
Kurento Documentation

Release 6.7.1

Kurento

Mar 22, 2018

User Documentation

1 About Kurento and WebRTC	3
1.1 WebRTC media servers	3
1.2 Kurento Media Server	4
1.3 Kurento Design Principles	4
2 About OpenVidu	7
3 Getting Started	9
4 Installation Guide	11
4.1 Amazon Web Services	11
4.2 Local Installation	12
4.3 STUN and TURN servers	12
4.4 Check your installation	14
5 Installing Pre-Release Builds	15
5.1 Kurento Media Server	15
5.2 Kurento Java Client	16
5.3 Kurento JavaScript Client	18
6 Kurento Tutorials	19
6.1 Hello World	19
6.2 WebRTC Magic Mirror	45
6.3 RTP Receiver	68
6.4 WebRTC One-To-Many broadcast	69
6.5 WebRTC One-To-One video call	93
6.6 WebRTC One-To-One video call with recording and filtering	118
6.7 WebRTC Many-To-Many video call (Group Call)	136
6.8 Media Elements metadata	145
6.9 WebRTC Media Player	152
6.10 WebRTC outgoing Data Channels	163
6.11 WebRTC incoming Data Channel	171
6.12 WebRTC recording	183
6.13 WebRTC repository	198
6.14 WebRTC statistics	209
7 Features	215

7.1	Kurento API, Clients, and Protocol	215
7.2	Kurento Modules	217
7.3	RTP Streaming	217
7.4	Congestion Control (REMB)	217
8	Configuration Guide	221
8.1	Media Server	221
8.2	MediaElement	221
8.3	SdpEndpoint	222
8.4	WebRtcEndpoint	222
8.5	HttpEndpoint	222
8.6	Debug Logging	222
9	Writing Kurento Applications	223
9.1	Global Architecture	223
9.2	Application Architecture	224
9.3	Media Plane	227
10	Writing Kurento Modules	229
10.1	OpenCV module	230
10.2	GStreamer module	230
10.3	For both kind of modules	230
10.4	Examples	232
11	Support	233
11.1	Usage Questions	233
11.2	Community Support	233
11.3	Commercial Support	234
12	Frequently Asked Questions	235
12.1	How To	235
12.2	Why do I get the error	237
13	Kurento API	239
13.1	Media Elements and Media Pipelines	239
13.2	Endpoints	240
13.3	Filters	242
13.4	Hubs	243
14	Kurento Client	245
14.1	Kurento Java Client	245
14.2	Kurento JavaScript Client	245
14.3	Reference Documentation	245
15	Kurento Protocol	247
15.1	JSON-RPC message format	248
15.2	Kurento API over JSON-RPC	249
15.3	Network issues	254
15.4	Example: WebRTC in loopback	255
15.5	Creating a custom Kurento Client	257
16	Kurento Modules	259
16.1	Module Tutorial - Pointer Detector Filter	260
16.2	Module Tutorial - Chroma Filter	280
16.3	Module Tutorial - Crowd Detector Filter	295

16.4	Module Tutorial - Plate Detector Filter	313
17	Kurento Utils JS	327
17.1	Overview	327
17.2	How to use it	327
17.3	Examples	328
17.4	Using data channels	329
17.5	Reference documentation	330
17.6	Souce code	333
17.7	Build for browser	334
18	Endpoint Events	335
18.1	MediaObject events	336
18.2	MediaElement events	336
18.3	BaseRtpEndpoint events	337
18.4	WebRtcEndpoint events	338
18.5	Sample sequence of events: WebRtcEndpoint	339
19	NAT Traversal	343
19.1	WebRTC with ICE	343
19.2	RTP without ICE	343
20	Securing Kurento Applications	347
20.1	Securing client applications	347
20.2	Securing server applications	349
21	WebRTC Statistics	351
21.1	Introduction	351
21.2	API description	351
21.3	Example	353
22	Debug Logging	355
22.1	Logging levels and components	356
22.2	Suggested levels	356
22.3	3rd-party libraries	358
23	Contribution Guide	359
24	Team	361
25	Code of Conduct	363
26	Kurento Business Features	365
26.1	Commercial Support	365
27	Developer Guide	367
27.1	Introduction	368
27.2	Development tools	368
27.3	Source code repositories	369
27.4	Development 101	372
27.5	Working with KMS sources	373
27.6	How-To	382
28	Continuous Integration	385

29 Release Procedures	387
29.1 Introduction	387
29.2 General considerations	388
29.3 Project Inventory	388
29.4 C/C++ modules	389
29.5 Java modules	389
30 Security Hardening	391
30.1 Hardening validation	392
30.2 Hardening in Kurento	392
30.3 PIC/PIE in GCC	392
30.4 PIC/PIE in CMake	393
31 Writing this documentation	395
31.1 Kurento documentation Style Guide	396
31.2 Sphinx documentation generator	397
31.3 Read The Docs builds	397
32 Congestion Control (RMCAT)	399
32.1 Google Congestion Control	399
32.2 REMB	400
33 NAT Types and NAT Traversal	401
33.1 Basic Concepts	402
33.2 Types of NAT	403
33.3 NAT Traversal	404
34 Glossary	407
35 Indices and tables	415

Kurento is a [WebRTC](#) Media Server and a set of client APIs that simplify the development of advanced video applications for web and smartphone platforms. Its features include group communications, transcoding, recording, mixing, broadcasting and routing of audiovisual flows.

The code is open source, released under the terms of [Apache License Version 2.0](#) and available on [GitHub](#).

You can read more on our page [About Kurento and WebRTC](#). Check now how to [get started](#) with Kurento and take a look at our [tutorials](#), which showcase some demo applications.

The main documentation for the project is organized into different sections:

- [User Documentation](#)
- [Feature Documentation](#)
- [Project Documentation](#)
- [Business Documentation](#)

Information about development of *Kurento itself* is also available:

- [Developer Documentation](#)

CHAPTER 1

About Kurento and WebRTC

Kurento is a [WebRTC](#) Media Server and a set of client APIs that simplify the development of advanced video applications for web and smartphone platforms. Its features include group communications, transcoding, recording, mixing, broadcasting and routing of audiovisual flows.

The code is open source, released under the terms of [Apache License Version 2.0](#) and available on [GitHub](#).

Kurento follows an architecture based on composable modules that can be mix-and-matched, activated, or deactivated at any point in time. Developers can create additional modules to add new functionalities that will be able to be plugged-in dynamically.

With Kurento, it's an easy task to add third-party media processing algorithms to any WebRTC application, like integrating Computer Vision, Augmented Reality, video indexing, and speech analysis. For example, features such as speech recognition, sentiment analysis or face recognition can be developed by specialized teams, and then seamlessly added to Kurento as new modules.

1.1 WebRTC media servers

[WebRTC](#) is a set of protocols, mechanisms and APIs that provide browsers and mobile applications with Real-Time Communications (RTC) capabilities over peer-to-peer connections. It has been conceived as a technology that allows browsers to communicate directly without the mediation of any kind of infrastructure. However, this model is only enough for creating basic web applications; features such as group communications, media stream recording, media broadcasting, or media transcoding are difficult to implement on top of it. For this reason, many applications end up requiring an intermediate media server.

Conceptually, a WebRTC media server is just a multimedia middleware where media traffic passes through when moving from source to destinations.

Media servers are capable of processing incoming media streams and offer different outcomes, such as:

- Group Communications: Distributing among several receivers the media stream that one peer generates, i.e. acting as a Multi-Conference Unit (“MCU”).
- Mixing: Transforming several incoming stream into one single composite stream.

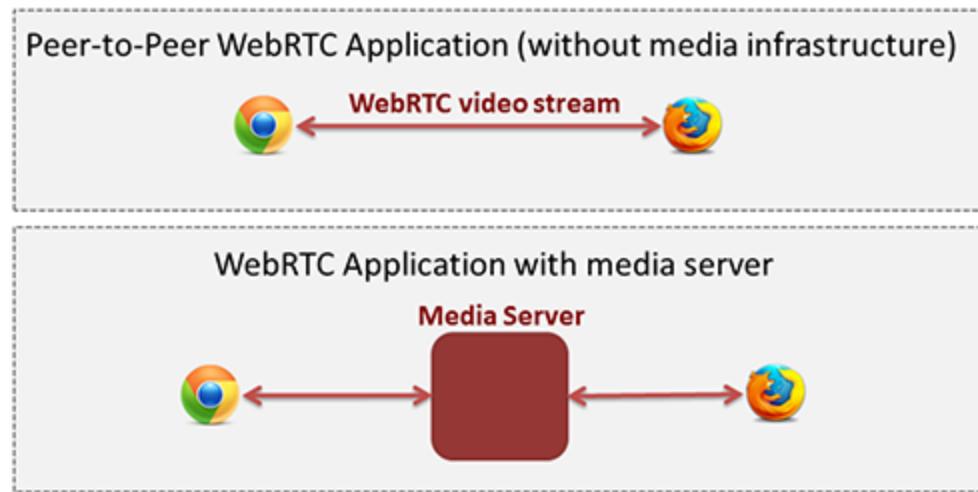


Fig. 1.1: *Peer-to-peer WebRTC approach vs. WebRTC through a media server*

- Transcoding: On-the-fly adaptation of codecs and formats between incompatible clients.
- Recording: Storing in a persistent way the media exchanged among peers.

1.2 Kurento Media Server

Kurento's main component is the **Kurento Media Server** (KMS), responsible for media transmission, processing, recording, and playback. KMS is built on top of the fantastic *GStreamer* multimedia library, and provides the following features:

- Networked streaming protocols, including *HTTP*, *RTP* and *WebRTC*.
- Group communications (MCU and SFU functionality) supporting both media mixing and media routing/dispatching.
- Generic support for filters implementing Computer Vision and Augmented Reality algorithms.
- Media storage that supports writing operations for *WebM* and *MP4* and playing in all formats supported by *GStreamer*.
- Automatic media transcoding between any of the codecs supported by GStreamer, including VP8, H.264, H.263, AMR, OPUS, Speex, G.711, and more.

1.3 Kurento Design Principles

Kurento is designed based on the following main principles:

Separate Media and Signaling Planes *Signaling* and *Media* are two separate planes and Kurento is designed so that applications can handle separately those facets of multimedia processing.

Distribution of Media and Application Services Kurento Media Server and applications can be collocated, escalated or distributed among different machines.

A single application can invoke the services of more than one Kurento Media Server. The opposite also applies, that is, a Kurento Media Server can attend the requests of more than one application.

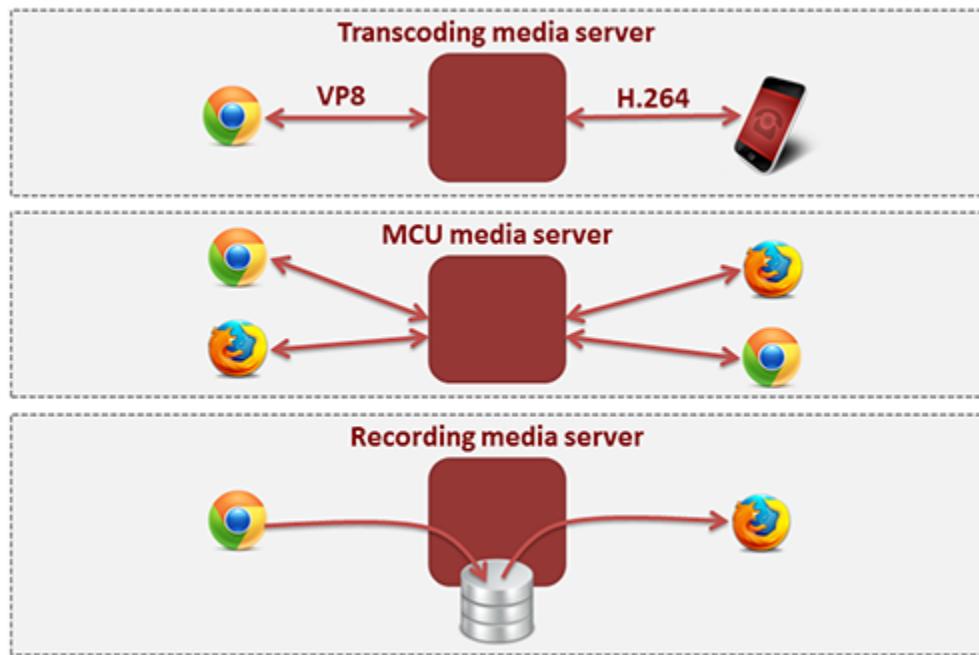


Fig. 1.2: Typical WebRTC Media Server capabilities

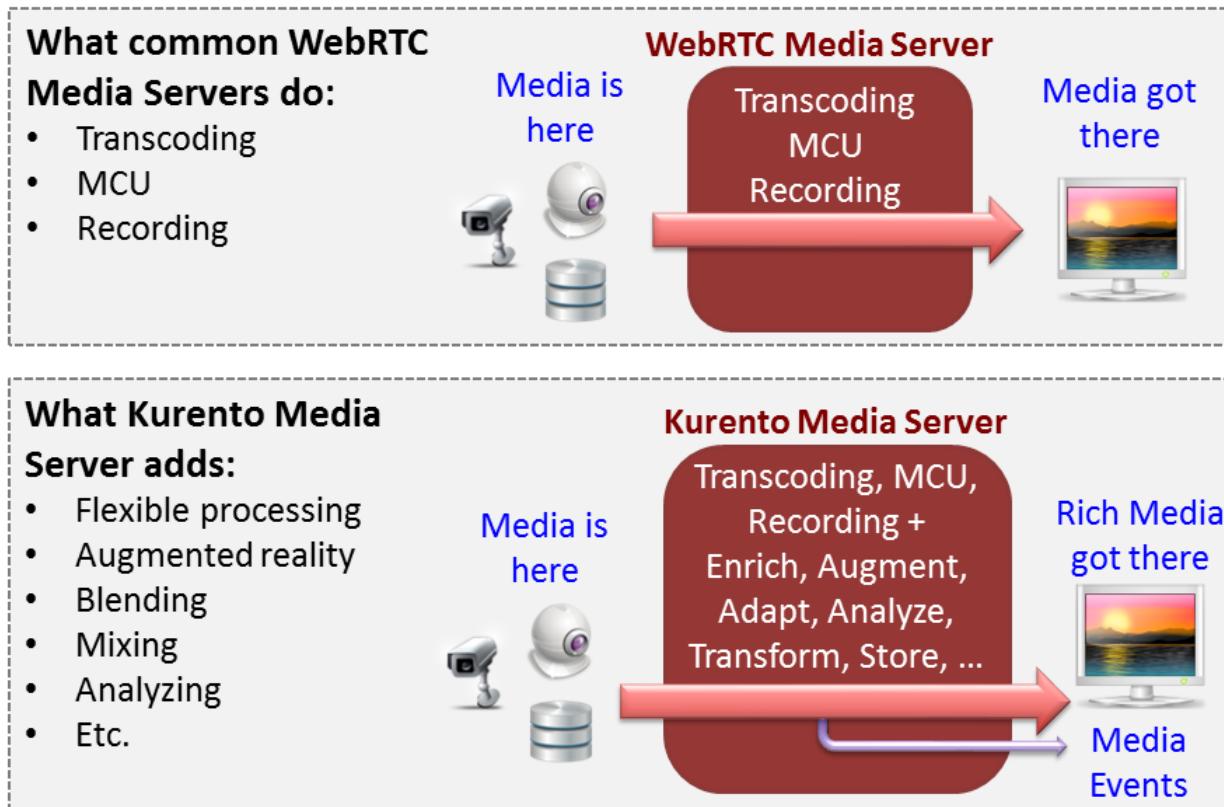


Fig. 1.3: Kurento Media Server capabilities

Suitable for the Cloud Kurento is suitable to be integrated into cloud environments to act as a PaaS (Platform as a Service) component.

