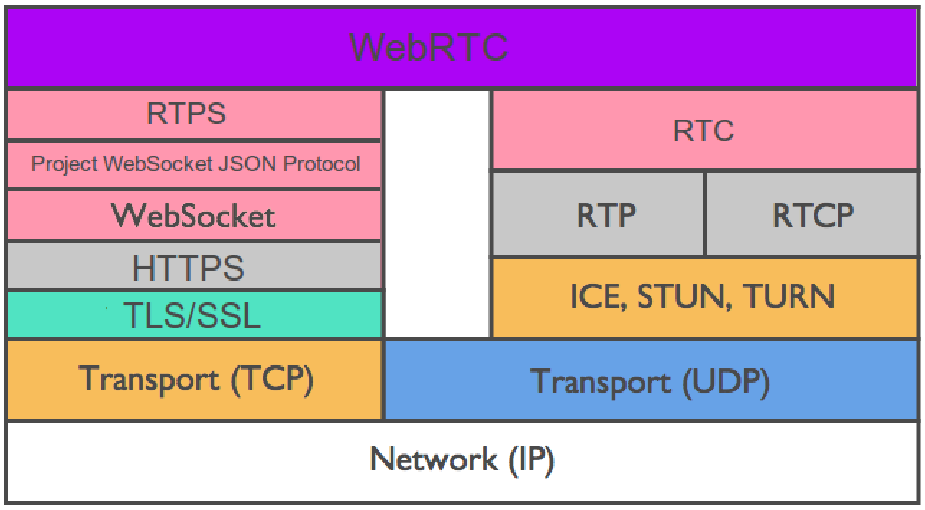


Figure 2. WebRTC architecture layers.

1.2 Objects and interfaces

1.2.1 Promise

The promise object represents the eventual completion (or failure) of the asynchronous operation, and its resulting value. The promise object represents a value what can be available now, in the future or never.
A promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A promise is in one of these states:

- **Pending:** initial state, neither fulfilled nor rejected.
- **Fulfilled:** meaning that the operation completed successfully.

- **Rejected:** meaning that the operation failed.

When a promise change its state from pending to either fulfilled or rejected the associated handlers queued up by a promise's `.then()` method are called.
The promise `.then()` and `.catch()` method returns a promise so they can be chained.

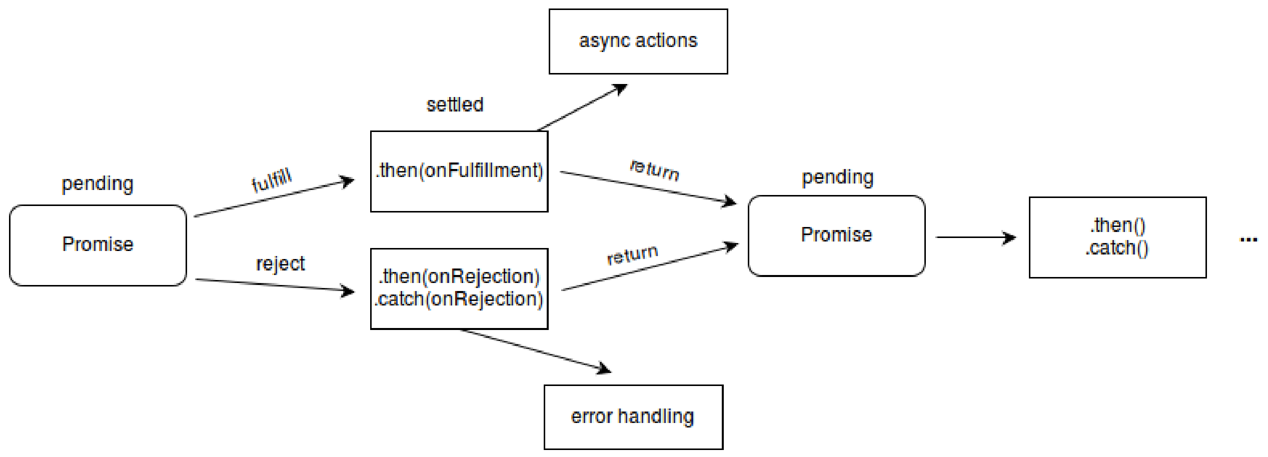


Figure 3. Promise Object flow diagram.

Without promise object would not be able to negotiate the WebRTC connection as there are many features that are asynchronous.

1.2.2 RTCSessionDescription

The RTCSessionDescription interface describes one end of a connection and how it's configured.
This interface is used in the peers connection negotiation being the way to set the **offer** and **answer** for each peer in the line. So each offer and answer is inside a RTCSessionDescription.
Each RTCSessionDescription consists of a description **type** indicating which part of the offer/answer negotiation process it describes and the **SDP** descriptor of the session.

This object is made in JSON to be able to be encapsulated on higher user's protocols.
The following is a real example of an offer RTCSessionDescription:

```
{
      type: "offer",
      sdp:  "v=0
             o=- 3295886605196749790 0 IN IP4 0.0.0.0
             s=-
             t=0 0
             a=ice-options:trickle
             m=video 9 UDP/TLS/RTP/SAVPF 96
             c=IN IP4 0.0.0.0
             a=setup:actpass
             a=ice-ufrag:YmeDxY55O/zxxlW6AaF558gmHp1LZKMT
             a=ice-pwd:tHdH7t3MdeOX9PjQLCmTLOuSV7qk+PeT
             a=sendrecv
             a=rtcp-mux
             a=rtcp-rsize
```

```
            a=rtpmap:96 VP8/90000
            a=rtcp-fb:96 nack
            a=rtcp-fb:96 nack pli
            a=mid:video0
            a=fingerprint:sha-256
            13:F7:09:62:CF:04:DE:EC:29:B4:56:E7:28:E5:D9:60:E4:66:F5:6F:2B:D8:7B
            :C9:51:3C:50:E7:CA:19:12:95"
}
```

1.2.3 RTCIceCandidate

The RTCIceCandidate interface represents a candidate **Internet Connectivity Establishment (ICE)** server which may be used to establish an RTCPeerConnection.

This object is sent like the RTCSessionDescription through the signalling server.
In a basic WebRTC app this object is not directly created as is only handled when the interface RTCPeerConnection fires its event what it means that a new RTCIceCandidate is created and need to be sent. Once is sent the other peer set it to its RTCPeerConnection.

1.2.4 MediaStream

The MediaStream interface represents a stream of media content. A stream consists of several **tracks** such as video of audio tracks. Each track is specified as an instance of **MediaStreamTrack**.

This component is added to the RTCPeerConnection interface to be established in the connection once is created.
This object is not usually created by its constructor, is more common to create it by calling the **`MediaDevices.getUserMedia()`** method what returns an MediaStream object.
Once is created you can add tracks to it by using the method **`MediaStream.addTrack()`** what fires its corresponding event.

It has three properties: **`MediaStrea.active, MediaStream.ended`** and **`MediaStream.id`**.

1.2.5 RTCPeerConnection

To handle the media connection between peers the WebRTC API provides the RTCPeerConnection interface. It represents a WebRTC connection between the local computer and a remote peer. It provides methods to connect, maintain, monitor and close the connection with the remote peer.

This is the main object for the WebRTC connection.
Its **constructor** is **`RTCPeerConnection(struct configuration)`** where the configuration parameter is optional and is used, for example, to set properties like which STUN and TURN servers must be used.

With it we can check the connection state with the property: **`RTCPeerConnection.connectionState`** what returns a specified string with the state connection.

Once is created we set the **event handlers** to set the behaviour when events are fired giving a function to be called when this happens.
The most important ones are:

- **`RTCPeerConnection.onicecandidate`**: This happens whenever the local ICE agent needs to deliver a message to the other peer through the signalling server.
- **`RTCPeerConnection.onnegotiationneeded`**: This event is fired when a change has occurred which requires session negotiation.
- **`RTCPeerConnection.ontrack`**: This event is sent when a new incoming `MediaStreamTrack` has been created and associated with the connection.

When the handlers are set then we add the MediaStream objects to it.

In this server there are many RTCPeerConnection objects without MediaStream objects due one peer can be receiving several streams meanwhile is sending only one. So this RTCPeerConnection object can have added a MediaStream objects in both ends, in one end, or no one.

Then to negotiate and set the descriptions we use the following **methods**:

- **`RTCPeerConnection.createOffer()`**: returns a promise with the local SDP offer created by the method. Then the SDP offer should be sent to the other peer.
- **`RTCPeerConnection.createAnswer()`**: has to be used when we have already received the remote offer and the remote description is set. Returns a promise with the local SDP answer created to be sent to the other peer.
- **`RTCPeerConnection.setLocalDescripction(RTCSessionDescription description)`**: used to set the local description what specifies the properties of the local end of the description.
- **`RTCPeerConnection.setRemoteDescripction(RTCSessionDescription description)`**: used to set the remote description what specifies the properties of the remote end of the description.
- **`RTCPeerConnection.addIceCandidate(RTCIceCandidate candidate)`**: when receives a new ICE candidate from the remote peer over its signalling channel, it delivers the newly-received candidate to the browser's ICE agent by calling this method.

1.3 Connection phases

There are several ways to establish a connection using this API as you can create multipartys, broadcasts, etc. I will explain how is it the standard connection way to connect two peers streaming audio and video.

This is also how I do it in the browser JavaScript app I have coded to use in this server.

The main phases of a basic WebRTC app connection are:

1. Create the RTCPeerConnection object to be able receive or send the media. When is created the event handlers should be attached to it so when the events are fired they can be treated.
   The minimum handlers to establish a connection are: **`RTCPeerConnection.onicecandidate`** and **`RTCPeerConnection.ontrack`**.

2. Get the user media requesting it to the current web-cam and/or microphone available on the device.
   This depends on the device and the browser the user is using but there is a standard of **`navigator.mediaDevices.getUserMedia(struct constraints)`** and this method returns a promise with the stream.
   (The constraints is a struct which defines which media has to be required to the device; used to be **`{video: true, audio: true}`**).

3. Connect to the signalling server (this could be done also as the first or second step).
   Most common way to do it is by the Node.js package **socket.io**.
   In my case the connection is by the **Web Socket Secure (WSS)** protocol.
   The WebSocket constructor is: **`WebSocket(string signalling_server_adress)`**. This object handles the connection with the signalling server. Once is created we have to attach its event handlers.
   The most important and the basic ones are two:

   - **`WebSocket.onmessage`**: This is fired when we receive a message from the signalling server. The function what handles it get the message as a parameter.
   - **`WebSocket.onclose`**: This is happens when the connection is closed.

   To send a message to the signalling server we use the method **`WebSocket.send(string message)`**.

4. **Negotiate** the connection. Once the previous phases are done the signalling server makes the peers know that they can start the negotiation.
   One peer send an **offer** through the signalling server and its set in the second peer. Then the second peer send the **answer** to the first peer and they start to exchange RTCIceCandidate objects.

5. Get the remote **stream**. When the negotiation is done and the connection is established must be fired the events with the MediaStream objects to get the remote streams and play it in the browser app.

2 WebSocket JSON protocol

There has to be a communication between the peers through the signalling server to establish connection.

This communication takes place when peers connects to the signalling server and the SFU server starts the connection.

This protocol is meant to be used in this project environment due in other place would make no sense.

The WebSocket secure connection is the way the peers have to communicate and this has to be done with text.

There is another technologies that allows to exchange JSON objects like the module Socket.io for Node.js but the GStreamer requires to use the WebSocket so its implemented with it.

The text sent is JSON objects converted to plain text, for example the JSON object **`{example: "hello", field2: -4}`** would be converted to: **`"{"example":"hello","field2":-4}"`**. JavaScript language has a native method to do it: **`stringify(JSONobject)`**, as it also has the method to do the opposite, transform text into JSON object, **`parse(string)`**.

Therefore the protocol precise how this JSON fields has to be to set an standard and let the different components understand each other.

The JSON object in this protocol contain the following fields:

- **type**: Specifying which kind of message we are sending. It can be:
  - **txt**: The data contains plain text, is made to exchange plain messages for debugging or other purposes.
  - **Id**: This message is sent when a new peer (excluding the GStreamer server) is connected and registered to the signalling server. The data contains a single integer number what is the ID of the new peer.
  - **gstServerON**: Is sent by broadcast (except the GStreamer server) when the SFU server connects to the signalling server so its already running and ready to attend connections. The data is meaningless the message in itself indicates that the server is running.
  - **socketON, socketOFF**: When a socket goes online or offline. Sent as a broadcast from the signalling server this message indicates that a socket is either online or offline. Data field is a JSON object with two string fields: **id** with the id and **ip** with the IP of the peer.
  - **offer**: This is a negotiation message what contains an RTCSessionDescription offer in the data field. This message requires to have a **index** field.
  - **answer**: This is a negotiation message what contains an RTCSessionDescription answer in the data field. This message requires to have a **index** field.

    - **candidate**: This is a negotiation message to exchange the candidates, the data contains a RTCIceCandidate. This message requires to have a **index** field.
- **data**: Here is the body of the JSON object and where the information is sent. This field can be just text or another JSON object so it lets to encapsulate other protocols.
- **From**: Specifies the ID of the peer who is actually sending the message.
- **To**: Specifies the ID of the peer who is the message receiver.
- **Index**: This field is only required in the negotiation messages and indicates the number of RTCPeerConnection (standard peers case) or the GstWebRTCBin (GStreamer server case) what the candidate is referencing to due they may contain several of them. It is an integer.

Each message must contain at least the fields type, data, from and to to be a valid message.
The user can define more types of message as long as respect the protocol.

3 Browser JavaScript App

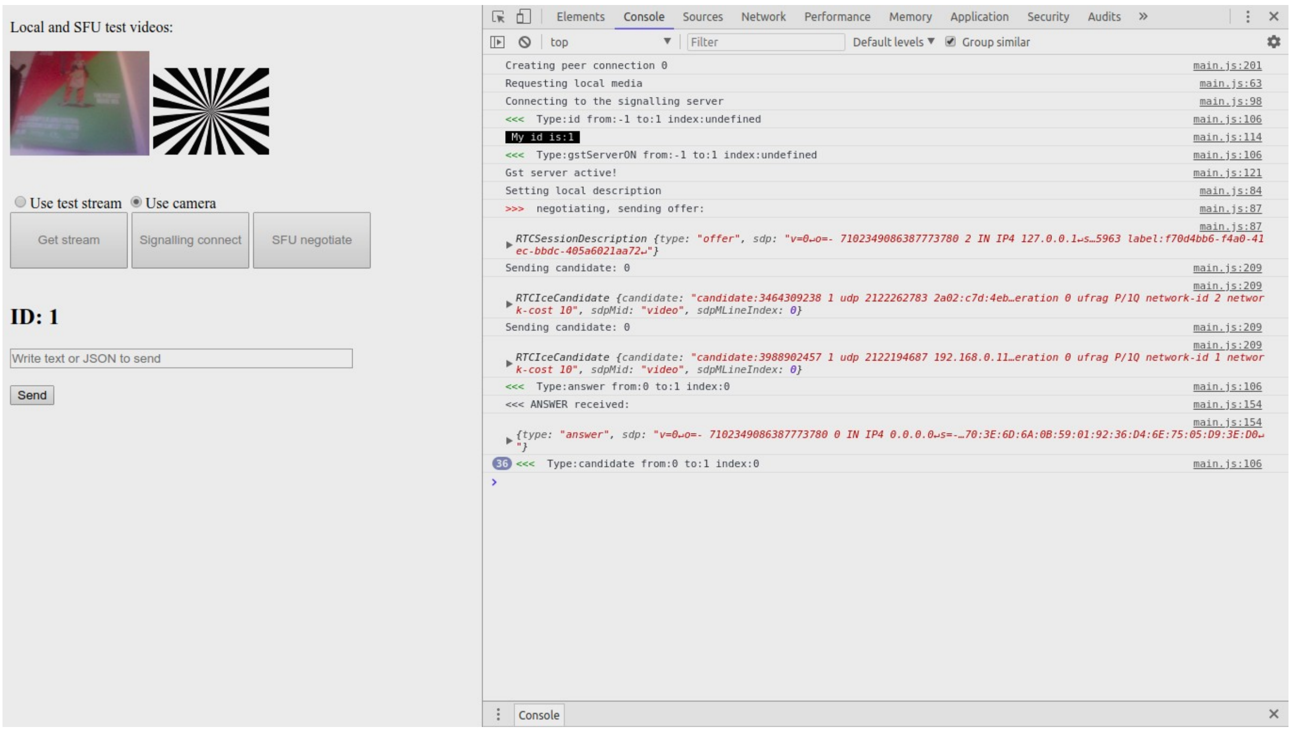
The **browser JavaScript App** is one of the main components of the project. This app is written in **JavaScript** and **HTML** using the Google **WebRTC API**. It works as a peer of the server following the phases described in the previous section.

It is a HTML web page with a JavaScript script what makes all the background running and connection required to be a peer.

This app is made to be served by the signalling by its HTTPS server. It works on a secure HTTP and also the WebSocket works in a secure way to guarantee the media and data is encrypted in the network as the WebRTC require.

Is meant to run over **Google Chrome browser** or **Mozilla Firefox browser**. This browsers are the ones who implements all the WebRTC technology and API to make the app works how has to be (can fail in other browsers due they may have not all the required WebRTC implementation).

In the figure 4 we can see how it looks the app on a Google Chrome browser after connect to the signalling server and the SFU server from who is receiving a video test:



```
Creating peer connection 0                                                        main.js:201
Requesting local media                                                            main.js:63
Connecting to the signalling server                                               main.js:98
<<<  Type:id from:-1 to:1 index:undefined                                         main.js:106
My id is:1                                                                        main.js:114
<<<  Type:gstServerON from:-1 to:1 index:undefined                                main.js:106
Gst server active!                                                                main.js:121
Setting local description                                                         main.js:84
>>>  negotiating, sending offer:                                                  main.js:87
                                                                                  main.js:87
▸ RTCSessionDescription {type: "offer", sdp: "v=0↵o=- 7102349086387773780 2 IN IP4 127.0.0.1↵s…5963 label:f70d4bb6-f4a0-41
  ec-bbdc-405a6021aa72↵"}
Sending candidate: 0                                                              main.js:209
                                                                                  main.js:209
▸ RTCIceCandidate {candidate: "candidate:3464309238 1 udp 2122262783 2a02:c7d:4eb…eration 0 ufrag P/1Q network-id 2 networ
  k-cost 10", sdpMid: "video", sdpMLineIndex: 0}
Sending candidate: 0                                                              main.js:209
                                                                                  main.js:209
▸ RTCIceCandidate {candidate: "candidate:3988902457 1 udp 2122194687 192.168.0.11…eration 0 ufrag P/1Q network-id 1 networ
  k-cost 10", sdpMid: "video", sdpMLineIndex: 0}
<<<  Type:answer from:0 to:1 index:0                                              main.js:106
<<< ANSWER received:                                                              main.js:154
                                                                                  main.js:154
▸ {type: "answer", sdp: "v=0↵o=- 7102349086387773780 0 IN IP4 0.0.0.0↵s=-…70:3E:6D:6A:0B:59:01:92:36:D4:6E:75:05:D9:3E:D0↵
  "}
36 <<<  Type:candidate from:0 to:1 index:0                                        main.js:106
>
```

Figure 4. Screenshot of the browser JavaScript app.