Media Pipelines Chaining *Media Elements* via *Media Pipelines* is an intuitive approach to challenge the complexity of multimedia processing.

Application development Developers do not need to be aware of internal Kurento Media Server complexities: all the applications can be deployed in any technology or framework the developer likes, from client to server. From browsers to cloud services.

End-to-End Communication Capability Kurento provides end-to-end communication capabilities so developers do not need to deal with the complexity of transporting, encoding/decoding and rendering media on client devices.

Fully Processable Media Streams Kurento enables not only interactive interpersonal communications (e.g. Skype-like with conversational call push/reception capabilities), but also human-to-machine (e.g. Video on Demand through real-time streaming) and machine-to-machine (e.g. remote video recording, multisensory data exchange) communications.

Modular Processing of Media Modularization achieved through *media elements* and *pipelines* allows defining the media processing functionality of an application through a “graph-oriented” language, where the application developer is able to create the desired logic by chaining the appropriate functionalities.

Auditable Processing Kurento is able to generate rich and detailed information for QoS monitoring, billing and auditing.

Seamless IMS integration Kurento is designed to support seamless integration into the *IMS* infrastructure of Telephony Carriers.

Transparent Media Adaptation Layer Kurento provides a transparent media adaptation layer to make the convergence among different devices having different requirements in terms of screen size, power consumption, transmission rate, etc. possible.

CHAPTER 2

About OpenVidu

OpenVidu is a new project from the same team that created Kurento. It acts as a wrapper around an installation of Kurento Media Server and encapsulates most of its features, in order to greatly simplify some of the most typical use cases of WebRTC, such as conference rooms.

An application developer using OpenVidu doesn't need to worry about all the low-level technologies and protocols that form part of typical WebRTC communications. The main goal of this project is to provide a simple, effective, easy-to-use API; just include the OpenVidu client-side and OpenVidu Server for handling the media flows, and you'll have a full-featured WebRTC-capable application.

For more advanced needs, or for applications that require more versatile management of media processing pipelines, Kurento is still the go-to solution; however, if you are planning on building a service which matches one of the simplified use cases covered by OpenVidu, we strongly suggest to check it out as it will be easier and cheaper to go that route.

Check out the OpenVidu project page, which contains all the relevant information: <http://openvidu.io/>

CHAPTER 3

Getting Started

Generally speaking, these are the first steps that any user interested in Kurento should follow:

1. Know your use case

Choose between Kurento and [OpenVidu](#).

Kurento Media Server has been designed as a general-purpose platform that can be used to create any kind of multimedia streaming applications. This makes KMS a powerful tool, however it also means that there is some unavoidable complexity that the developer must face.

WebRTC is a complex standard with a lot of moving parts, and you need to know about each one of these components and how they work together to achieve the multimedia communications that the standard strives to offer.

If your intended application consists of a complex setup with different kinds of sources and varied use cases, then Kurento is the best leverage you can use.

However, if you intend to solve a simpler use case, such as those of video conference applications, the [OpenVidu](#) project builds on top of Kurento to offer a simpler and easier to use solution that will save you time and development effort.

2. Install KMS

The [installation guide](#) explains different ways in which Kurento can be installed in your system. The fastest and easiest one is to use our [pre-configured template for Amazon AWS](#).

3. Configure KMS

KMS is able to run as-is after a normal installation. However, there are several parameters that you might want to tune in the [configuration files](#).

4. Write an Application

Write an application that queries the [Kurento API](#) to make use of the capabilities offered by KMS. The easiest way of doing this is to build on one of the provided [Kurento Clients](#).

In general, you can use *any programming language* to write your application, as long as it speaks the [Kurento Protocol](#).

Have a look at the [features](#) offered by Kurento, and follow any of the multiple [tutorials](#) that explain how to build basic applications.

5. Ask for help

If you face any issue with Kurento itself or have difficulties configuring the plethora of mechanisms that form part of WebRTC, don't hesitate to [ask for help](#) to our community of users.

Still, there are times when the problems at hand require more specialized study. If you wish to get help from expert people with more inside knowledge of Kurento, we also offer the option of direct [support contracts](#).

6. Enjoy!

Kurento is a project that aims to bring the latest innovations closer to the people, and help connect them together. Make a great application with it, and let us know! We will be more than happy to find out about who is using Kurento and what is being built with it :-)

CHAPTER 4

Installation Guide

Kurento Media Server (KMS) can be made available through two different methods: either a local native installation, or an EC2 instance in the [Amazon Web Services \(AWS\)](#) cloud service.

Using AWS is suggested to users who don't want to worry about properly configuring a server and all software packages, because the provided setup does all this automatically.

On the other hand, the local installation will allow to have total control of the installation process. This method makes use of public package repositories that hold the latest released versions of KMS. Besides that, a common need is to also install a [STUN](#) or [TURN](#) server, especially if KMS or any of its clients are located behind a [NAT](#). This document includes some details about that topic.

If you want to try pre-release builds of KMS, then head to the section [Installing Pre-Release Builds](#).

4.1 Amazon Web Services

The Kurento project provides an *AWS CloudFormation* template file. It can be used to create an EC2 instance that comes with everything needed and totally pre-configured to run KMS, including a [Coturn](#) server. Follow these steps to use it:

1. Access the [AWS CloudFormation Console](#).
2. Click on *Create Stack*.
3. Look for the section *Choose a template*, and choose the option *Specify an Amazon S3 template URL*. Then, in the text field that gets enabled, paste this URL: <https://s3-eu-west-1.amazonaws.com/aws.kurento.org/KMS-Coturn-cfn.yaml>.
4. Follow through the steps of the configuration wizard.

Note: The template file includes a *Coturn* server. The default user/password for this server is `kurento/kurento`. You can optionally change the username, but **make sure to change the default password**.

5. Finish the Stack creation process. Wait until the status of the newly created Stack reads *CREATE_COMPLETE*.
6. Select the Stack and then open the *Outputs* tab, where you'll find the instance's public IP address, and the Kurento Media Server endpoint URL that must be used by *Client Applications*.

4.2 Local Installation

With this method, you will install KMS from the native Ubuntu package repositories made available by the Kurento project.

KMS has explicit support for two Long-Term Support (*LTS*) distributions of Ubuntu: **Ubuntu 14.04 (Trusty)** and **Ubuntu 16.04 (Xenial)**. Only the 64-bits editions are supported.

Currently, the main development environment for KMS is Ubuntu 16.04 (Xenial), so if you are in doubt, this is the preferred Ubuntu distribution to choose. However, all features and bug fixes are still being backported and tested on Ubuntu 14.04 (Trusty), so you can continue running this version if required.

1. Define what version of Ubuntu is installed in your system. Open a terminal and copy **only one** of these commands:

```
# KMS for Ubuntu 14.04 (Trusty)
DISTRO="trusty"
```

```
# KMS for Ubuntu 16.04 (Xenial)
DISTRO="xenial"
```

2. Add the Kurento repository to your system configuration. Run these two commands in the same terminal you used in the previous step:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 5AFA7A83
```

```
sudo tee "/etc/apt/sources.list.d/kurento.list" >/dev/null <<EOF
# Kurento Media Server - Release packages
deb [arch=amd64] http://ubuntu.openvidu.io/6.7.1 $DISTRO kms6
EOF
```

3. Install KMS:

```
sudo apt-get update
sudo apt-get install kurento-media-server
```

This will install the KMS release version that was specified in the previous commands.

The server includes service files which integrate with the Ubuntu init system, so you can use the following commands to start and stop it:

```
sudo service kurento-media-server start
sudo service kurento-media-server stop
```

4.3 STUN and TURN servers

If Kurento Media Server or its client application are located behind a *NAT* (e.g. in any cloud provider), you need to use a *STUN* or a *TURN* server in order to achieve *NAT traversal*. In most cases, STUN is effective in addressing the

NAT issue with most consumer network devices (routers). However, it doesn't work for many corporate networks, so a TURN server becomes necessary.

Apart from that, you need to open all UDP ports in your system configuration, as STUN will use any random port from the whole [0-65535] range.

Note: The features provided by TURN are a superset of those provided by STUN. This means that *you don't need to configure a STUN server if you are already using a TURN server*.

4.3.1 STUN server

To configure a STUN server in KMS, uncomment the following lines in the WebRtcEndpoint configuration file, located at `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`:

```
stunServerAddress=<serverIp>
stunServerPort=<serverPort>
```

Note: Be careful since comments inline (with ;) are not allowed for parameters in the configuration files. Thus, the following line **is not correct**:

```
stunServerAddress=<serverIp> ; Only IP addresses are supported
```

... and must be changed to something like this:

```
; Only IP addresses are supported
stunServerAddress=<serverIp>
```

The parameter `serverIp` should be the public IP address of the STUN server. It must be an IP address, **not a domain name**.

It should be easy to find some public STUN servers that are made available for free. For example:

```
173.194.66.127:19302
173.194.71.127:19302
74.125.200.127:19302
74.125.204.127:19302
173.194.72.127:19302
74.125.23.127:3478
77.72.174.163:3478
77.72.174.165:3478
77.72.174.167:3478
77.72.174.161:3478
208.97.25.20:3478
62.71.2.168:3478
212.227.67.194:3478
212.227.67.195:3478
107.23.150.92:3478
77.72.169.155:3478
77.72.169.156:3478
77.72.169.164:3478
77.72.169.166:3478
77.72.174.162:3478
77.72.174.164:3478
```

```
77.72.174.166:3478  
77.72.174.160:3478  
54.172.47.69:3478
```

4.3.2 TURN server

To configure a TURN server in KMS, uncomment the following lines in the WebRtcEndpoint configuration file, located at `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`:

```
turnURL=<user>:<password>@<serverIp>:<serverPort>
```

The parameter `serverIp` should be the public IP address of the TURN server. It must be an IP address, **not a domain name**.

See some examples of TURN configuration below:

```
turnURL=kurento:kurento@111.222.333.444:3478
```

... or using a free access [Numb](#) TURN/STUN server:

```
turnURL=user:password@66.228.45.110:3478
```

Note that it is somewhat easy to find free STUN servers available on the net, because their functionality is pretty limited and it is not costly to keep them working for free. However, this doesn't happen with TURN servers, which act as a media proxy between peers and thus the cost of maintaining one is much higher.

It is rare to find a TURN server which works for free while offering good performance. Usually, each user opts to maintain their own private TURN server instances.

[Coturn](#) is an open source implementation of a TURN/STUN server. In the [FAQ](#) section there is a description about how to install and configure it.

4.4 Check your installation

To verify that KMS is up and running, use this command and look for the `kurento-media-server` process:

```
ps -ef | grep kurento-media-server  
> nobody 1270 1 0 08:52 ? 00:01:00 /usr/bin/kurento-media-server
```

Unless configured otherwise, KMS will open the port 8888 to receive requests and send responses by means of the [Kurento Protocol](#). Use this command to verify that this port is listening for incoming packets:

```
sudo netstat -tupan | grep kurento  
> tcp6 0 0 :::8888 ::::* LISTEN 1270/kurento-media-server
```

CHAPTER 5

Installing Pre-Release Builds

Some components of KMS are built nightly, with the code developed during that same day. Other components are built immediately when code is merged into the source repositories.

These builds end up being uploaded to *Development* repositories so they can be installed by anyone. Use these if you want to develop *Kurento itself*, or if you want to try the latest changes before they are officially released.

Warning: Pre-release builds always represent the current state on the software development; 99% of the time this is stable code, very close to what will end up being released.

However, in the other 1% of cases it might include undocumented changes, regressions, bugs or deprecations. It's safer to be conservative and **never** use pre-release builds in a production environment.

Note: If you are looking to build KMS from the source code, then you should check the section aimed at development of *KMS itself*: [Developing KMS](#).

5.1 Kurento Media Server

The steps to install a pre-release version of KMS are pretty much the same as those explained in [Local Installation](#), with the only change of using a different package repository.

1. Define what version of Ubuntu is installed in your system. Open a terminal and copy **only one** of these commands:

```
# KMS for Ubuntu 14.04 (Trusty)
DISTRO="trusty"
```

```
# KMS for Ubuntu 16.04 (Xenial)
DISTRO="xenial"
```

2. Add the Kurento repository to your system configuration. Run these two commands in the same terminal you used in the previous step:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 5AFA7A83
```

```
sudo tee "/etc/apt/sources.list.d/kurento.list" >/dev/null <<EOF
# Kurento Media Server - Nightly packages
deb [arch=amd64] http://ubuntu.openvidu.io/dev $DISTRO kms6
EOF
```

3. Install KMS:

```
sudo apt-get update
sudo apt-get install kurento-media-server
```

This will install the latest KMS pre-release (potentially unstable) version.

5.2 Kurento Java Client

The development builds of the Kurento Java Client are made available for Maven in <http://maven.kurento.org/>. To use these, you need to add first this repository to your Maven configuration.

Adding a repository to Maven can be done at three scope levels:

- **Project level.**

This will add access to development builds only for the project where the configuration is done. Open the project's *pom.xml* and include this:

```
<project>
  ...
  <repositories>
    <repository>
      <id>kurento-snapshots</id>
      <name>Kurento Snapshots</name>
      <url>https://maven.openvidu.io/archiva/repository/snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>kurento-snapshots</id>
      <name>Kurento Snapshots</name>
      <url>https://maven.openvidu.io/archiva/repository/snapshots/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</project>
```

```
...
</project>
```

After this is included, there are two ways to use the updated versions:

1. In the same *pom.xml*, look for the desired *<dependency>* and change its version. For example:

```
<dependency>
  <groupId>org.kurento</groupId>
  <artifactId>kurento-client</artifactId>
  <version>6.6.3-SNAPSHOT</version>
</dependency>
```

2. If you have not specified a dependency version, use the *-U* switch in your next Maven run to force updating all dependencies.