3.1 HTML

The HTML part of the app is really basic and is how the user interacts with the JavaScript app and act as a peer.

All the information messages are logged in the browser console which can be opened pressing F12. The console is useful to check all the messages fields and who are sending it and also for the messages we send. To enable some buttons or change variables we also need to use it so its an important part of this JavaScript app.

Is divided in 3 sections:

3.1.1 Video elements

A div with with, at first, two video elements where the media can be played. The original app plays its own web cam stream in the first one and the remote peer in the second one but this can be changed for test and debugging purpose. When the app is actually creating more than 2 RTCPeerConnection objects it create new video elements in the HTML to display the stream.

3.1.2 Buttons

Three button elements to interact with the JavaScript app.

- **Get stream:** This button request the selected stream and play it. If “Use camera” is selected request the web Cam media of the device is running on.
- **Signalling connect:** connects a WebSocket secure to the signalling server and it send back the **ID** it has assigned to us. When the signalling server deliver us the ID it send us a message if the SFU server is **online** to let us know. If its offline the server will send us this message when goes online.

  Is used if the peer is started before the SFU server. Could be also used to connect to another peer directly without using the SFU server.

  This button is only enabled if the SFU server is running so if we want to connect directly we need to enable via the browser console. It works only if we have got our ID.
- **Negotiate:** With the “Negotiate” button we start the negotiation with the SFU server..

3.1.3 Input field

There is an input to send messages as a JSON or as a text.

To send a JSON message it should start with the character **“{“** so the app take it as a JSON. The JSON messages should follow the defined WebSocket JSON protocol or the signalling server and the other peers will not understand and be discarded.

To send text messages just type the text and press enter or the send button, the message will be sent as a broadcast including the SFU server.
If the text starts with an integer, for example “3Example msg”, the message will be sent to the peer with the ID=3.
This is meant to debug and test the connection and the protocol between peers.

3.2 JavaScript script

The script of the JavaScript app is the implementation of the browser JavaScript app to interact with the signalling server and the SFU server as a peer. Here is the code and the logic of the app. It handle the HTML components logic running in the background using the WebRTC API.

The design of the WebRTC flow has been modified due this peer is made to connect to the SFU server. First request the web cam and connects to the signalling server, if the SFU server was already ON (is how it suppose to be) it sends to this peer an offer to start the negotiation. When another peer is turned ON the SFU server will send another offer to us to send us the other peer stream.

Each peer starts with a single RTCPeerConnection what sends the web cam stream and receive a video test. RTCPeerConnection objects are stored in a table by its **index**. The index is the number of the RTCPeerConnection (the first one to send our stream has index 0). When other peer is turned on, another RTCPeerConnection is created to receive the other peer stream.

The following are the signature of the functions used, each one matches with a button plus the private ones.

- **`void start():`** Create the first RTCPeerConnection (index 0) and request the web cam stream.
- **`void negotiate(int index, int to):`** Sends an offer to negotiate the index RTCPeerConnection with the peer with ID to.
- **`void connectSignServer():`** Create a connection to the signalling server with the URL specified in the variable **`web_socket_sign_url`** (the default is the same location as is in the current URL of the browser due this app should be served by the signalling server). In this function are handled the events **`onmessage`** and **`onclose.`** The **`onmessage`** event handles all possible messages types like txt, GstServerON or candidates.
- **`void createPeerConnection(int index, int to):`** Creates a new RTCPeerConnection object in the table with the index specified with the peer wit ID to. Here are set the event handlers **`onicecandidate, onnegotiationneeded`** and **`ontrack`**.
- **`HTML input field listener:`** This is not a method but acts as it. Takes the text in the input field and handle it to send it as a JSON or text with the defined WebSocket JSON protocol.

4 Signalling server

Signalling server is the proxy for the communication of the peers at the first time when the connection is not established yet. Its used to negotiate between peers and also serves the JavaScript App.
This has two functionalities, the main one is to be the gateway where the peers connect to and can communicate before any connection is established. Then this also serves the JavaScript app to be used as a peer.

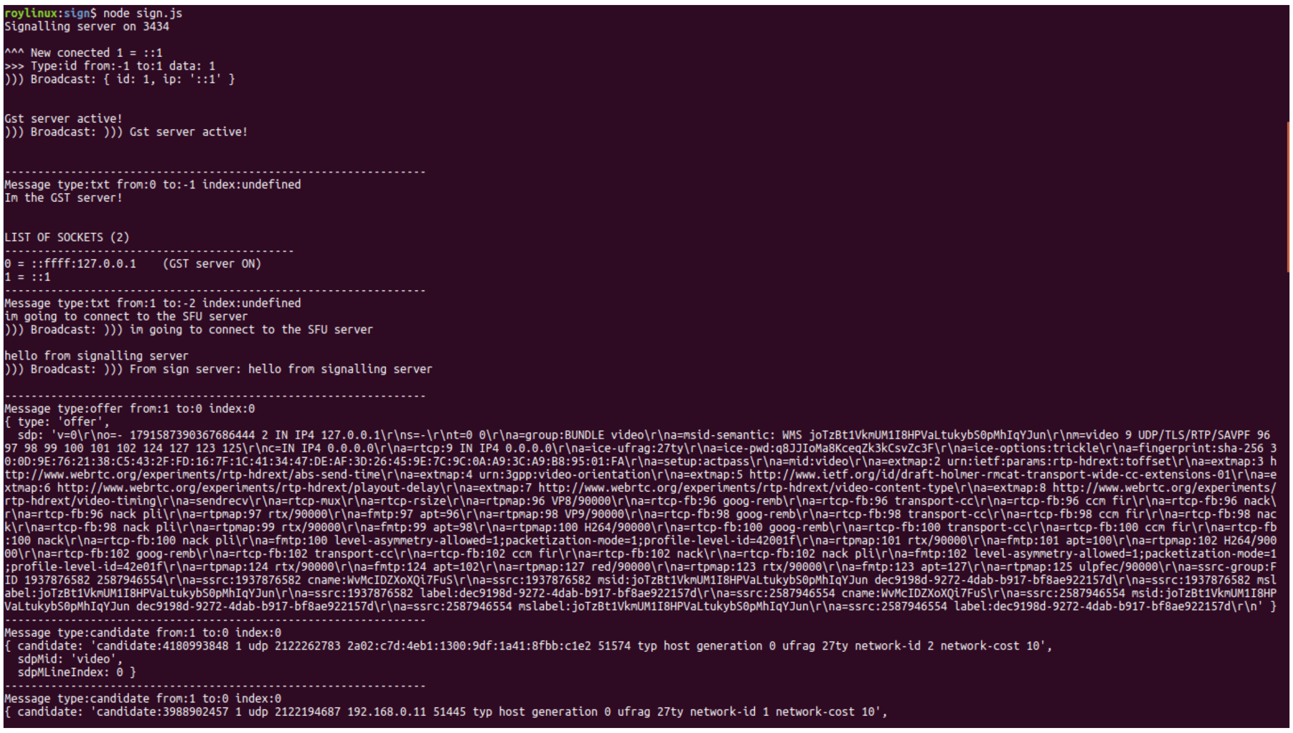
This server is made on **Node.js** runtime due it works on JavaScript and allows the communication with the peers easily by WebSocket secure technology that is the one GStreamer requires.

Node.js can be installed just executing the two following instructions in the command line terminal:

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -
sudo apt-get install -y nodejs
```

To run the signalling server run the command **`node [name of the sign.js file]`**.

The following figure 5 shows a caption of the signalling server after show the connected peers table, exchanging some text messages and a peer-server WebRTC negotiation:



```
roylinux:sign$ node sign.js
Signalling server on 3434

^^^ New conected 1 = ::1
>>> Type:id from:-1 to:1 data: 1
))) Broadcast: { id: 1, ip: '::1' }


Gst server active!
))) Broadcast: ))) Gst server active!


----------------------------------------------------------------
Message type:txt from:0 to:-1 index:undefined
Im the GST server!


LIST OF SOCKETS (2)
--------------------------------------------
0 = ::ffff:127.0.0.1    (GST server ON)
1 = ::1
----------------------------------------------------------------
Message type:txt from:1 to:-2 index:undefined
im going to connect to the SFU server
))) Broadcast: ))) im going to connect to the SFU server

hello from signalling server
))) Broadcast: ))) From sign server: hello from signalling server

----------------------------------------------------------------
Message type:offer from:1 to:0 index:0
{ type: 'offer',
  sdp: 'v=0\r\no=- 1791587390367686444 2 IN IP4 127.0.0.1\r\ns=-\r\nt=0 0\r\na=group:BUNDLE video\r\na=msid-semantic: WMS joTzBt1VkmUM1I8HPVaLtukybS0pMhIqYJun\r\nm=video 9 UDP/TLS/RTP/SAVPF 96
97 98 99 100 101 102 124 127 123 125\r\nc=IN IP4 0.0.0.0\r\na=rtcp:9 IN IP4 0.0.0.0\r\na=ice-ufrag:27ty\r\na=ice-pwd:q8JJIoMa8KceqZk3kCsvZc3F\r\na=ice-options:trickle\r\na=fingerprint:sha-256 3
0:0D:9E:76:21:38:C5:43:2F:FD:16:7F:1C:41:34:47:DE:AF:3D:26:45:9E:7C:9C:0A:A9:3C:A9:B8:95:01:FA\r\na=setup:actpass\r\na=mid:video\r\na=extmap:2 urn:ietf:params:rtp-hdrext:toffset\r\na=extmap:3 h
ttp://www.webrtc.org/experiments/rtp-hdrext/abs-send-time\r\na=extmap:4 urn:3gpp:video-orientation\r\na=extmap:5 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01\r\na=e
xtmap:6 http://www.webrtc.org/experiments/rtp-hdrext/playout-delay\r\na=extmap:7 http://www.webrtc.org/experiments/rtp-hdrext/video-content-type\r\na=extmap:8 http://www.webrtc.org/experiments/
rtp-hdrext/video-timing\r\na=sendrecv\r\na=rtcp-mux\r\na=rtcp-rsize\r\na=rtpmap:96 VP8/90000\r\na=rtcp-fb:96 goog-remb\r\na=rtcp-fb:96 transport-cc\r\na=rtcp-fb:96 ccm fir\r\na=rtcp-fb:96 nack\
r\na=rtcp-fb:96 nack pli\r\na=rtpmap:97 rtx/90000\r\na=fmtp:97 apt=96\r\na=rtpmap:98 VP9/90000\r\na=rtcp-fb:98 goog-remb\r\na=rtcp-fb:98 transport-cc\r\na=rtcp-fb:98 ccm fir\r\na=rtcp-fb:98 nac
k\r\na=rtcp-fb:98 nack pli\r\na=rtpmap:99 rtx/90000\r\na=fmtp:99 apt=98\r\na=rtpmap:100 H264/90000\r\na=rtcp-fb:100 goog-remb\r\na=rtcp-fb:100 transport-cc\r\na=rtcp-fb:100 ccm fir\r\na=rtcp-fb
:100 nack\r\na=rtcp-fb:100 nack pli\r\na=fmtp:100 level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=42001f\r\na=rtpmap:101 rtx/90000\r\na=fmtp:101 apt=100\r\na=rtpmap:102 H264/900
00\r\na=rtcp-fb:102 goog-remb\r\na=rtcp-fb:102 transport-cc\r\na=rtcp-fb:102 ccm fir\r\na=rtcp-fb:102 nack\r\na=rtcp-fb:102 nack pli\r\na=fmtp:102 level-asymmetry-allowed=1;packetization-mode=1
;profile-level-id=42e01f\r\na=rtpmap:124 rtx/90000\r\na=fmtp:124 apt=102\r\na=rtpmap:127 red/90000\r\na=rtpmap:123 rtx/90000\r\na=fmtp:123 apt=127\r\na=rtpmap:125 ulpfec/90000\r\na=ssrc-group:F
ID 1937876582 2587946554\r\na=ssrc:1937876582 cname:WvMcIDZXoXQi7FuS\r\na=ssrc:1937876582 msid:joTzBt1VkmUM1I8HPVaLtukybS0pMhIqYJun dec9198d-9272-4dab-b917-bf8ae922157d\r\na=ssrc:1937876582 msl
abel:joTzBt1VkmUM1I8HPVaLtukybS0pMhIqYJun\r\na=ssrc:1937876582 label:dec9198d-9272-4dab-b917-bf8ae922157d\r\na=ssrc:2587946554 cname:WvMcIDZXoXQi7FuS\r\na=ssrc:2587946554 msid:joTzBt1VkmUM1I8HP
VaLtukybS0pMhIqYJun dec9198d-9272-4dab-b917-bf8ae922157d\r\na=ssrc:2587946554 mslabel:joTzBt1VkmUM1I8HPVaLtukybS0pMhIqYJun\r\na=ssrc:2587946554 label:dec9198d-9272-4dab-b917-bf8ae922157d\r\n' }
----------------------------------------------------------------
Message type:candidate from:1 to:0 index:0
{ candidate: 'candidate:4180993848 1 udp 2122262783 2a02:c7d:4eb1:1300:9df:1a41:8fbb:c1e2 51574 typ host generation 0 ufrag 27ty network-id 2 network-cost 10',
  sdpMid: 'video',
  sdpMLineIndex: 0 }
----------------------------------------------------------------
Message type:candidate from:1 to:0 index:0
{ candidate: 'candidate:3988902457 1 udp 2122194687 192.168.0.11 51445 typ host generation 0 ufrag 27ty network-id 1 network-cost 10',
```

Figure 5. Terminal screenshot of the signalling server.

4.1 Requires

This server requires 4 module which can be installed using the **npm** module manager:

1. **ws:** This module allows to create a WebSocket secure server to let the peers connect to. In this channel is possible to send text and binary data but in this project the text data is the only one used.

As the peers works with JSON and this is a text channel there is the need to stringify all the JSON objects and this objects has to be done in the defined WebSocket JSON protocol.

2. **fs:** Provides an API to interact with the file system what is necessary to serve the browser JavaScript App.
3. **express:** Is a web application framework used to serve the web page using its function `static`.
4. **https:** Provides a way of making Node.js transfer data over **HTTP TLS**/**SSL** protocol, which is the secure HTTP protocol.

npm can be installed with the command **`sudo apt-get install npm.`**

To install the modules execute the command **`npm install [name of the module]`**.

4.2 IDs

Each peer has an ID provided by this server, the ID sent when the peer connects to this server. The IDs of the peers starts at 1. There is **special IDs**: -1 is the signalling server and 0 is the GStreamer server. When a peer is disconnected its ID is deleted and the ID is stored to be reused so the IDs does not goes to big numbers. When a peer is connected it gives the minimum not used ID.

4.3 Servers

There are a single server even though it acts as two.

To create the server we use the **`express.static`** utility to create an app which serves the files in the right folder where the browser JavaScript app is placed.

Then we create the HTTPS server using the app has been made with the module express to serve the files the express app can serve. As this server works in secure mode with the TLS/SSL protocol we need to provide the HTTPS server the credentials with the certificates to encrypt the data. This server is listening on the specified port (3434 by default).

Once is done a WebSocket server is created and attached to the already created HTTPS server so it uses the same certificates and is listening on the same port as the other server.

4.4 User input

To let the user interact with the socket peers using the signalling server terminal it has been implemented an Stdin listener to handles when the users write text in the server.

If the user press the Enter button with no text before a table with the active peers and its IDs and IPs is shown. Also display if the SFU server is online or not.

Following the browser JavaScript app HTML input field style, if the user types a text this is sent as a broadcast to all the socket peers connected including the SFU server. The text can be preceded by a number to send the text only to one single peer as in the browser JavaScript app HTML input field so “2Hello world” will send the message “Hello world” to the peer with the ID = 2.

4.5 Peer Sockets

This server has a **map** of sockets with key ID and its value is a struct which contains the socket connection, the IP of the peer and the ID (the field ID may seems redundant but is used for broadcasts for example).

The WebSocket secure connection is the main utility of this server as it allows the peers to establish the connection.
When a socket connects to this server the connection event of the WebSocket secure server is fired.
On this event it checks the origin of the connection request and if its from the SFU server. If the origin of the connection is the SFU server it fills the 0 position of the sockets map and sends a broadcast to all the sockets to let them know that the SFU servers is online. If the origin is a normal socket, get a new id for it and register the socket in the map, once is registered this signalling server sends the new socket ID to the socket and broadcast a “**socketON**” message.

In the connection event handler are set the two event handlers that a socket can fires.

1. **On message:** When a socket sends a message first is parsed to get it as a JSON object if its not in JSON notation an error is threw. The message is prompted and is resent the the socket what is specified in the field “to” of the message.
   A message may have different possible destinations:

   - **Normal peer:** > 0.
   - **SFU server:** 0.
   - **Signalling server:** -1.
   - **Broadcast:** -2.

   If there is no one of this destinations the message is discarded as an erroneous message.
   When the message has to be resent to a normal peer is not modified at all.

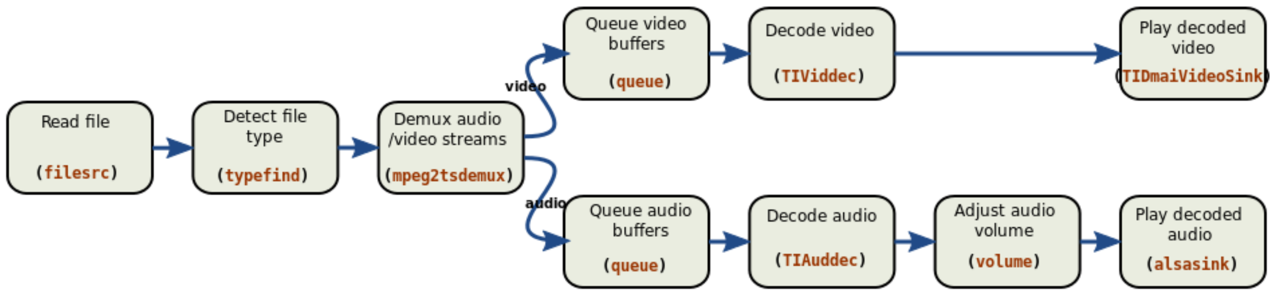
2. **On close:** When a socket is disconnected from this signalling server its IDs is deleted and pushed to the **savedIDs** table to re use it if its possible. The signalling server also sends a “**socketOFF**” message to let the peers handle the disconnection if its necessary.

5 GStreamer

GStreamer is a pipeline-based multimedia framework that links together a wide variety of media processing systems to complete complex workflows. GStreamer can be used by command line interface or by its C language framework and library to create more complex programs.

GStreamer supports a wide variety of media-handling components, including simple audio playback, audio and video playback, recording, streaming and editing. The pipeline design serves as a base to create many types of multimedia applications such as video editors, transcoders, streaming media broadcasters and media players.

GStreamer processes media by connecting a number of processing elements into a pipeline like the one we see
in the below figure 6. Each element is provided by a plug-in. Elements can be grouped into bins, which can be further aggregated, thus forming a hierarchical graph.
Elements communicate by means of pads. A source pad on one element can be connected to a sink pad on another. When the pipeline is in the playing state, data buffers flow from the source pad to the sink pad. Pads negotiate the kind of data that will be sent using capabilities.