- **User and System levels.**

The file *settings.xml* provides configuration for all projects, but its contents have a different reach depending on where it is located:

- At *\$HOME/.m2/settings.xml*, it defines the settings that will be applied for a single user.
- At */etc/maven/settings.xml*, it defines the settings that will be applied for all Maven users on a machine.

To use this method, first edit the settings file at one of the mentioned locations, and include this:

```
<settings>
  ...
  <profiles>
    <profile>
      <id>kurento</id>
      <repositories>
        <repository>
          <id>kurento-snapshots</id>
          <name>Kurento Snapshots</name>
          <url>https://maven.openvidu.io/archiva/repository/snapshots/</url>
          <releases>
            <enabled>false</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>kurento-snapshots</id>
          <name>Kurento Snapshots</name>
          <url>https://maven.openvidu.io/archiva/repository/snapshots/</url>
          <releases>
            <enabled>false</enabled>
          </releases>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
</settings>
```

```
</profiles>
...
</settings>
```

After this is included, use the `-Pkurento` switch in your next Maven run to enable the new profile, so all artifacts get downloaded into your local repository. Once in your local repository, Maven can successfully resolve the dependencies and the profile no longer needs to be activated in future runs.

For more information about adding snapshot repositories to Maven, check their official documentation: [Guide to Testing Development Versions of Plugins](#).

5.3 Kurento JavaScript Client

5.3.1 Node.js

If you are using the Kurento JavaScript Client from a Node.js application and want to use the latest development version of this library, you have to change the *dependencies* section in the application's *package.json*. This way, NPM will point directly to the development repository:

```
"dependencies": {
  "kurento-client": "Kurento/kurento-client-js#master",
}
```

5.3.2 Browser JavaScript

If you are using the Kurento JavaScript Client from a browser application, with Bower to handle JS dependencies, and want to use the latest development version of this library, you have to change the *dependencies* section in the application's *bower.json*. This way, Bower will point directly to the development repository:

```
"dependencies": {
  "kurento-client": "master",
  "kurento-utils": "master",
}
```

Alternatively, if your browser application is pointing directly to JavaScript libraries from HTML resources, then you have to change to development URLs:

```
<script type="text/javascript"
  src="http://builds.openvidu.io/dev/master/latest/js/kurento-client.min.js">
</script>
```

CHAPTER 6

Kurento Tutorials

This section contains tutorials showing how to use Kurento framework to build different types of *WebRTC* and multi-media applications. These tutorials come in three flavors:

- **Java:** Showing applications where clients interact with *Spring Boot*-based applications, that host the logic orchestrating the communication among clients and control Kurento Media Server capabilities.
- **Browser JavaScript:** These show applications executing at the browser and communicating directly with the Kurento Media Server. In these tutorials all logic is directly hosted by the browser. Hence, no application server is necessary.
- **Node.js:** In which clients interact with an application server made with Node.js technology. The application server holds the logic orchestrating the communication among the clients and controlling Kurento Media Server capabilities for them.

Note: The tutorials have been created with learning objectives. They are not intended to be used in production environments where different unmanaged error conditions may emerge. **Use at your own risk!**

Note: These tutorials require HTTPS in order to use WebRTC. Following instructions will provide further information about how to enable application security.

6.1 Hello World

This is one of the simplest WebRTC applications you can create with Kurento. It implements a *WebRTC loopback* (a WebRTC media stream going from client to Kurento Media Server and back to the client)

6.1.1 Kurento Java Tutorial - Hello World

This web application has been designed to introduce the principles of programming with Kurento for Java developers. It consists on a [WebRTC](#) video communication in mirror (*loopback*). This tutorial assumes you have basic knowledge of Java, JavaScript, HTML and WebRTC. We also recommend reading the section [About Kurento and WebRTC](#) before starting this tutorial.

Note: This application uses HTTPS. It will work fine if you run it in `localhost` and accept a security exception in the browser, but you should secure your application if running remotely. For more info, check [Configure Java applications to use HTTPS](#).

Quick start

Follow these steps to run this demo application:

1. Install Kurento Media Server: [Installation Guide](#).
2. Run these commands:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-hello-world
git checkout 6.7.1
mvn clean compile exec:java -Dkms.url=ws://localhost:8888/kurento
```

3. Open the demo page with a WebRTC-compliant browser (Chrome, Firefox): <https://localhost:8443/>
4. Click on *Start* to begin the demo.
5. Grant access to your webcam.
6. As soon as the loopback connection is negotiated and established, you should see your webcam video in both the local and remote placeholders.
7. Click on *Stop* to finish the demo.

Understanding this example

Kurento provides developers a **Kurento Java Client** to control the **Kurento Media Server**. This client library can be used in any kind of Java application: Server Side Web, Desktop, Android, etc. It is compatible with any framework like Java EE, Spring, Play, Vert.x, Swing and JavaFX.

This *Hello World* demo is one of the simplest web applications you can create with Kurento. The following picture shows a screenshot of this demo running:

The interface of the application (an HTML web page) is composed by two HTML5 `<video>` tags: one showing the local stream (as captured by the device webcam) and the other showing the remote stream sent by the media server back to the client.

The logic of the application is quite simple: the local stream is sent to the Kurento Media Server, which sends it back to the client without modifications. To implement this behavior, we need to create a [Media Pipeline](#) composed by a single [Media Element](#), i.e. a **WebRtcEndpoint**, which holds the capability of exchanging full-duplex (bidirectional) WebRTC media flows. This media element is connected to itself so that the media it receives (from browser) is sent back (to browser). This media pipeline is illustrated in the following picture:

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento**

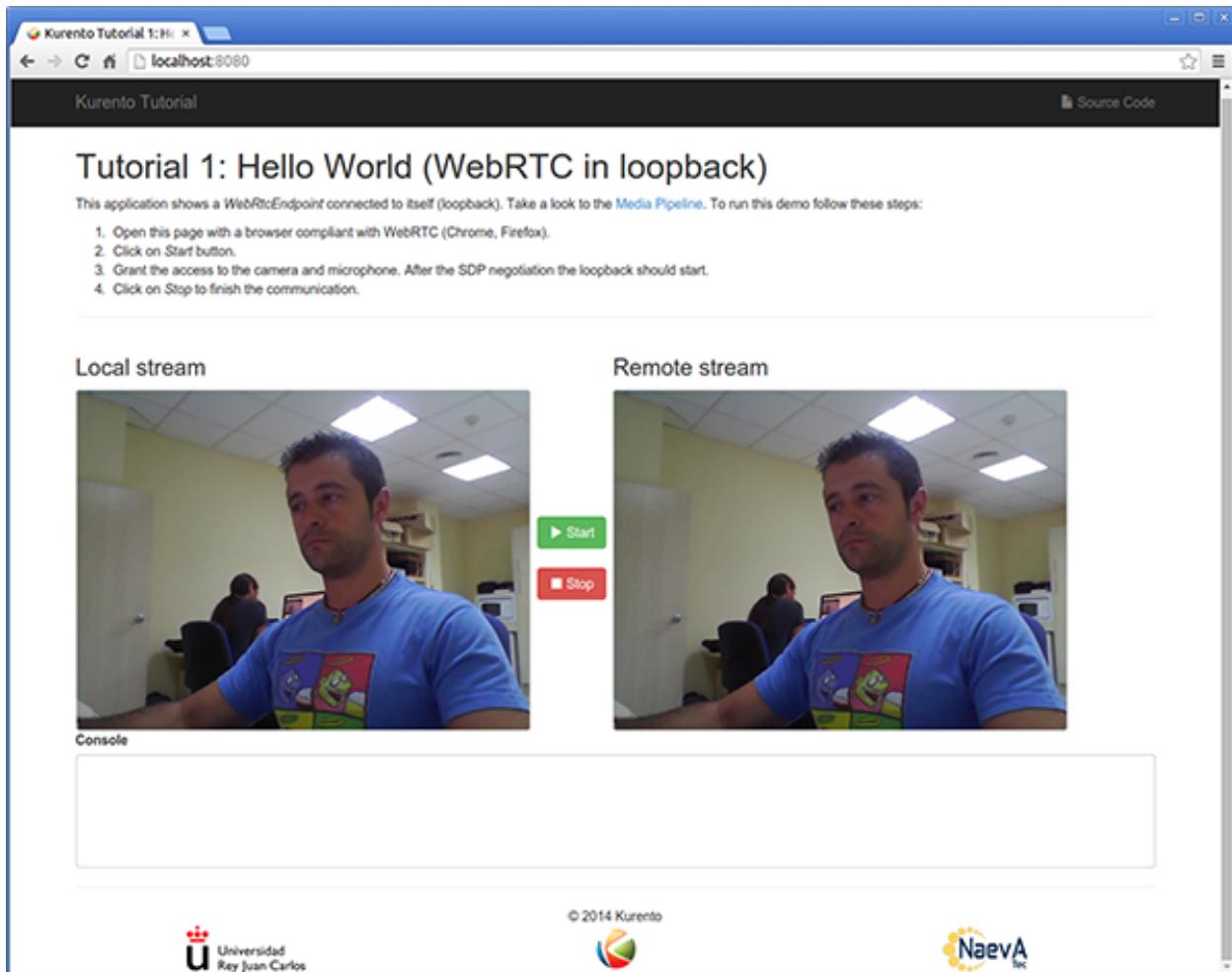


Fig. 6.1: *Kurento Hello World Screenshot: WebRTC in loopback*

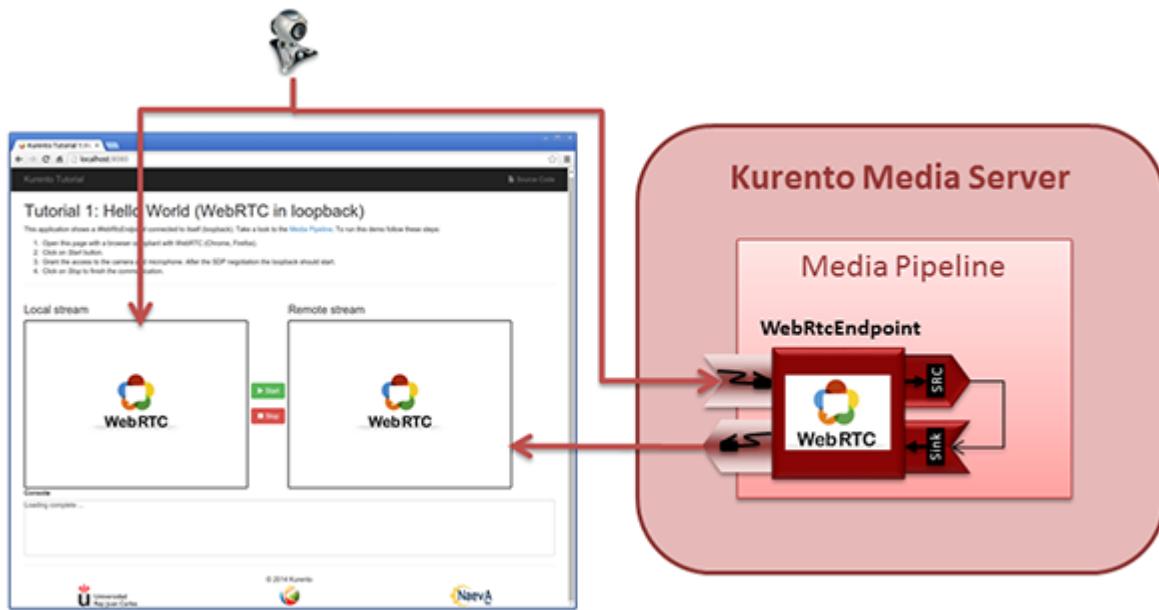


Fig. 6.2: *Kurento Hello World Media Pipeline in context*

Java Client API, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used:

1. A WebSocket is created between client and application server to implement a custom signaling protocol.
2. Another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server.

This communication takes place using the **Kurento Protocol**. For a detailed description, please read this section: [Kurento Protocol](#).

The diagram below shows a complete sequence diagram, of the interactions with the application interface to: i) JavaScript logic; ii) Application server logic (which uses the Kurento Java Client); iii) Kurento Media Server.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

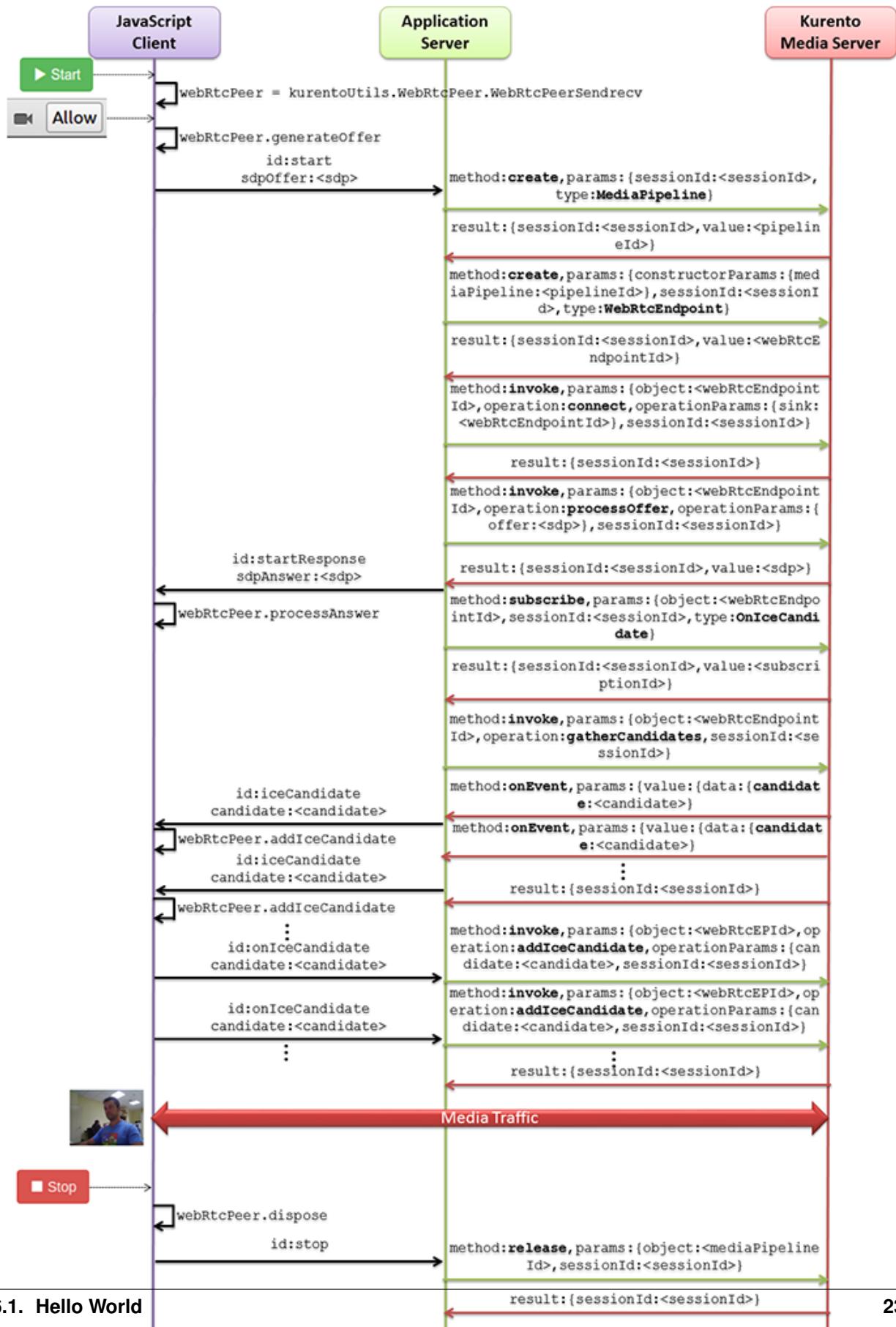
This demo has been developed using **Java** in the server-side, based on the [Spring Boot](#) framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

In the following, figure you can see a class diagram of the server side code:

The main class of this demo is [HelloWorldApp](#).

As you can see, the *KurentoClient* is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at



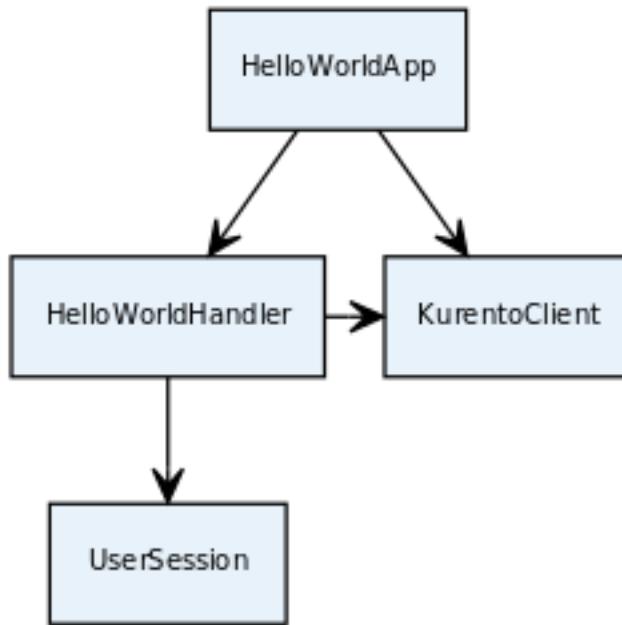


Fig. 6.4: Server-side class diagram of the `HelloWorld` app

`localhost`, listening in port 8888. If you reproduce this example, you'll need to insert the specific location of your `Kurento Media Server` instance there.

Once the `Kurento Client` has been instantiated, you are ready for communicating with `Kurento Media Server` and controlling its multimedia capabilities.