```
gst-launch filesrc location="video.ts" ! typefind ! mpeg2tsdemux name=demux \
       demux. ! 'video/x-h264' ! queue ! TIViddec ! TIDmaiVideoSink \
       demux. ! 'audio/mpeg'   ! queue ! TIAuddec ! volume volume=5 ! alsasink
```

Figure 6. Simple pipeline diagram and command with GStreamer.

When the we use a GStreamer application the first we should use is the function **`gst_init()`** what Initializes the GStreamer library, setting up internal path lists, registering built-in elements, and loading standard plug-ins.
Then the most common work-flow would start connecting all the signals required for the program. Then creates the main loop and set it running. The program will start when some event fires a signal and end when one of this event handlers set the loop to quit.

It is designed to work on a variety of operating systems, e.g. Linux kernel-based operating systems, the BSDs, OpenSolaris, Android, macOS, iOS, Windows, OS/400. GNOME desktop environment of Ubuntu is a heavy user of GStreamer so the project is developed on a Ubuntu OS where GStreamer is installed natively and get the plug-ins even install them from files is more easy.

5.1 Plug-ins

GStreamer uses a plug-in architecture which makes the most of GStreamer functionalities implemented as shared libraries. Plug-in libraries get dynamically loaded to support a wide spectrum of codecs, container formats, input/output drivers and effects.

Plug-ins can be installed semi-automatically when they are first needed. For that purpose distributions can register a back-end that resolves feature-descriptions to package-names.

The plug-ins come grouped into three sets:

1. **Good**: This package contains the GStreamer plug-ins from the "good" set, a set of high quality plug-ins.
2. **Bad**: GStreamer Bad Plug-ins comprises a set of plug-ins not up-to-par compared to the rest. They might closely approach good-quality plug-ins, but they lack something: perhaps a good code review, some documentation, a set of tests, a real live maintainer, or some actual wide use.
3. **Ugly**: This package contains plug-ins from the "ugly" set, a set of poor-quality plug-ins that might pose distribution problems.

5.2 Installation

To install this plug-ins or the GStreamer on Linux its enough to type this command on the Linux console: **`sudo apt-get install gstreamer1.0-plugins-[plug-in set name]`** this will download and install the plug-in in the version of Ubuntu releases have actually.

On this project I am working with the plug-in “WebRTC” contained in the bad plug-ins set. As this plug-in is a really new element is only contained in the version 1.13.1 or later we need to install this plug-in manually by file due the Ubuntu releases version is older.

All versions of GStreamer and its plug-ins can be found here: https://gstreamer.freedesktop.org/src/

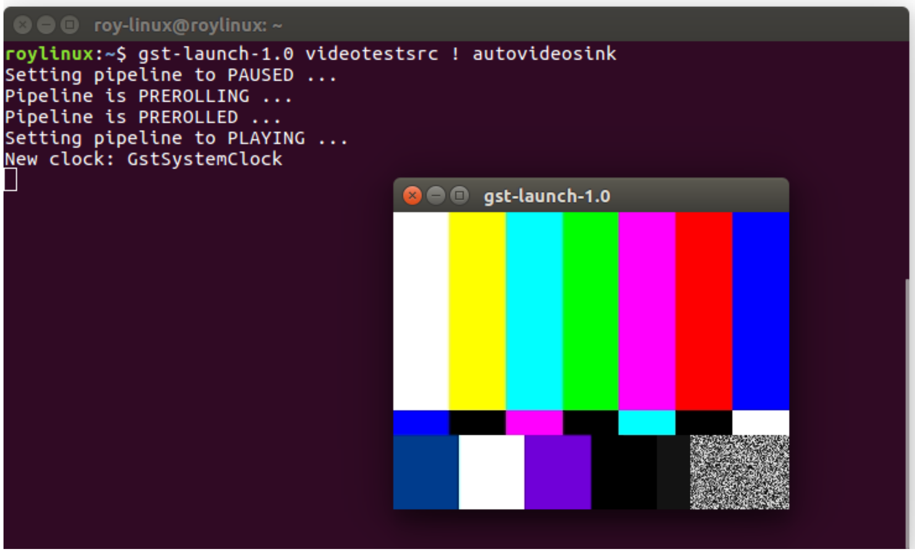
To install GStreamer and/or its packages manually by file we should download the compressed file and decompress it.
Once is done we open a terminal and change the directory to the folder will create the file when is decompressed. There we execute the script “configure” should be placed in there with the command **`./configure`**, when it ends we use the command **`sudo make`** and when it ends the command **`sudo make install`**. This should install the GStreamer or package we have downloaded by file.

To check if a plug-in is installed we can use the command **`gst-inspect-1.0 [name of the plugin]`**. If its installed it will show the plug-in information.

5.3 Elements and objects

Developing GStreamer code by a program using its library there is many objects that GStreamer just redefines to make it from its framework like the **`gint`** what is a normal **`int`** redefined. This distinctions is made to differentiate where is a component or element from the GStreamer environment. There is also objects what are more complex and make sense only in a GStreamer program.

In the below figure 7 there is a caption of the really basic GStreamer command **`gst-launch-1.0 videotestrc ! Autovideosink`** used to play a test video on a emergent window. This command is launched with **`gst-launch-1.0`** what initiates the GStreamer program. The elements in a created command pipeline are linked using "!". In this example the two elements are successfully linked due they are compatible and auto elements.