```

@SpringBootApplication
@EnableWebSocket
public class HelloWorldApp implements WebSocketConfigurer {
    @Bean
    public HelloWorldHandler handler() {
        return new HelloWorldHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(handler(), "/helloworld");
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(HelloWorldApp.class, args);
    }
}
  
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate with the application server, by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` that processes `WebSocket` requests in the

path /helloworld.

The class `HelloWorldHandler` implements `TextWebSocketHandler` to handle text WebSocket requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the WebSocket. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

```
public void handleTextMessage(WebSocketSession session, TextMessage message)
    throws Exception {
    [...]
    switch (messageId) {
        case "start":
            start(session, jsonMessage);
            break;
        case "stop": {
            stop(session);
            break;
        }
        case "onIceCandidate":
            onRemoteIceCandidate(session, jsonMessage);
            break;
        default:
            sendError(session, "Invalid message, ID: " + messageId);
            break;
    }
    [...]
}
```

The `start()` method performs the following actions:

- **Configure media processing logic.** This is the part in which the application configures how Kurento has to process the media. In other words, the media pipeline is created here. To that aim, the object `KurentoClient` is used to create a `MediaPipeline` object. Using it, the media elements we need are created and connected. In this case, we only instantiate one `WebRtcEndpoint` for receiving the WebRTC stream and sending it back to the client.

```
final MediaPipeline pipeline = kurento.createMediaPipeline();

final WebRtcEndpoint webRtcEp =
    new WebRtcEndpoint.Builder(pipeline).build();

webRtcEp.connect(webRtcEp);
```

- **Create event listeners.** All objects managed by Kurento have the ability to emit several types of events, as explained in [Endpoint Events](#). The client applications can listen for them in order to have more insight about what is going on inside the processing logic of the media server. It is a good practice to listen for all possible events, so the client application has as much information as possible.

```
// Common events for all objects that inherit from BaseRtpEndpoint
addErrorListener(
    new EventListener<ErrorEvent>() { ... });
addMediaFlowInStateChangeListener(
    new EventListener<MediaFlowInStateChangeEvent>() { ... });
addMediaFlowOutStateChangeListener(
    new EventListener<MediaFlowOutStateChangeEvent>() { ... });
addConnectionStateChangedListener(
    new EventListener<ConnectionStateChangedEvent>() { ... });
addMediaStateChangedListener(
```

```
new EventListener<MediaStateChangedEvent>() { ... });
addMediaTranscodingStateChangeListener(
    new EventListener<MediaTranscodingStateChangeEvent>() { ... });

// Events specific to objects of class WebRtcEndpoint
addIceCandidateFoundListener(
    new EventListener<IceCandidateFoundEvent>() { ... });
addIceComponentStateChangeListener(
    new EventListener<IceComponentStateChangeEvent>() { ... });
addIceGatheringDoneListener(
    new EventListener<IceGatheringDoneEvent>() { ... });
addNewCandidatePairSelectedListener(
    new EventListener<NewCandidatePairSelectedEvent>() { ... });
```

- **WebRTC SDP negotiation.** In WebRTC, the *SDP Offer/Answer* model is used to negotiate the audio or video tracks that will be exchanged between peers, together with a subset of common features that they support. This negotiation is done by generating an SDP Offer in one of the peers, sending it to the other peer, and bringing back the SDP Answer that will be generated in response.

In this particular case, the SDP Offer has been generated by the browser and is sent to Kurento, which then generates an SDP Answer that must be sent back to the browser as a response.

```
// 'webrtcSdpOffer' is the SDP Offer generated by the browser;
// send the SDP Offer to KMS, and get back its SDP Answer
String webrtcSdpAnswer = webRtcEp.processOffer(webrtcSdpOffer);
sendMessage(session, webrtcSdpAnswer);
```

- **Gather ICE candidates.** While the SDP Offer/Answer negotiation is taking place, each one of the peers can start gathering the connectivity candidates that will be used for the *ICE* protocol. This process works very similarly to how a browser notifies its client code of each newly discovered candidate by emitting the event `RTCPeerConnection.onicecandidate`; likewise, Kurento's *WebRtcEndpoint* will notify its client application for each gathered candidate via the event `IceCandidateFound`.

```
webRtcEp.gatherCandidates();
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences.

These libraries are brought to the project as Maven dependencies which download all required files from WebJars.org; they are loaded in the `index.html` page, and are used in the `index.js` file.

In the following snippet we can see the creation of the WebSocket in the path `/helloworld`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in function `start`, the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```
var ws = new WebSocket('ws://' + location.host + '/helloworld');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);
```

```

switch (parsedMessage.id) {
case 'startResponse':
    startResponse(parsedMessage);
    break;
case 'error':
    if (state == I_AM_STARTING) {
        setState(I_CAN_START);
    }
    onError('Error message from server: ' + parsedMessage.message);
    break;
case 'iceCandidate':
    webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
        if (error)
            return console.error('Error adding candidate: ' + error);
    });
    break;
default:
    if (state == I_AM_STARTING) {
        setState(I_CAN_START);
    }
    onError('Unrecognized message', parsedMessage);
}
}

function start() {
    console.log('Starting video call ...');

    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(videoInput, videoOutput);

    console.log('Creating WebRtcPeer and generating local sdp offer ...');

    var options = {
        localVideo : videoInput,
        remoteVideo : videoOutput,
        onicecandidate : onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function(error) {
            if (error)
                return console.error(error);
            webRtcPeer.generateOffer(onOffer);
        });
}

function onOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}

```

```
function onIceCandidate(candidate) {
    console.log('Local candidate' + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
    };
    sendMessage(message);
}

function startResponse(message) {
    setState(I_CAN_STOP);
    console.log('SDP answer received from server. Processing ...');

    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function stop() {
    console.log('Stopping video call ...');
    setState(I_CAN_START);
    if (webRtcPeer) {
        webRtcPeer.dispose();
        webRtcPeer = null;

        var message = {
            id : 'stop'
        }
        sendMessage(message);
    }
    hideSpinner(videoInput, videoOutput);
}

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('Senging message: ' + jsonMessage);
    ws.send(jsonMessage);
}
```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with [WebJars](#).

6.1.2 JavaScript - Hello world

This web application has been designed to introduce the principles of programming with Kurento for JavaScript developers. It consists on a [WebRTC](#) video communication in mirror (*loopback*). This tutorial assumes you have basic knowledge of JavaScript, HTML and WebRTC. We also recommend reading the [Introducing Kurento](#) section before starting this tutorial.

Note: This tutorial has been configured for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

You'll need to install Kurento Media Server before running this example. Read [installation guide](#) for further information.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by a HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-hello-world
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the parameter `ws_uri` to the URL, as follows:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

Kurento provides developers a **Kurento JavaScript Client** to control **Kurento Media Server**. This client library can be used in any kind of JavaScript application including desktop and mobile browsers.

This *hello world* demo is one of the simplest web applications you can create with Kurento. The following picture shows an screenshot of this demo running:

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one showing the local stream (as captured by the device webcam) and the other showing the remote stream sent by the media server back to the client.

The logic of the application is quite simple: the local stream is sent to the Kurento Media Server, which sends it back to the client without modifications. To implement this behavior, we need to create a [Media Pipeline](#) composed by a

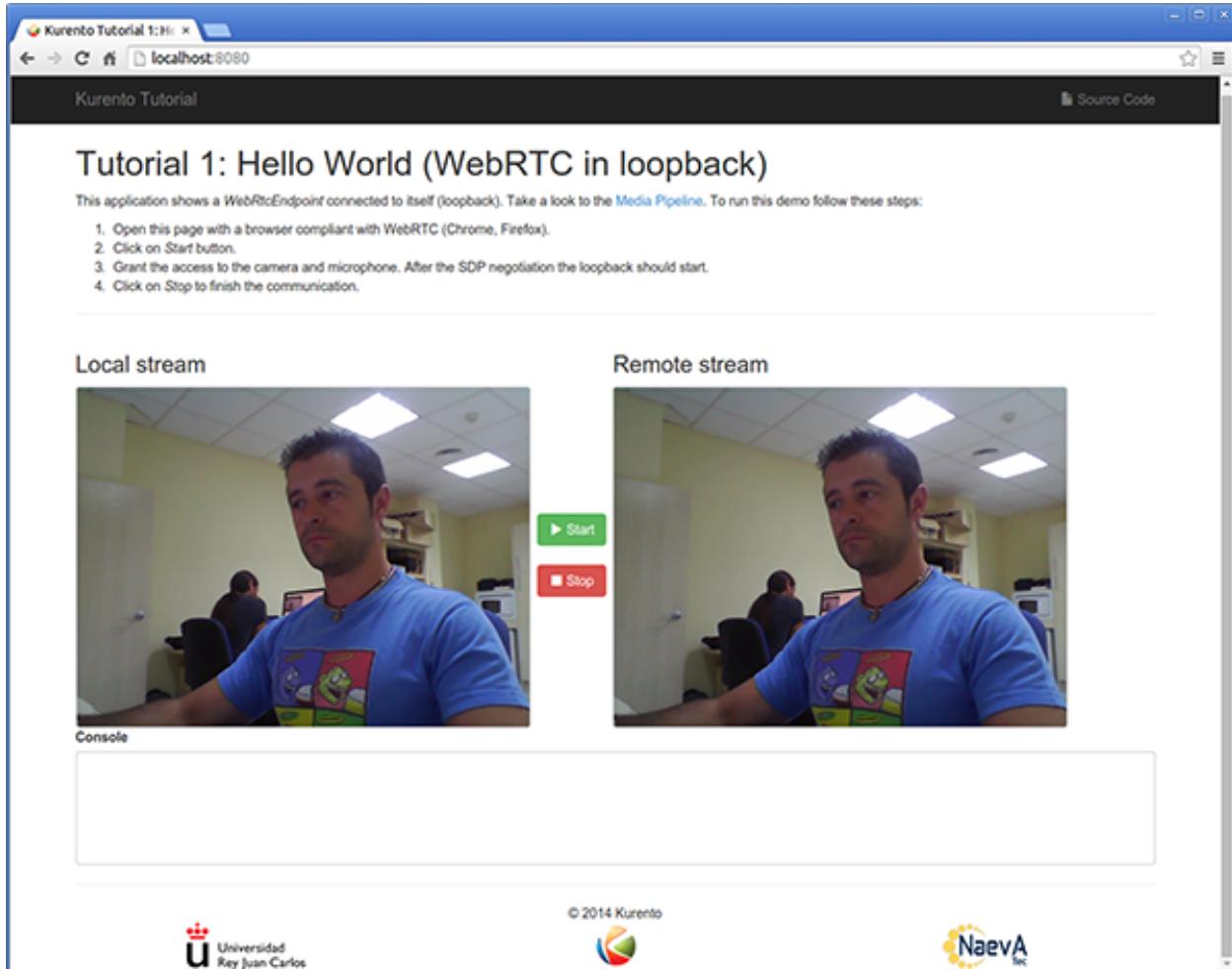


Fig. 6.5: *Kurento Hello World Screenshot: WebRTC in loopback*

single *Media Element*, i.e. a **WebRtcEndpoint**, which holds the capability of exchanging full-duplex (bidirectional) WebRTC media flows. This media element is connected to itself, so that the media it receives (from browser) is send back (to browser). This media pipeline is illustrated in the following picture:

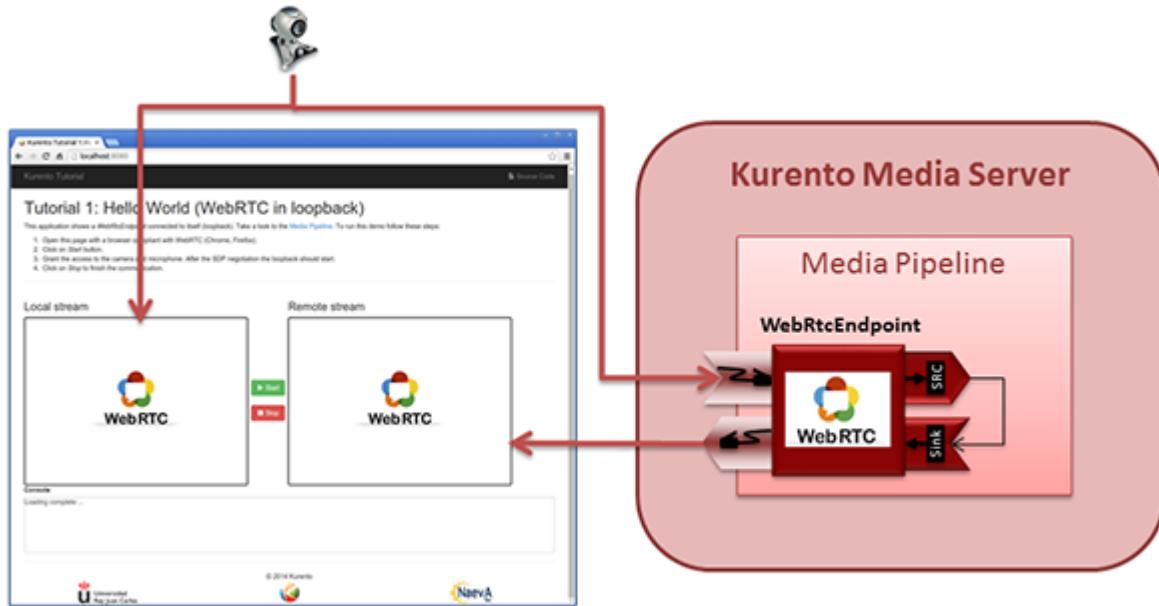


Fig. 6.6: Kurento Hello World Media Pipeline in context

This is a web application, and therefore it follows a client-server architecture. Nevertheless, due to the fact that we are using the Kurento JavaScript client, there is not need to use an application server since all the application logic is held by the browser. The Kurento JavaScript Client is used directly to control Kurento Media Server by means of a WebSocket bidirectional connection:

The following sections analyze in deep the client-side (JavaScript) code of this application, the dependencies, and how to run the demo. The complete source code can be found in [GitHub](#).

JavaScript Logic

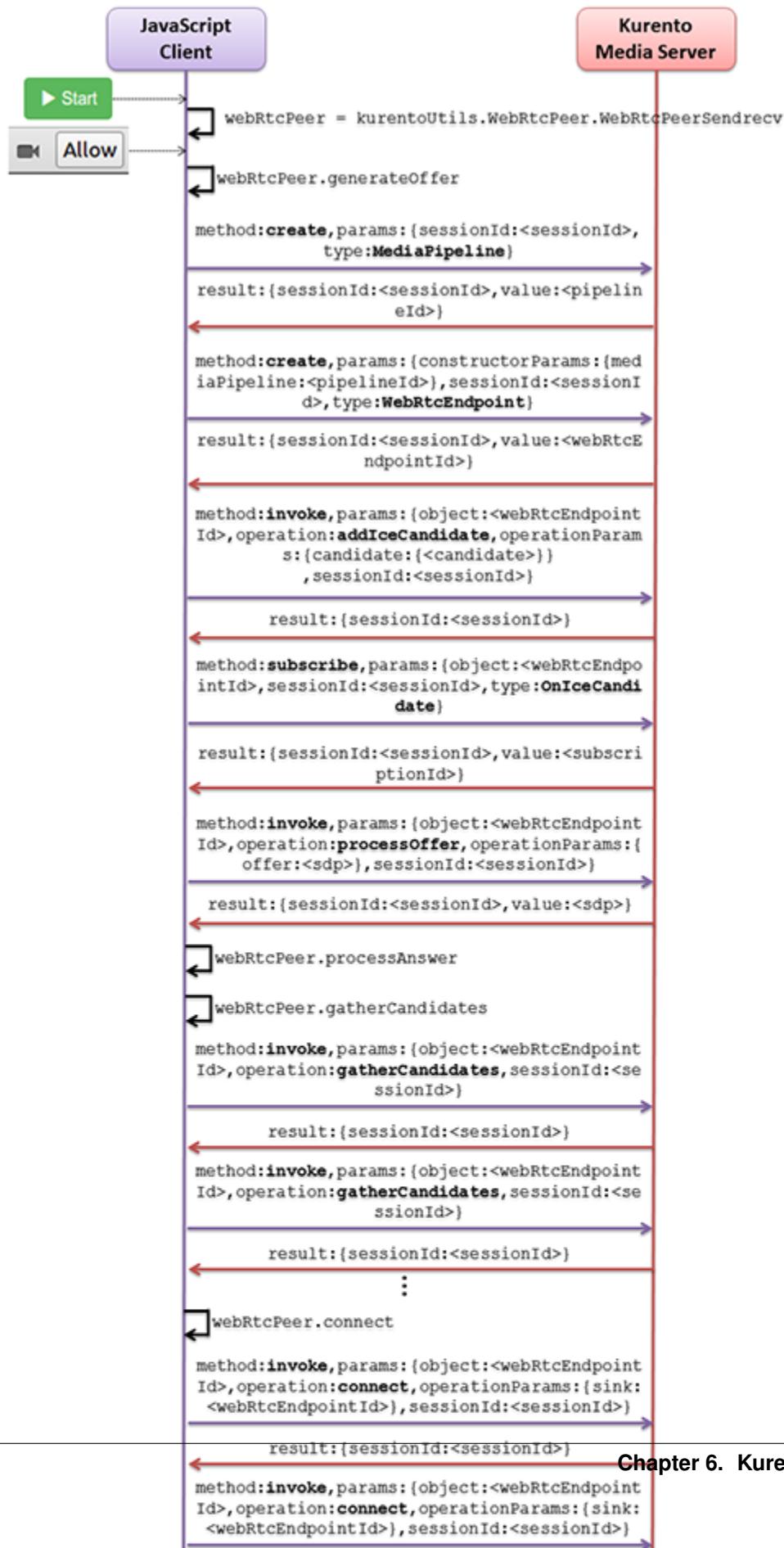
The Kurento *hello-world* demo follows a *Single Page Application* architecture ([SPA](#)). The interface is the following HTML page: `index.html`. This web page links two Kurento JavaScript libraries:

- **kurento-client.js** : Implementation of the Kurento JavaScript Client.
- **kurento-utils.js** : Kurento utility library aimed to simplify the WebRTC management in the browser.

In addition, these two JavaScript libraries are also required:

- **Bootstrap** : Web framework for developing responsive web sites.
- **jquery.js** : Cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- **adapter.js** : WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **ekko-lightbox** : Module for Bootstrap to open modal images, videos, and galleries.
- **demo-console** : Custom JavaScript console.

The specific logic of the *Hello World* JavaScript demo is coded in the following JavaScript file: `index.js`. In this file, there is a function which is called when the green button labeled as *Start* in the GUI is clicked.



```

var startButton = document.getElementById("start");

startButton.addEventListener("click", function() {
  var options = {
    localVideo: videoInput,
    remoteVideo: videoOutput
  };

  webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
    if(error) return onError(error)
    this.generateOffer(onOffer)
  });
}

[...]
}

```

The function `WebRtcPeer.WebRtcPeerSendrecv` abstracts the WebRTC internal details (i.e. `PeerConnection` and `getUserStream`) and makes possible to start a full-duplex WebRTC communication, using the HTML video tag with id `videoInput` to show the video camera (local stream) and the video tag `videoOutput` to show the remote stream provided by the Kurento Media Server.

Inside this function, a call to `generateOffer` is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the `KurentoClient` class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port 8888 at the same host than the HTTP serving the application.

```

[...]

var args = getopts(location.search,
{
  default:
  {
    ws_uri: 'ws://' + location.hostname + ':8888/kurento',
    ice_servers: undefined
  }
});

[...]

kurentoClient(args.ws_uri, function(error, client) {
  [...]
});

```

Once we have an instance of `kurentoClient`, we need to create a *Media Pipeline*, as follows:

```

client.create("MediaPipeline", function(error, _pipeline) {
  [...]
});

```

If everything works correctly, we will have an instance of a media pipeline (variable `_pipeline` in this example). With it, we are able to create *Media Elements*. In this example we just need a single `WebRtcEndpoint`.

In WebRTC, `SDP` is used for negotiating media exchanges between applications. Such negotiation happens based on the SDP offer and answer exchange mechanism by gathering the `ICE` candidates as follows:

```

pipeline = _pipeline;

pipeline.create("WebRtcEndpoint", function(error, webRtc) {

```

```
if(error) return onError(error);

setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {
    if(error) return onError(error);

    webRtcPeer.processAnswer(sdpAnswer, onError);
});

webRtc.gatherCandidates(onError);

[...]
});
```

Finally, the *WebRtcEndpoint* is connected to itself (i.e., in loopback):

```
webRtc.connect(webRtc, function(error) {
    if(error) return onError(error);

    console.log("Loopback established");
});
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},{  
    "urls":"stun:stun2.example.net"}]  
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",  
    "username":"user","credential":"myPassword"}]
```

Dependencies

All dependencies of this demo can to be obtained using *Bower*. The list of these dependencies are defined in the `bower.json` file, as follows:

```
"dependencies": {  
    "kurento-client": "6.7.1",  
    "kurento-utils": "6.7.1"  
}
```