```
roylinux:~$ gst-launch-1.0 videotestsrc ! autovideosink
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
```

Figure 7. Ubuntu terminal and GStreamer window.

This is a list of the most important GStreamer objects and components I use in my GStreamer SFU server:

5.3.1 Basics

List of the basic types defined for easy-of-use and portability with the common types in C language they are referencing to:

- **`gint: int`**
- **`gchar: char`**
- **`gboolean: bool`**
- **`gpointer: operator "*"`**
- **`GstPromise: Javascript promise`** equivalent but to handle it needs additional methods implemented by GStreamer.
- **`GstWebRTCSessionDescription: JavaScript RTCSessionDescription`** equivalent but to handle it needs additional methods implemented by GStreamer.
-

5.3.2 GstPad

GstPad is an object contained by elements that allows links to other elements.

A GstElement is linked to other elements via "pads", which are extremely light-weight generic link points.

Pads have a **`GstPadDirection`**, **source** pads produce data, **sink** pads consume data.

A **`GstElement`** creating a pad will typically use the various **`gst_pad_set_*_function()`** calls to register callbacks for events, queries or dataflow on the pads.

Method **`gst_pad_get_parent()`** will retrieve the GstElement that owns the pad.

After two pads are retrieved from an element by **`gst_element_get_static_pad()`**, the pads can be linked with **`gst_pad_link()`**.

With **`gst_pad_is_linked()`** method its possible to checks if a pad is linked to another pad or not.

5.3.3 GstElement

GstElement is the abstract base class needed to construct an element that can be used in a GStreamer pipeline.

A GstElement can be created using the **`gst_element_factory_find()`** and **`gst_element_factory_create()`** functions to create element instances or **`gst_element_factory_make()`** as a shortcut.

An element name is an arbitrary name that can be used for the user to distinguish different elements of the same type. The name of a GstElement can be get with **`gst_element_get_name()`** and set with **`gst_element_set_name()`**.

Elements can have pads (of the type **`GstPad`**). These pads link to pads on other elements. **`GstBuffer`** flow between these linked pads. A GstElement has a **`GLis`**t of **`GstPad`** structures for all their input (or sink) and output (or source) pads. Core and plug-in writers can add and remove pads with **`gst_element_add_pad()`** and **`gst_element_remove_pad()`**.

Elements can be linked through their pads. If the link is straightforward, use the **`gst_element_link()`** convenience function to link two elements, or **`gst_element_link_many()`** for more elements in a row.

The the function **`gst_element_sync_state_with_parent()`** tries to change the state of the element to the same as its parent. If this function returns FALSE, the state of element is undefined.

5.3.4 GstStructure

Generic structure containing fields of names and values.
A GstStructure is a collection of key/value pairs.
Field values can be changed with **`gst_structure_set_value()`** or **`gst_structure_set()`**.
Field values can be retrieved with **`gst_structure_get_value()`**.

5.3.5 GmainLoop

The main event loop manages all the available sources of events. These events can come from any number of different types of sources such as file descriptors (plain files, pipes or sockets) and timeouts.

The GMainLoop data type represents a main event loop. A GMainLoop is created with **`g_main_loop_new()`**. After adding the initial event sources, **`g_main_loop_run()`** is called. This continuously checks for new events from each of the event sources and dispatches them. Finally, the processing of an event from one of the sources leads to a call to **`g_main_loop_quit()`** to exit the main loop, and **`g_main_loop_run()`** returns.

5.3.6 GstPipeline

This is one of the main components of a GStreamer program due all the elements which are going to be connect to let the dataflow goes through them will be in it.
Is possible to set more than one pipeline in the same program as this server that contain as many pipelines as peers are connected on it.

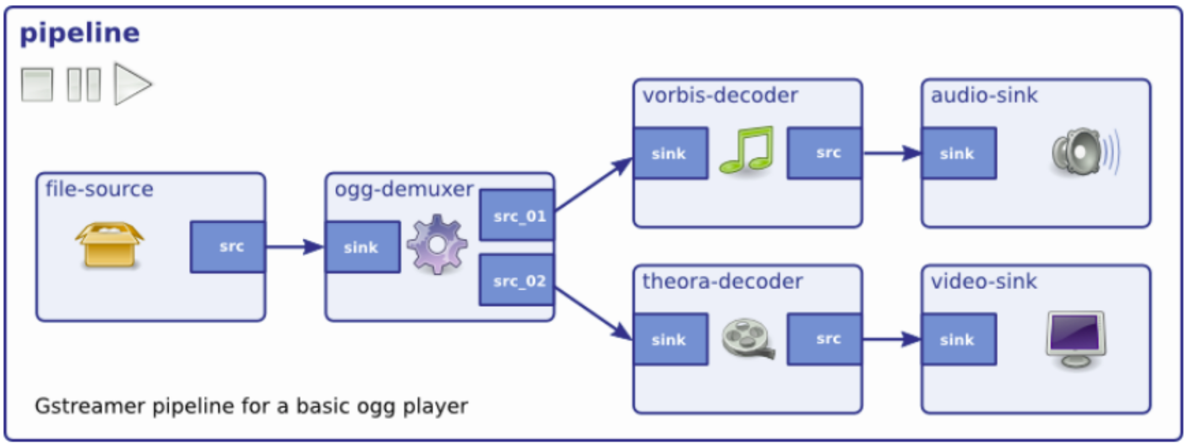


Figure 8. Example of pipeline with two branches.

A **`GstPipeline`** is a special **`GstBin`** used as the top-level container for the filter graph. The **`GstPipeline`** will manage the selection and distribution of a global **`GstClock`** as well as provide a **`GstBus`** to the application.

The method **`gst_pipeline_new()`** is used to create a pipeline. when you are done with the pipeline, use **`gst_object_unref()`** to free its resources including all added **`GstElement`** objects.
Elements are added and removed from the pipeline using the **`GstBin`** methods **`gst_bin_add()`** and **`gst_bin_remove()`**.
There is another way to create a pipeline when we are coding an application. The method **`gst_parse_launch()`** create a new pipeline based on command line syntax. We can give as a parameter the string **`"videotestsrc | autovideosink"`** and this will create a pipeline as we did in the figure 7.

Before changing the state of the **`GstPipeline`** a **`GstBus`** can be retrieved with **`gst_pipeline_get_bus()`**. This bus can then be used to receive **`GstMessage`** from the elements in the pipeline.

A GstPipeline maintains a running time for the elements. The running time is defined as the difference between the current clock time and the base time.

5.3.7 GstWebRTCBin

This component (**`webrtcbin`** in the applipcation) is so important on this project because is the way to connect the SFU server with the RTCPeerConnection objects of the peers. This object is the equivalent of the RTCPeerConnection in the JavaScript app but as this is C language is not so flexible and easy to use.

This element has, as the RTCPeerConnection on JavaScript, **event handlers** which are fired on specific events.
There is events attached on it in the SFU server:

- **on-ice-canidate**: Same event as JavaScript described previously.
- **on-pad-added**: Equivalent of **`ontrack`** in JavaScript but here there is a new source GstPad created with the media stream coming from the other end of the connection.
- **on-negotiation-needed**: Same event as JavaScript described previously. This event is fired when the **`webrtcbin`** goes to the state playing.
- **set-local-description**: in JavaScript this was a method from the object but in GStreamer is a signal with no callbacks what we give the **`GstWebRTCSessionDescription`** and the element to set the local description on it.
- **set-remote-description**: is the same event as **set-local-description** but used for the remote one.

This signals are attached to the **`webrtcbin`** component with the method **`gst_signal_connect()`** which takes as a parameters the **`webrtcbin`**, a string with the mentioned signal, the function to be called back when it gets fired and an optional **`gpointer`** with user data.

We can link a component to its static sink GstPad what will be received in the other end of the connection when the negotiation is done.

5.3.8 other objects

This are the description of smaller objects of GStreamer also used in the SFU server.

Videotestsrc: The videotestsrc element is used to produce test video data in a wide variety of formats. The video test data produced can be controlled with the "pattern" property.

Autovideosink: autoaudiosink is an audio sink that automatically detects an appropriate audio sink to use.

Fakesink: This sink just consume data but don’t use it to do nothing. Is used for testing and store fluxes meanwhile are processed or waiting or another event.

Queue: Data is queued until one of the limits specified by the “max-size-buffers”, “max-size-bytes” and/or “max-size-time” properties has been reached. Any attempt to push more buffers into the queue will block the pushing thread until more space becomes available.
The queue will create a new thread on the source pad to decouple the processing on sink and source pad.
The default queue size limits are 200 buffers, 10MB of data, or one second worth of data, whichever is reached first.

Tee: Split data to multiple pads. Branching the data flow is useful when e.g. capturing a video where the video is shown on the screen and also encoded and written to a file.
One needs to use separate queue elements (or a multi-queue) in each branch to provide separate threads for each branch. Otherwise a blocked dataflow in one branch would stall the other branches.

5.4 JSON-Glib

JSON-GLib is a library aimed at providing an API for efficient parsing and writing of JSON streams, using GLib's data types and API.

As the WebRTC and the signalling server uses JSON objects we need a way to handle this ones in the C language and to do so with this library.

To create a JSON object we use the method **`json_object_new()`** what returns the JSON object.

Then with this JSON object we can get and set members. The members can be of the type **`int,`** **`string`** or **`object`**. The object member is to set another JSON object so we can encapsulate JSON objects what is the main point of this notation.

To for example, set a **`int`** member we use the method **`json_object_set_int_member()`** and we give the JSON object we want to set the member on, the string for the key of the member and the **`int`** we want to set as a value. For the other types is the same process.

Then is also a process to parse a JSON object what when is wrote as a plain text and to do the opposite, turn a JSON object a plain text. There is no value on explain the methods on this process and in this project the are just encapsulated in two functions to make it easy.

5.5 LibSoup

LibSoup is an HTTP client/server library. It uses **`GObjects`** and the glib main loop and also has a synchronous API, for use in threaded applications.

In this project I use the module **SoupWebsocketConnection** to connect to the WebSocket server there is in the signalling server using the WebSocket protocol.

The flow for the connection is the following:

First we create a **`SoupSession`** with the method **`soup_session_new_with_options()`** where we provide options for the session like for example that it must use SSL as we need a secure connection. After that we create a **`SoupMessage`** with **`soup_message_new()`** ,where we provide the signalling server uri.

Then we call **`soup_session_websocket_connect_async()`** method what asynchronously creates a SoupWebsocketConnection to communicate with a remote server. We pass the **`SoupSession`**, the **`SoupMessage`**, a string with the origin (in this case is “gstServer”) to let the signalling server know, and a function to callback when the connection is done.

When this function is fired we call the method **`soup_session_websocket_connect_finish()`** to get a SoupWebsocketConnection that can be used to communicate with the server.

For final we set the GStreamer signals with **`g_signal_connect()`** to the connection for the signals **`message`** and **`closed`** with the functions to callback when this events are fired.

To send messages we use the method **`soup_session_websocket_connect_async()`**.

6 WebRTC SFU server

The purpose of this server is to provide a way to the peers to handle its connections more efficiently. When a peer is in a real-time stream room with multiple peers it has to send several fluxes and get the same number of it. With this server the peer only has to send one single flux and the server spread it to the other peers making lighter the data they have to handle and allowing a higher number of peers on a room.