To get these dependencies, just run the following shell command:

```
bower install
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [Bower](#).

6.1.3 Node.js - Hello world

This web application has been designed to introduce the principles of programming with Kurento for Node.js developers. It consists on a *WebRTC* video communication in mirror (*loopback*). This tutorial assumes you have basic

knowledge of JavaScript, Node.js, HTML and WebRTC. We also recommend reading the *Introducing Kurento* section before starting this tutorial.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-hello-world
git checkout 6.7.1
npm install
npm start
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules
npm cache clean
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

Kurento provides developers a **Kurento JavaScript Client** to control **Kurento Media Server**. This client library can be used from compatible JavaScript engines including browsers and Node.js.

This *hello world* demo is one of the simplest web application you can create with Kurento. The following picture shows an screenshot of this demo running:

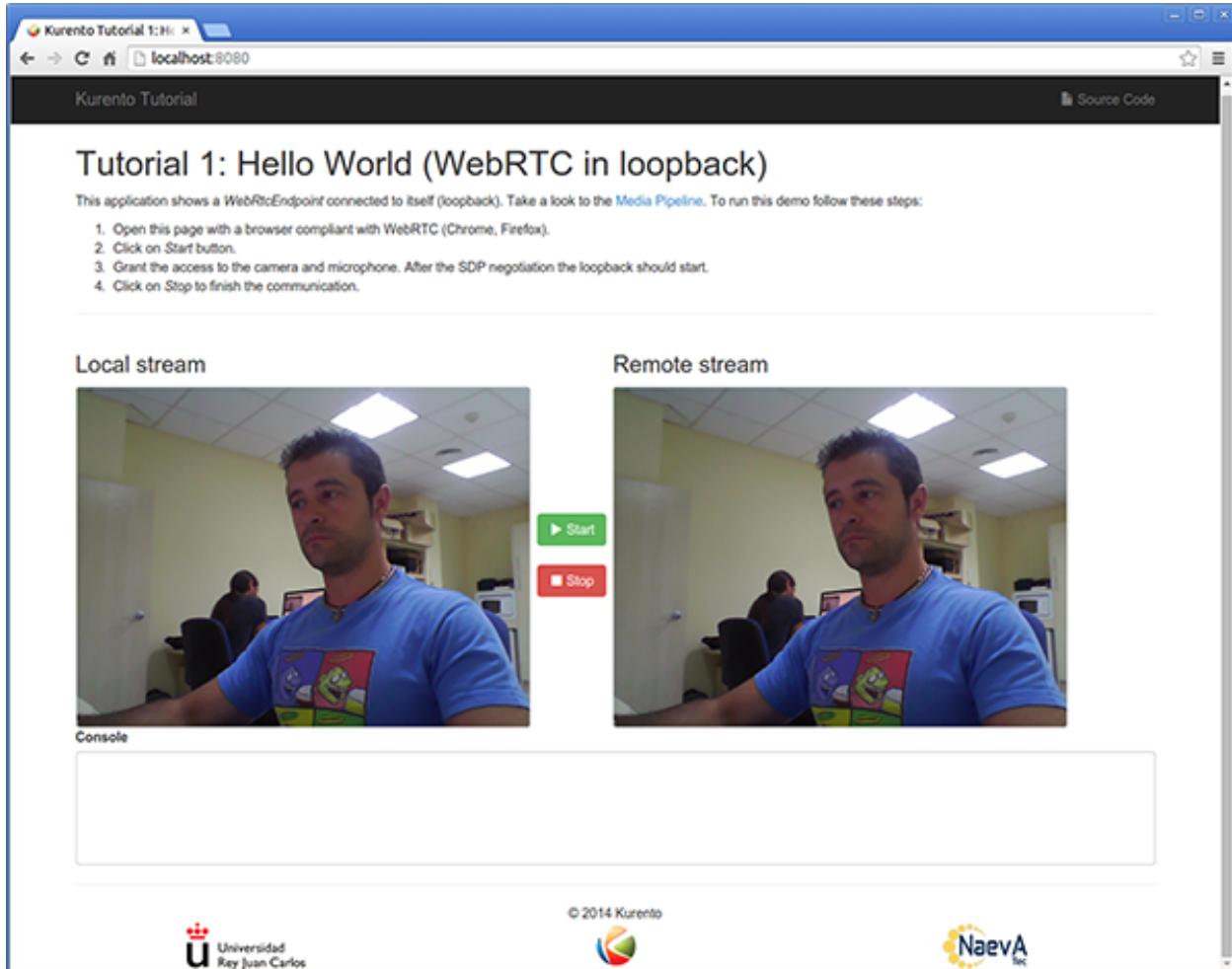


Fig. 6.8: *Kurento Hello World Screenshot: WebRTC in loopback*

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one showing the local stream (as captured by the device webcam) and the other showing the remote stream sent by the media server back to the client.

The logic of the application is quite simple: the local stream is sent to the Kurento Media Server, which returns it back to the client without modifications. To implement this behavior we need to create a *Media Pipeline* composed by a single *Media Element*, i.e. a **WebRtcEndpoint**, which holds the capability of exchanging full-duplex (bidirectional) WebRTC media flows. This media element is connected to itself so that the media it receives (from browser) is send back (to browser). This media pipeline is illustrated in the following picture:

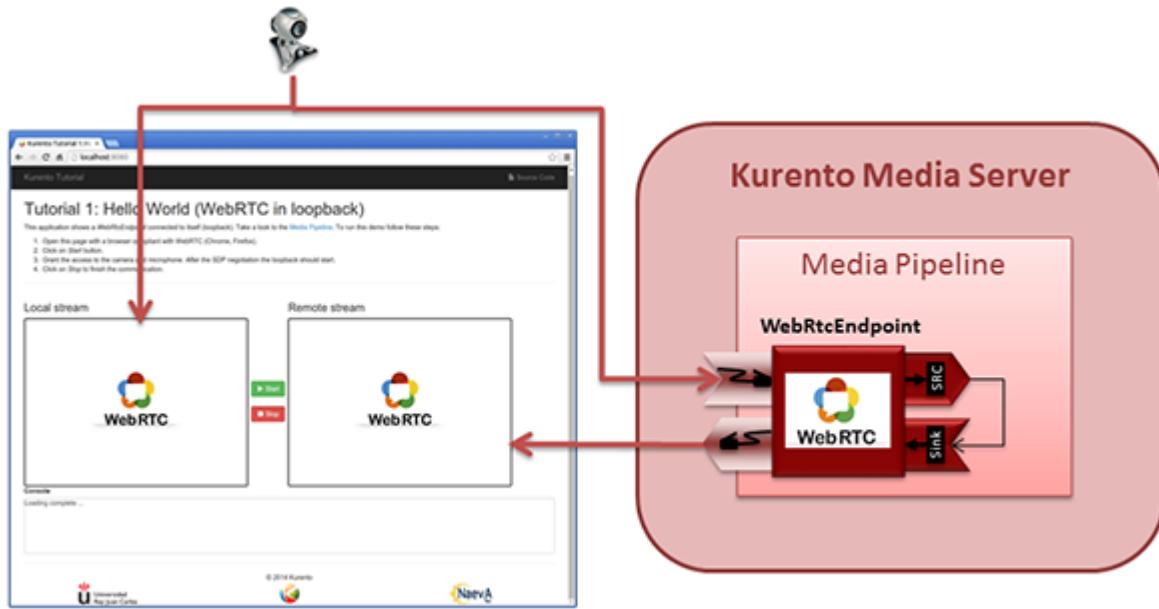


Fig. 6.9: *Kurento Hello World Media Pipeline in context*

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side we use a Node.js application server consuming the **Kurento JavaScript Client** API to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Javascript Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

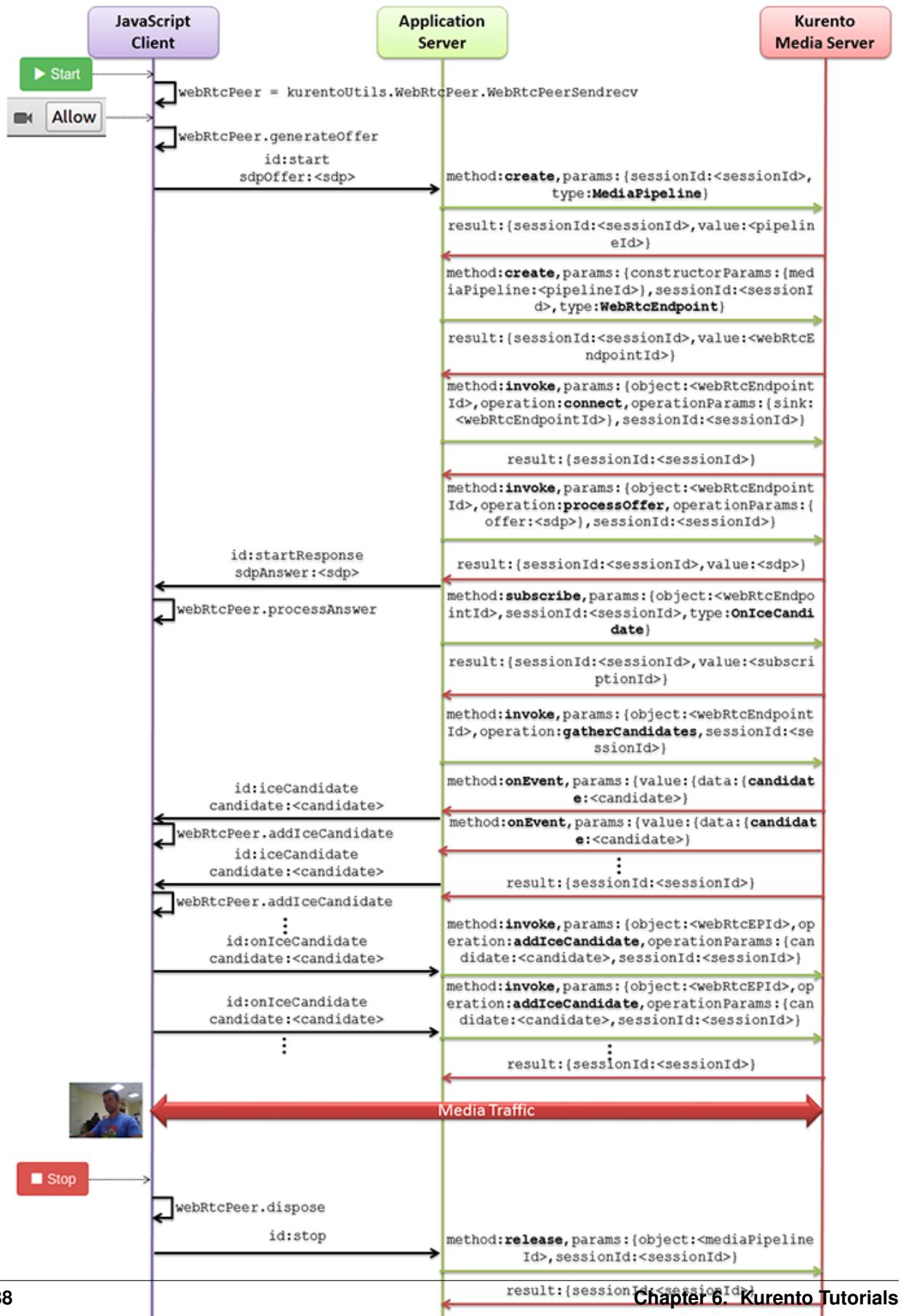
The diagram below shows an complete sequence diagram from the interactions with the application interface to: i) JavaScript logic; ii) Application server logic (which uses the Kurento JavaScript Client); iii) Kurento Media Server.

The following sections analyze in deep the server and client-side code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using the **express** framework for Node.js, but express is not a requirement for Kurento. The main script of this demo is `server.js`.

In order to communicate the JavaScript client and the Node application server a WebSocket is used. The incoming messages to this WebSocket (variable `ws` in the code) are conveniently handled to implemented the signaling protocol depicted in the figure before (i.e. messages `start`, `stop`, `onIceCandidate`).



```

var ws = require('ws');

[...]

var wss = new ws.Server({
  server : server,
  path : '/helloworld'
});

function(ws) {
  var sessionId = null;
  var request = ws.upgradeReq;
  var response = {
    writeHead : {}
  };

  sessionHandler(request, response, function(err) {
    sessionId = request.session.id;
    console.log('Connection received with sessionId ' + sessionId);
  });
}

ws.on('error', function(error) {
  console.log('Connection ' + sessionId + ' error');
  stop(sessionId);
});

ws.on('close', function() {
  console.log('Connection ' + sessionId + ' closed');
  stop(sessionId);
});

ws.on('message', function(_message) {
  var message = JSON.parse(_message);
  console.log('Connection ' + sessionId + ' received message ', message);

  switch (message.id) {
    case 'start':
      sessionId = request.session.id;
      start(sessionId, ws, message.sdpOffer, function(error, sdpAnswer) {
        if (error) {
          return ws.send(JSON.stringify({
            id : 'error',
            message : error
          }));
        }
        ws.send(JSON.stringify({
          id : 'startResponse',
          sdpAnswer : sdpAnswer
        }));
      });
      break;

    case 'stop':
      stop(sessionId);
      break;
  }
});

```

```

        case 'onIceCandidate':
            onIceCandidate(sessionId, message.candidate);
            break;

        default:
            ws.send(JSON.stringify({
                id : 'error',
                message : 'Invalid message ' + message
            }));
            break;
    }

});
);

```

In order to control the media capabilities provided by the Kurento Media Server, we need an instance of the *KurentoClient* in the Node application server. In order to create this instance, we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it's located at *localhost* listening in port 8888.

```

var kurento = require('kurento-client');

var kurentoClient = null;

var argv = minimist(process.argv.slice(2), {
    default: {
        as_uri: 'https://localhost:8443/',
        ws_uri: 'ws://localhost:8888/kurento'
    }
});

[...]

function getKurentoClient(callback) {
    if (kurentoClient !== null) {
        return callback(null, kurentoClient);
    }

    kurento(argv.ws_uri, function(error, _kurentoClient) {
        if (error) {
            console.log("Could not find media server at address " + argv.ws_uri);
            return callback("Could not find media server at address" + argv.ws_uri
                + ". Exiting with error " + error);
        }

        kurentoClient = _kurentoClient;
        callback(null, kurentoClient);
    });
}

```

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server. Our first operation is to create a *Media Pipeline*, then we need to create the *Media Elements* and connect them. In this example, we just need a single *WebRtcEndpoint* connected to itself (i.e. in loopback). These functions are called in the *start* function, which is fired when the *start* message is received:

```

function start(sessionId, ws, sdpOffer, callback) {
    if (!sessionId) {

```

```

        return callback('Cannot use undefined sessionId');
    }

getKurentoClient(function(error, kurentoClient) {
    if (error) {
        return callback(error);
    }

    kurentoClient.create('MediaPipeline', function(error, pipeline) {
        if (error) {
            return callback(error);
        }

        createMediaElements(pipeline, ws, function(error, webRtcEndpoint) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            if (candidatesQueue[sessionId]) {
                while(candidatesQueue[sessionId].length) {
                    var candidate = candidatesQueue[sessionId].shift();
                    webRtcEndpoint.addIceCandidate(candidate);
                }
            }
        });

        connectMediaElements(webRtcEndpoint, function(error) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            webRtcEndpoint.on('OnIceCandidate', function(event) {
                var candidate = kurento.getComplexType('IceCandidate')(event.
→candidate);
                ws.send(JSON.stringify({
                    id : 'iceCandidate',
                    candidate : candidate
                }));
            });
        });

        webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            sessions[sessionId] = {
                'pipeline' : pipeline,
                'webRtcEndpoint' : webRtcEndpoint
            }
            return callback(null, sdpAnswer);
        });
    });

    webRtcEndpoint.gatherCandidates(function(error) {
        if (error) {
            return callback(error);
        }
    })
});

```

```
        });
    });
});
});
});

function createMediaElements(pipeline, ws, callback) {
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            return callback(error);
        }

        return callback(null, webRtcEndpoint);
    });
}

function connectMediaElements(webRtcEndpoint, callback) {
    webRtcEndpoint.connect(webRtcEndpoint, function(error) {
        if (error) {
            return callback(error);
        }
        return callback(null);
    });
}
```

As of Kurento Media Server 6.0, the WebRTC negotiation is done by exchanging *ICE* candidates between the WebRTC peers. To implement this protocol, the `webRtcEndpoint` receives candidates from the client in `onIceCandidate` function. These candidates are stored in a queue when the `webRtcEndpoint` is not available yet. Then these candidates are added to the media element by calling to the `addIceCandidate` method.

```
var candidatesQueue = {};

[...]

function onIceCandidate(sessionId, _candidate) {
    var candidate = kurento.getComplexType('IceCandidate')(_candidate);

    if (sessions[sessionId]) {
        console.info('Sending candidate');
        var webRtcEndpoint = sessions[sessionId].webRtcEndpoint;
        webRtcEndpoint.addIceCandidate(candidate);
    }
    else {
        console.info('Queueing candidate');
        if (!candidatesQueue[sessionId]) {
            candidatesQueue[sessionId] = [];
        }
        candidatesQueue[sessionId].push(candidate);
    }
}
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to

simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application. These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/helloworld`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication.

```
var ws = new WebSocket('ws://' + location.host + '/helloworld');
var webRtcPeer;

const I_CAN_START = 0;
const I_CAN_STOP = 1;
const I_AM_STARTING = 2;

[...]

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Error message from server: ' + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate)
            break;
        default:
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Unrecognized message', parsedMessage);
    }
}
```

In the function `start` the method `WebRtcPeer.WebRtcPeerSendrecv` of *kurento-utils.js* is used to create the `webRtcPeer` object, which is used to handle the WebRTC communication.

```
videoInput = document.getElementById('videoInput');
videoOutput = document.getElementById('videoOutput');

[...]

function start() {
    console.log('Starting video call ...')

    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(videoInput, videoOutput);

    console.log('Creating WebRtcPeer and generating local sdp offer ...');
```

```

var options = {
    localVideo: videoInput,
    remoteVideo: videoOutput,
    onicecandidate : onIceCandidate
}

webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
    if(error) return onError(error);
    this.generateOffer(onOffer);
});
}

function onIceCandidate(candidate) {
    console.log('Local candidate' + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
    };
    sendMessage(message);
}

function onOffer(error, offerSdp) {
    if(error) return onError(error);

    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}

```

Dependencies

Server-side dependencies of this demo are managed using [npm](#). Our main dependency is the Kurento Client JavaScript ([kurento-client](#)). The relevant part of the `package.json` file for managing this dependency is:

```

"dependencies": {
    [...]
    "kurento-client" : "6.7.1"
}

```

At the client side, dependencies are managed using [Bower](#). Take a look to the `bower.json` file and pay attention to the following section:

```

"dependencies": {
    [...]
    "kurento-utils" : "6.7.1"
}

```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [npm](#) and [Bower](#).

6.2 WebRTC Magic Mirror

This web application consists on a *WebRTC loopback* video communication, adding a funny hat over detected faces. This is an example of a Computer Vision and Augmented Reality filter.

6.2.1 Java - WebRTC magic mirror

This web application extends the *Hello World Tutorial*, adding media processing to the basic *WebRTC* loopback.

Note: This tutorial has been configured to use https. Follow the instructions to secure your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the *installation guide* for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-magic-mirror
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This application uses computer vision and augmented reality techniques to add a funny hat on top of faces. The following picture shows a screenshot of the demo running in a web browser:

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

- **WebRtcEndpoint**: Provides full-duplex (bidirectional) *WebRTC* capabilities.
- **FaceOverlay filter**: Computer vision filter that detects faces in the video stream and puts an image on top of them. In this demo the filter is configured to put a *Super Mario* hat.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server**

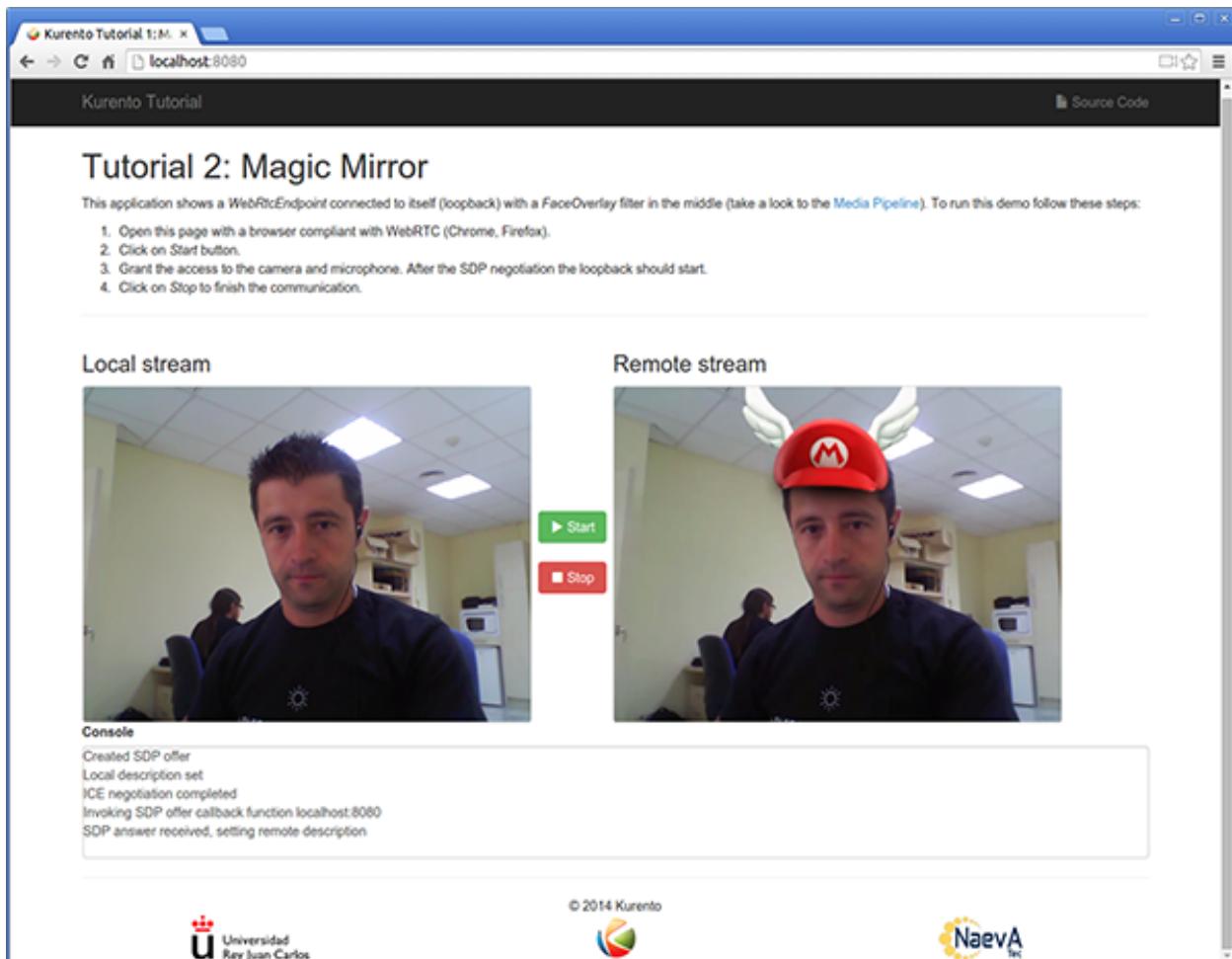


Fig. 6.11: *Kurento Magic Mirror Screenshot: WebRTC with filter in loopback*

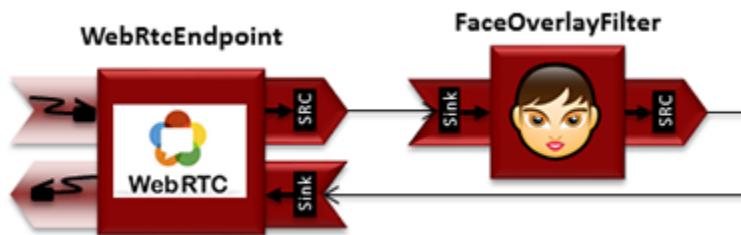


Fig. 6.12: *WebRTC with filter in loopback Media Pipeline*

capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

To communicate the client with the Java EE application server we have designed a simple signaling protocol based on [JSON](#) messages over [WebSocket](#)'s. The normal sequence between client and server is as follows: i) Client starts the Magic Mirror. ii) Client stops the Magic Mirror.

If any exception happens, server sends an error message to the client. The detailed message sequence between client and application server is depicted in the following picture:

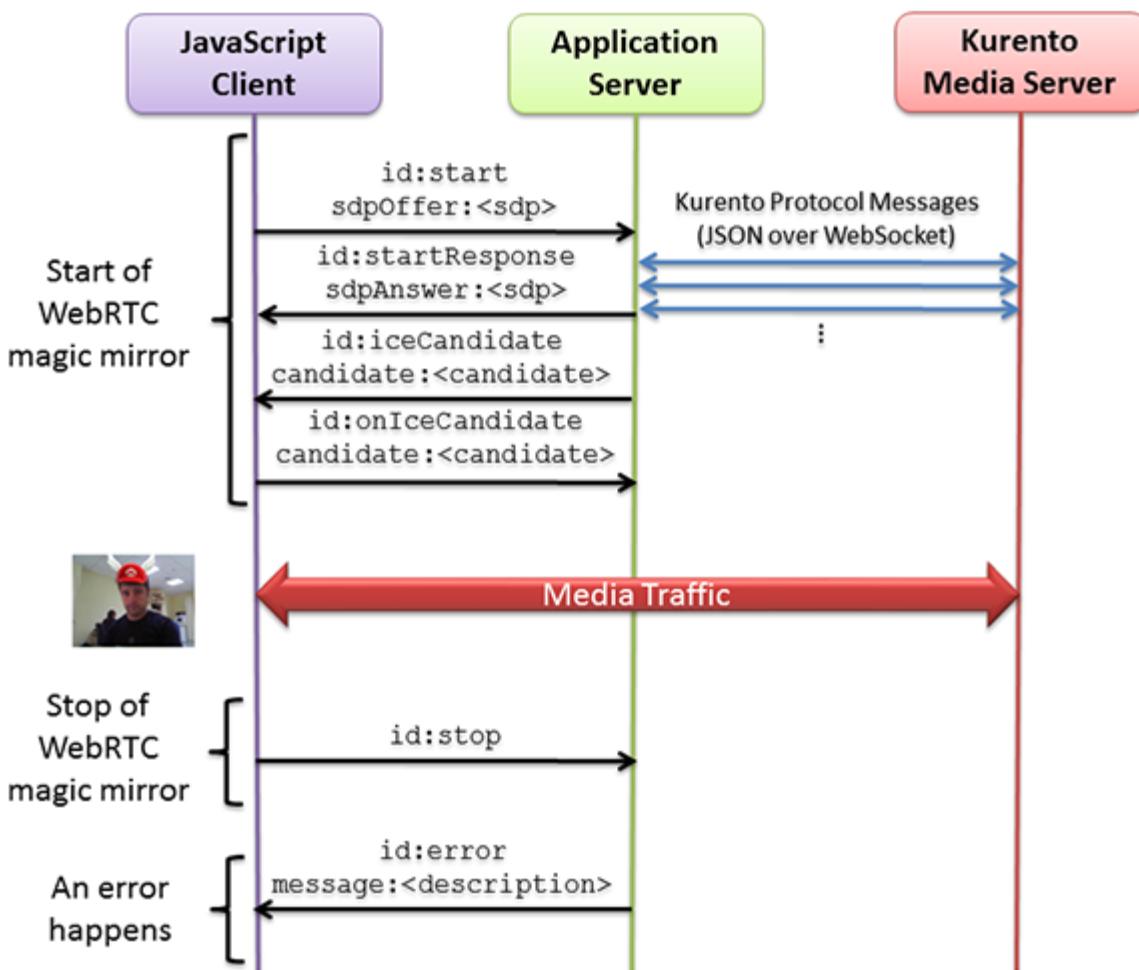


Fig. 6.13: One to one video call signaling protocol

As you can see in the diagram, an [SDP](#) and [ICE](#) candidates needs to be exchanged between client and server to establish the [WebRTC](#) session between the Kurento client and server. Specifically, the SDP negotiation connects the WebRtcPeer at the browser with the WebRtcEndpoint at the server. The complete source code of this demo can be found in [GitHub](#).

Application Server Side

This demo has been developed using **Java** in the server-side, based on the *Spring Boot* framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

In the following figure you can see a class diagram of the server side code:

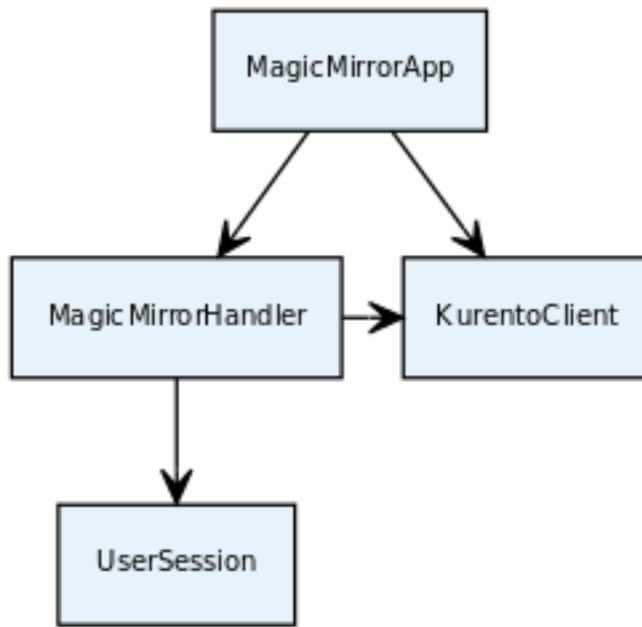


Fig. 6.14: Server-side class diagram of the *MagicMirror* app

The main class of this demo is named `MagicMirrorApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to your applications. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at *localhost*, listening in port 8888. If you reproduce this tutorial, you'll need to insert the specific location of your Kurento Media Server instance there.

```

@EnableWebSocket
@SpringBootApplication
public class MagicMirrorApp implements WebSocketConfigurer {

    final static String DEFAULT_KMS_WS_URI = "ws://localhost:8888/kurento";

    @Bean
    public MagicMirrorHandler handler() {
        return new MagicMirrorHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }
}

```

```

@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(handler(), "/magicmirror");
}

public static void main(String[] args) throws Exception {
    new SpringApplication(MagicMirrorApp.class).run(args);
}
}

```

This web application follows a *Single Page Application* architecture ([SPA](#)), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/magicmirror`.

`MagicMirrorHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kinds of incoming messages to the *Server* : `start`, `stop` and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```

public class MagicMirrorHandler extends TextWebSocketHandler {

    private final Logger log = LoggerFactory.getLogger(MagicMirrorHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, UserSession> users = new ConcurrentHashMap<
        String, UserSession>();

    @Autowired
    private KurentoClient kurento;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {
        JSONObject jsonMessage = gson.fromJson(message.getPayload(), JSONObject.class);

        log.debug("Incoming message: {}", jsonMessage);

        switch (jsonMessage.get("id").getAsString()) {
            case "start":
                start(session, jsonMessage);
                break;
            case "stop": {
                UserSession user = users.remove(session.getId());
                if (user != null) {
                    user.release();
                }
                break;
            }
            case "onIceCandidate": {
                JSONObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

                UserSession user = users.get(session.getId());
                if (user != null) {

```

```

        IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate").
→getAsString(),
                jsonCandidate.get("sdpMid").getAsString(), jsonCandidate.get(
→"sdpMLineIndex").getAsInt());
                user.addCandidate(candidate);
            }
            break;
        }
        default:
            sendError(session, "Invalid message with id " + jsonMessage.get("id").
→getAsString());
            break;
        }
    }

private void start(WebSocketSession session, JsonObject jsonMessage) {
    ...
}

private void sendError(WebSocketSession session, String message) {
    ...
}
}

```

In the following snippet, we can see the `start` method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (`WebRtcEndpoint` and `FaceOverlayFilter`) and make the connections among them. A `startResponse` message is sent back to the client with the SDP answer.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // User session
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
        user.setWebRtcEndpoint(webRtcEndpoint);
        users.put(session.getId(), user);

        // ICE candidates
        webRtcEndpoint.addIceCandidateFoundListener(new EventListener
→<IceCandidateFoundEvent>() {
            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {
                    synchronized (session) {
                        session.sendMessage(new TextMessage(response.toString()));
                    }
                } catch (IOException e) {
                    log.debug(e.getMessage());
                }
            }
        });
    }

    // Media logic
    FaceOverlayFilter faceOverlayFilter = new FaceOverlayFilter.Builder(pipeline).
→build();
}

```

```

        String appServerUrl = System.getProperty("app.server.url", MagicMirrorApp.
        ↪DEFAULT_APP_SERVER_URL);
        faceOverlayFilter.setOverlayedImage(appServerUrl + "/img/mario-wings.png", -0.
        ↪35F, -1.2F, 1.6F, 1.6F);

        webRtcEndpoint.connect(faceOverlayFilter);
        faceOverlayFilter.connect(webRtcEndpoint);

        // SDP negotiation (offer and answer)
        String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
        String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

        JSONObject response = new JSONObject();
        response.addProperty("id", "startResponse");
        response.addProperty("sdpAnswer", sdpAnswer);

        synchronized (session) {
            session.sendMessage(new TextMessage(response.toString()));
        }

        webRtcEndpoint.gatherCandidates();

    } catch (Throwable t) {
        sendError(session, t.getMessage());
    }
}

```

Note: Notice the hat URL is provided by the application server and consumed by the KMS. This logic is assuming that the application server is hosted in local (*localhost*), and by the default the hat URL is <https://localhost:8443/img/mario-wings.png>. If your application server is hosted in a different host, it can be easily changed by means of the configuration parameter `app.server.url`, for example:

```
mvn compile exec:java -Dapp.server.url=https://app_server_host:app_server_port
```

The `sendError` method is quite simple: it sends an `error` message to the client when an exception is caught in the server-side.

```

private void sendError(WebSocketSession session, String message) {
    try {
        JSONObject response = new JSONObject();
        response.addProperty("id", "error");
        response.addProperty("message", message);
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Exception sending message", e);
    }
}

```

Client-Side

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called **kurento-utils.js**

to simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application.