6.1 WebRTC network topologies

This server is called SFU server because it uses the Selective Forward Unit topology to distribute the connections in the network. There is other ones to do so but I will explain the 3 main ones.

The 3 different topologies to distribute the real-time connections are:

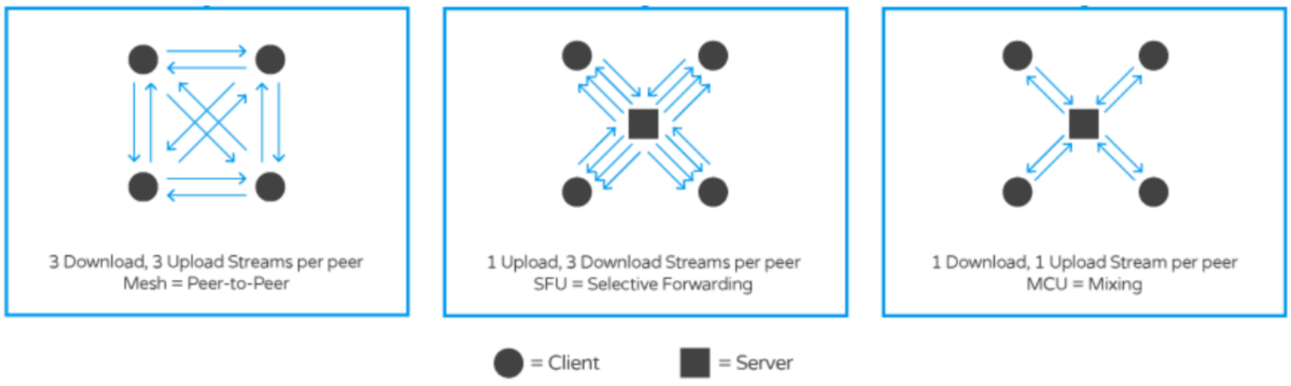

3 Download, 3 Upload Streams per peer
Mesh = Peer-to-Peer
1 Upload, 3 Download Streams per peer
SFU = Selective Forwarding
1 Download, 1 Upload Stream per peer
MCU = Mixing
● = Client
■ = Server

Figure 9. Three different topologies for 4 peers real-time connection.

6.1.1 Mesh (peer to peer)

This is the normal distribution for a WebRTC connection and for what this technology is meant.

For mesh distributions there is one advantage that is easy to implement and needs not much back-end infrastructure to operate. The problem is that it don’t scales well.

As we can see the topology for a 4 peers network in the figure 9 first picture that each peer has to send n-1 streams and receive the same amount of streams (where n is the number of participants).

This distribution requires a lot of uplink bandwidth from the peers. This peers used to be small devices as phones or laptops in a powerless network where.

The conclusion is that this distribution may works for small number of peers.

I have tested this topology on several networks and devices and the maximum number of peers it could handle without failures or missing data was 5.

6.1.2 SFU (Selective Forward Unit)

This is the type of this GStreamer server. This topology is perfect to solve the problem we have in the mesh distribution as it set the number of upload streams to 1. Networks as Wi-Fi or 3G/4G have the bigger connection speed on the download (around 85% download and 15% upload).

This kind of servers works as a proxy for the stream and they just resend the media to the right peer. The stream is not processed or changed in any way just split as many times as number of peers -1 there is in the network.

This takes helps the small devices with their small uplink bandwidth of the peers as they are only sending its media to the server and even though they still having to download the same number of streams as in the mesh distribution this is more efficient for the reason that download bandwidth is always many times higher.

The number of participants in this kind of network topology without failures, being the peers not powerful use to oscillate between 20 and 30. So this increase the 5 maximum peers of the mesh 5 times in average.

6.1.3 MCU (Multi-point Control Unit)

An MCU server takes all the incoming data streams and mix it together to a single one stream to send it to the peer. Each peer is sending 1 stream and receiving 1 stream.

The best of this kind of server is that allows to change the media fluxes, for example, if a device has a low download speed, the server can change the quality of the media is sending to it. There is also another features it can do as changing the media format or make it more efficient.

This allow a high number of peers in the distribution but also depends on some factors.
There is a need of a really powerful server to allow mix all streams meanwhile is serving the rest of the features. The back-end infrastructure and architecture requires to be quite big, expensive and complicated.

When a peer receives the stream is n-1 streams together so it has to separate and play it. This could also be an issue for the devices when there is a lot of peers in the connection. The number of maximum peers working well in this topology even its always bigger than mesh it also depends on the server efficiency and the CPU devices, so its around 25 and 55.

6.2 Structure

This is WebRTC server is made in the SFU topology distribution where each peer send a single media stream, it is received with the WebRTCBin GStreamer object what is the homologue of the RTCPeerConnection object in the WebRTC API.

For each peer there is a pipeline in the server which the peer owns. In this pipeline there many WebRTCBin objects but only one is from this peer. The others are owned by the rest of the peers if they exist. So for each peer there is initialized a pipeline and its WebRTCBin object which is used to receive the stream of the peer.

When a second peer is turned on another pipeline and WebRTCBin object is created for this second peer and a second WebRTCBin is created in both pipelines to send the two media streams from one to the other.

With this structure, taking P as the number of peers connected, there is P pipelines, and P^2 WebRTCBin objects. Each peer owns 1 pipeline with its WebRTCBin object inside, it also owns P-1 WebRTCBin objects more that are inside the other peers pipelines.

With this structure is true that each peer is sending a single media stream (in its pipeline with its WebRTCBin object) and receiving P-1 media stream (in the pipelines of the rest of the peers but owned by it). In the following figure 10 we can see a graphic of this structure.

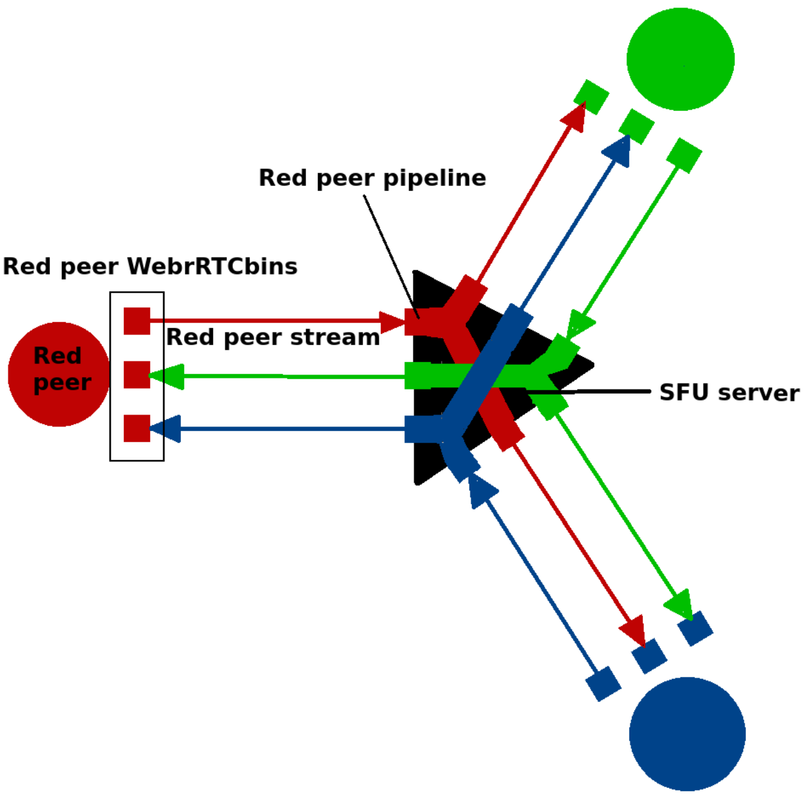


Figure 10. Diagram of the WebRTC SFU server with 3 peers.

6.3 Application

The server is written in C language in a single `.c` file where is the main and all the functions and objects to make it work. The application should work for two or more peers, even though that will depend on the development level of it.

It is made to run on Linux distribution but the only requirements is the GStreamer in the 1.13.1 minimum version.

The file can be compiled with the command **`gcc [name of the .c file] $(pkg-config --cflags --libs gstreamer-webrtc-1.0 gstreamer-sdp-1.0 libsoup-2.4 json-glib-1.0) -o [name of the output application]`**

There is no more command line interaction with the application server. It will show all the displayed messages by the command console.

6.3.1 Workflow

When the app goes to run, first we **`gst_init()`** as we mentioned before to initialize the GStreamer library, then we set up the peers structure where all application data about peers, WebRTCBin objects and pipelines is stored.

Then it connects to the signalling server by WebSocket Secure (WSS) connection to the URI we have provided in a previous declared constant. There is no confirmation for the signalling server part because we already got the method **`soup_session_websocket_connect_finish()`** that would tell if something went wrong.
In this connection we set two signals for this connection with the signalling server:

1. **Message**: When this event happens, we got a message from the signalling server, the function **`on_sign_message()`** is called back.
   In this function is where we handle all types of messages described in the protocol of the section 2. First we parse the message due it comes in plain text format, and we get the fields of the incoming JSON object.

2. **Closed**: This signal is fired when the signalling server is closed, the callback function for this event is **`on_sign_closed()`** where we just reset the peers structure and quit the main loop.

When a peer is turned on signalling server broadcast a **`socketON`** message with the new peer id. This will trigger the message signal we set previously and then the server will register the peer in the peers structure with its id.
As each peer has its own pipeline when its registered it calls a method that creates the pipeline and initialize it with a WebRTCBin inside, both are owned by the new peer.
In the new WebRTCBin of the peer we set the 3 signals to handle the possible events on it:

1. **`on-ice-candidate`**: This signal is called when there is an incoming ice candidate to be handled on the WebRTCBin, the callback function is **`send_ice_candidate()`** this function will send the candidate to the peer to let it set in it respective RTCPeerConnection object.
2. **`pad-added`**: When the peer sends a stream via its RTCPeerConnection, this stream is received in this WebRTCBin and this signal is fired indicating that a new **`src_pad`** has been added so we can connect it to handle the media flowing through it.
3. **`on-negotiation-needed`**: This signal indicates that some change has been done in a WebRTCBin, for example it state has changed to playing, and its required to negotiate a new connection with the peer. The callback function is **`negotiate()`** what creates a offer to send to the peer and start the proper negotiation.

When this is done, the pipeline of the peer is put to playing state and this will trigger the signal **`on-negotiation-needed`**, then the server will send an offer to the peer and this one must return an answer to negotiate the parameter and features of the connection.

When we receive the answer we will set it with the previous explained method and the same with the ICE candidates. The answer and the ICE candidates comes through the signalling server and we receive as a message.
When the negotiation is done, the peer WebRTCBin object must trigger the signal **`on-pad-added`** and there is where it connects the incoming **`src_pad`** with the required components like, queues and decoders and then to a **`tee`**. Here is where the stream is conduced to a **`fakesink`** where the flux will be stored till a new peer is connected.

In this moment of the application there is only one peer and its media stream flux is stored in the fakesink waiting to be split to another component. This will occur when a new peer connects to the server, when this happened after all the connection and negotiation the new peer must trigger the **`on-pad-added`** signal. When this signal is triggered we will create a new branch on the first peer tee and connect it to a new WebRTCBin in its pipeline, this new object will start a negotiation with the peer 2 and this way the peer 2 will receive the media of the peer 1 through this new WebRTCBin created.
At the same time the **`src_pad`** with the stream of the peer 2 will be connected to a tee and the first branch of it, instead to connect it to a fakesink, will be connected to a new WebRTCBin in the pipeline of the peer 2. The new WebRTCBin in the pipeline 2 with the media of the peer 1 will start a negotiation with the peer 1.

This events will be done when a 3th peer is turned on and each first WebRTCBin will be connected to a tee with many branches as many peers-1 are online. This way its created the structure described in the section 6.2.

6.3.2 Code, objects and methods

This section is to explain each object structure, method and others in the application code.

Includes

- **`sdp.h`**: Library to handle the SDP protocol for the negotiation.
- **`webrtc.h`**: This include is preceded by a definition of **`GST_USE_UNSTABLE_API`** what indicates that this API is not totally tested as I said is a very new component on GStreamer. This include will give us the methods to work with the WebRTCBin objects.
- **`soup.h`**: Library to use the WebSocket service and be able connect to the signalling server with the indicated methods. It also provides the secure mode which is used in this project.
- **`json-glib.h`**: This library provides the methods to work with JSON objects and create them. Also to “parse” and “stringify” plain text JSON objects.

Structs

There is 3 structs to handle all objects and peers in this server:

1. **`struct Wrbin{`**

```
    GstElement *wrbin;

    GstPad *sinkPad;
    GstPad *srcPad;

    gint ownerPeer;
    gint ownerPipe;

    gboolean negotiated;
};
```

This struct is made to store the WebRTCBin objects, this struct is contained by the following peer struct, its fields are:

- **`*wrbin`**: A pointer to **`GstElement`**, this will point to the WebRTCBin object. Default value is **`NULL`**.
- **`*sinkPad`**: With a **`GstPad`** pointer that points to the sink pad where is introduced the media in this WebRTCBin object. Default value is **`NULL`**.
- **`*srcPad`**: With a **`GstPad`** pointer that points to the source pad which appears when a new stream is coming from the peer. Default value is **`NULL`**.
- **`ownerPeer`**: This is an integer with the ID of the peer who owns this WebRTCBin object. Default value is 99.
- **`ownerPipe`**: This is an integer with the ID of the peer who owns the pipeline where this object is inside. Note that this field don’t have to be the same as **`ownerPeer`** due for example a second peer is connected the **`ownerPeer`** could be 1 because is owned by peer 1 but this field could be 2 because this is inside the pipeline from peer 2. This would mean that this WebRTCBin is made to receive the stream of peer 2 to peer 1. Default value is 99.
- **`negotiated`**: This field may not be used. It indicates if the WebRTCBin has negotiated with a peer. Default value is **`FALSE`**.

2. **`struct Peer{`**

```
    gint id;

    GstElement *pipel;

    gint nwrbins;
    struct Wrbin wrbins[16];

};
```

This struct stores the peers and its information, also stores the created pipelines and its WebRTCBin objects, its fields are:

- **`id`**: Integer with the peer id. This value comes from the signalling server who is the one who assign a id to the peers. Default value is 99.
- **`*pipel`**: A **`GstElement`** pointer object what points to the pipeline this peer owns. Default value is **`NULL`**.

- **`Nwrbins`**: Integer with the actual number of WebRTCBin objects this peer owns. Default value is 0.
- **`wrbins[16]`**: This is an array of the previous mentioned **`struct Wrbin`** with capacity for 16 items. Here are stored the WebRTCBin inside the **`struct Wrbin`** that this peer owns.

3.
```
typedef struct UserData{

    gint peerID;

    gint index;

} userData;
```

This is an struct with a **`typedef`** which defines the alias **`userData`**. This is done because in the methods of the GStreamer, LibSoup and JSON-GLib libraries its required to pass a pointer to the callback methods to give the User Data parameter.
Then if a the user would want to give a simple integer it would require to give a pointer to that integer, and this struct is to do so.

Its fields are:

- **`peerID`**: With an integer with the id of the peer we want to select.
- **`Index`**: With and integer of the index of the WebRTCBin object we want to select.

Constants

In the constants section of this code there is **`gchar* url_sign_server`** which must contain the signalling server URL.

Then two integers for the maximum number of peers and WebRTCBin for peer, in this case is **`MAX_PEERS=8, MAX_WRBINS=16`**.

Two **`#define`** with the pipe for encode the stream and link it to a WebRTCBin, this only stores a string to make the code shorter when the pipelines are made. One for video and one for audio.

Finally 4 **`UserData`** structs with all possible combinations for the peers. This is required due the callback functions require a pointer data so it has to be a pointer pointing to global data and this one being constant.

Variables

A special type for the main loop **`GmainLoop*`** that starts **`NULL`** and will point to the loop once it is created. This loop will be running and seeking for new events till some event make it quit.

Then the variable to handle the connection with the signalling server. **`SoupWebsocketConnection*`** is a type to store the connection with the serve which is used to send and receive the messages.

And the peers struct what consists on a integer with the current number of peers **`npeers`** that starts from 0 and the 8 peers array of Peer struct to store the peer information.

Methods

This is the signature and brief explanation of the SFU server application:

Initialisation

- **`userData* getConstUsDa(gint peerID, gint index)`**: Return the const userData struct with the values given. Is used to use the same object in the whole code.
- **`void init_peers(int id)`**: Initializes the peer with ID id of the array of peers to default values. To initialize the whole peers array id must be 0. This is necessary due C variables default value is undefined.

JSON methods

- **`gchar* json_stringify (JsonObject *object)`**: Given a JSON object returns the equivalent in plain text in a gchar pointer.
- **`JsonObject* json_parse(gchar *msg)`**: Is the opposite of the previous method. Return a JSON object given a JSON in plain text format.
  The **`msg`** parameter must be a correct JSON object in plain text format.
- **`void send_data_to(gchar *type, JsonObject *dataData, gint to, gint index)`**: This method sends a message to the signalling server according with the WebSocket JSON protocol described in section 2.

  The parameters are, the **type** of the message, the **JSON object** that will fit in the data field, the **id** of the destination peer and the **index** of the WebRTCBin object is referring the message to.

Negotiation

- **`void send_ice_candidate(GstElement * webrtc, guint mlineindex, gchar * candidate, userData *usDa)`**: Send an ice candidate to the peer with the id stored in the usDa parameter with the index stored in the usDa parameter.

  The first parameter is not used. The mLineindex and candidate are provided by the function who call this method due its a callback method.
- **`void on_offer_created(GstPromise * promise, userData *usDa)`**: This callback method is fired when an offer is created by a promise WebRTCBin object and set the local offer description and sends it to the peer according with the values in the usDa parameter.
- **`void on_answer_created(GstPromise * promise, userData *usDa)`**: This callback method is fired when an answer is created by a promise WebRTCBin object and set the local answer description and send it to the remote peer according with the values in the usDa parameter.
- **`void negotiate(GstPromise *promise, userData *usDa)`**: This method is called to starts a negotiation with a remote peer and a WebRTCBin according with the values in the usDa parameter. The first parameter promise pointer is not used.

Pad handlers

- **`void fake_link_srcpad(GstElement *webrtc, GstPad *new_pad, userData *usDa)`**: This function must be used for the first peer to connect, this links the incoming source pad with the first peer stream to a fakesink to store it meanwhile is not other peer where send it to.
  The first parameter webrtc is not used.

- **`void play_from_srcpad(GstElement *webrtc, GstPad *new_pad, userData *usDa)`**: This function is not used in the normal behaviour of the server but for testing and debugging. Creates an emergent window with the stream received from the peer.
  The first parameter webrtc is not used.

- **`void link_newSrcPad_to_newWrbin(gint newWrbin_ownerPeer, gint newWrbin_ownerPipe)`**: This function creates a new branch in the pipeline of the peer with ID newWrbin_ownerPipe and put the a new WebRTCBin object in it.
  Takes the src_pad of the first WebRTCBin object of the peer with ID newWrbin_ownerPipe and link it with the new WebRTCBin object. This object is put in the array of WebRTCBin objects of the peer with ID newWrbin_ownerPeer and this will start a negotiation for that new WebRTCBin.
  For example if the function is link_newSrcPad_to_newWrbin(1,3), will connect the stream of the peer 3 with a new connection to the peer 1.

- **`void on_pad_added(GstElement *webrtc, GstPad *new_pad, userData *usDa)`**: When a WebRTCBin creates a new source pad with the stream of a peer this is the method to callback. The new pad is **`new_pad`** and the peer and WebRTCBin object are stored in the usDa values.
  Here is where it has to be called the desired functions above to handle the new incoming src_pad.

- **`void set_wrbin_pads(userData *usDa)`**: This set the 3 signals described in the section 6.3.1 for a WebRTCBin. The peer and the WebRTCBin object are get according with the usDa parameter.

Pipeline

- **`void start_pipeline(userData *usDa)`**: This function starts a pipeline from the peer with id given in the usDa parameter and the first WebRTCBin to get the media from the peer. The pipeline is put to play state.

Signalling server connection

- **`void on_sign_message (SoupWebsocketConnection *ws_conn, SoupWebsocketDataType dataType, GBytes *message, gpointer user_data)`**: This is the callback function for an incoming message from the signalling server. Here the server handles which type type of message is according to the WebRTCBin JSON protocol described in section 2 and act consequently.

The **`ws_conn, dataType`** and **`*message`** parameters are provided by the event that fires this function. **`user_data`** parameter is not used.

- **`void on_sign_closed (SoupWebsocketConnection *ws_conn, gpointer user_data)`**: This just quit the loop. Is a callback when the signalling server closes the connection.
  Any parameter is used.

- **`void on_sign_server_connected (GObject *object, GAsyncResult *result, gpointer user_data)`**: This is a callback to end the connection with the signalling server. Sets the connection global variable. The **`*object`** and **`*result`** parameters are given by the event that fires this function.
  user_data parameter is not used.

- **`void connect_webSocket_signServer()`**: This starts the connection with the signalling server.

Main

- **`int main(int argc, char *argv[])`**: The main method of the application. Here starts the GStreamer intern mechanisms, set the peers array to the default values, starts the connection with the signalling server and creates and starts the main loop.

  When the loop quits, set peers array to default value, unref the signalling server connection and the loop objects.

References

- WebRTC (2017). WebRTC official webpage [Online]. Retrieved from https://webrtc.org/
- Mozilla MDN web docs (April 3, 2018). WebRTC API - Web APIs | MDN [Online]. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
- WIKIPEDIA (January, 2018). Wikipedia – WebRTC [Online]. Retrieved from https://es.wikipedia.org/wiki/WebRTC
- BLOCKGEEK (March 7, 2016). WebRTC Multiparty Video Alternatives, and Why SFU is the Winning Model [Online]. Retrieved from https://bloggeek.me/webrtc-multiparty-video-alternatives/
- GNOME(2018). GNOME developer center [Online]. Retrieved from https://developer.gnome.org/
- GNOME (2018). GNOME Wiki! [Online]. Retrieved from https://wiki.gnome.org/

Code

All the code files can be found on the following GitHub link: https://github.com/RosCoCorp/Gstreamer-WebRTC-SFU