These libraries are linked in the [index.html](#) web page, and are used in the [index.js](#). In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/magicmirror`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```
var ws = new WebSocket('ws://' + location.host + '/magicmirror');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError("Error message from server: " + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (error) {
                    console.error("Error adding candidate: " + error);
                    return;
                }
            });
            break;
        default:
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Unrecognized message', parsedMessage);
    }
}

function start() {
    console.log("Starting video call ...")
    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(videoInput, videoOutput);

    console.log("Creating WebRtcPeer and generating local sdp offer ...");

    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput,
        onicecandidate: onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function(error) {
```

```

        if (error) {
            return console.error(error);
        }
        webRtcPeer.generateOffer(onOffer);
    });
}

function onOffer(offerSdp) {
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}

function onIceCandidate(candidate) {
    console.log("Local candidate" + JSON.stringify(candidate));

    var message = {
        id: 'onIceCandidate',
        candidate: candidate
    };
    sendMessage(message);
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>

```

```
<artifactId>adapter.js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

6.2.2 JavaScript - Magic Mirror

This web application extends the [Hello World Tutorial](#), adding media processing to the basic [WebRTC](#) loopback.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

You'll need to install Kurento Media Server before running this example. Read [installation guide](#) for further information.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#) :

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-magic-mirror
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. Kurento Media Server must use WebSockets over SSL/TLS (WSS), so make sure you check [this](#) too. It is possible to locate the KMS in other machine simple adding the parameter `ws_uri` to the URL:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must be connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

This application uses computer vision and augmented reality techniques to add a funny hat on top of detected faces. The following picture shows a screenshot of the demo running in a web browser:

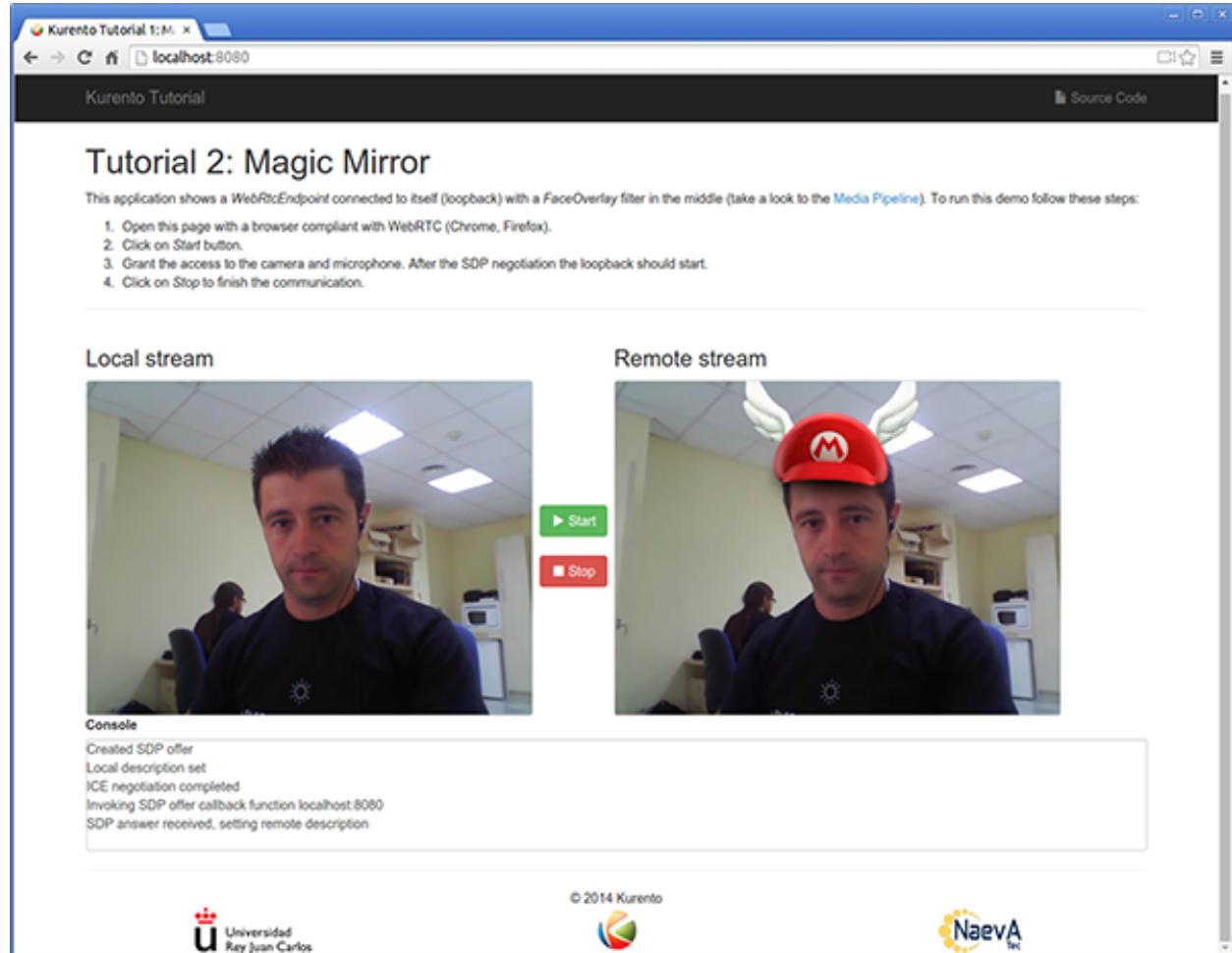


Fig. 6.15: *Kurento Magic Mirror Screenshot: WebRTC with filter in loopback*

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video

camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to the Kurento Media Server, processed and then is returned to the client as a remote stream.

To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

- **WebRtcEndpoint**: Provides full-duplex (bidirectional) *WebRTC* capabilities.
- **FaceOverlay filter**: Computer vision filter that detects faces in the video stream and puts an image on top of them. In this demo the filter is configured to put a *Super Mario hat*.

The media pipeline implemented is illustrated in the following picture:

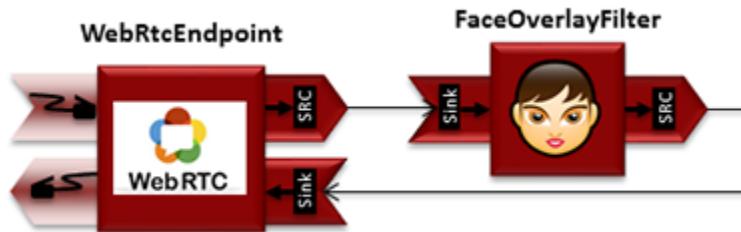


Fig. 6.16: *WebRTC with filter in loopback Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

JavaScript Logic

This demo follows a *Single Page Application* architecture (*SPA*). The interface is the following HTML page: `index.html`. This web page links two Kurento JavaScript libraries:

- **kurento-client.js** : Implementation of the Kurento JavaScript Client.
- **kurento-utils.js** : Kurento utility library aimed to simplify the WebRTC management in the browser.

In addition, these two JavaScript libraries are also required:

- **Bootstrap** : Web framework for developing responsive web sites.
- **jquery.js** : Cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- **adapter.js** : WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **ekko-lightbox** : Module for Bootstrap to open modal images, videos, and galleries.
- **demo-console** : Custom JavaScript console.

The specific logic of this demo is coded in the following JavaScript page: `index.js`. In this file, there is a function which is called when the green button labeled as *Start* in the GUI is clicked.

```

var startButton = document.getElementById("start");

startButton.addEventListener("click", function() {
  var options = {
    localVideo: videoInput,
    remoteVideo: videoOutput
  };

  webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
    if(error) return onError(error)
    this.generateOffer(onOffer)
  })
})
  
```

```
});  
[...]  
}
```

The function `WebRtcPeer.WebRtcPeerSendrecv` abstracts the WebRTC internal details (i.e. `PeerConnection` and `getUserStream`) and makes possible to start a full-duplex WebRTC communication, using the HTML video tag with id `videoInput` to show the video camera (local stream) and the video tag `videoOutput` to show the remote stream provided by the Kurento Media Server.

Inside this function, a call to `generateOffer` is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the `KurentoClient` class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port 8888 at the same host than the HTTP serving the application.

```
[...]  
  
var args = getopt(location.search,  
{  
  default:  
  {  
    ws_uri: 'ws://' + location.hostname + ':8888/kurento',  
    ice_servers: undefined  
  }  
});  
  
[...]  
  
kurentoClient(args.ws_uri, function(error, client){  
  [...]  
});
```

Once we have an instance of `kurentoClient`, the following step is to create a *Media Pipeline*, as follows:

```
client.create("MediaPipeline", function(error, _pipeline){  
  [...]  
});
```

If everything works correctly, we have an instance of a media pipeline (variable `pipeline` in this example). With this instance, we are able to create *Media Elements*. In this example we just need a `WebRtcEndpoint` and a `FaceOverlayFilter`. Then, these media elements are interconnected:

```
pipeline.create('WebRtcEndpoint', function(error, webRtcEp) {  
  if (error) return onError(error);  
  
  setIceCandidateCallbacks(webRtcPeer, webRtcEp, onError)  
  
  webRtcEp.processOffer(sdpOffer, function(error, sdpAnswer) {  
    if (error) return onError(error);  
  
    webRtcPeer.processAnswer(sdpAnswer, onError);  
  });  
  webRtcEp.gatherCandidates(onError);  
  
  pipeline.create('FaceOverlayFilter', function(error, filter) {  
    if (error) return onError(error);  
  
    filter.setOverlaidImage(args.hat_uri, -0.35, -1.2, 1.6, 1.6,
```

```
function(error) {
  if (error) return onError(error);

});

client.connect(webRtcEp, filter, webRtcEp, function(error) {
  if (error) return onError(error);

  console.log("WebRtcEndpoint --> filter --> WebRtcEndpoint");
});
});
});
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},{  
  "urls":"stun:stun2.example.net"}]  
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",  
  "username":"user","credential":"myPassword"}]
```

Dependencies

The dependencies of this demo has to be obtained using *Bower*. The definition of these dependencies are defined in the `bower.json` file, as follows:

```
"dependencies": {  
  "kurento-client": "6.7.1",  
  "kurento-utils": "6.7.1"  
}
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [Bower](#).

6.2.3 Node.js - WebRTC magic mirror

This web application extends the [Hello World Tutorial](#), adding media processing to the basic *WebRTC* loopback.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information.

Be sure to have installed *Node.js* and *Bower* in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-magic-mirror
git checkout 6.7.1
npm install
npm start
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules
npm cache clean
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

This application uses computer vision and augmented reality techniques to add a funny hat on top of faces. The following picture shows a screenshot of the demo running in a web browser:

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

- **WebRtcEndpoint**: Provides full-duplex (bidirectional) *WebRTC* capabilities.
- **FaceOverlay filter**: Computer vision filter that detects faces in the video stream and puts an image on top of them. In this demo the filter is configured to put a *Super Mario hat*.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side we use a Node.js application server consuming the **Kurento JavaScript Client** API to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento JavaScript Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

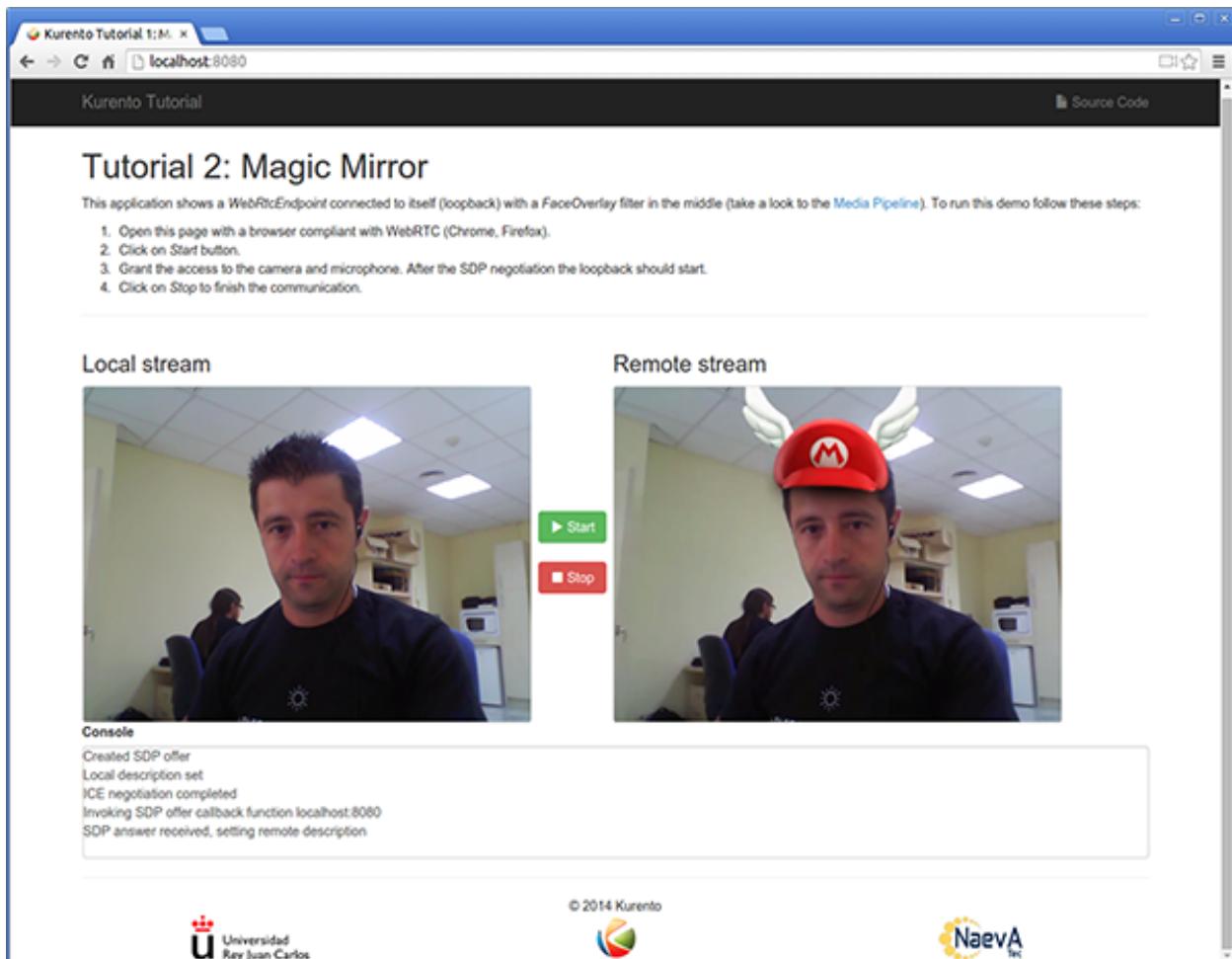


Fig. 6.17: Kurento Magic Mirror Screenshot: WebRTC with filter in loopback

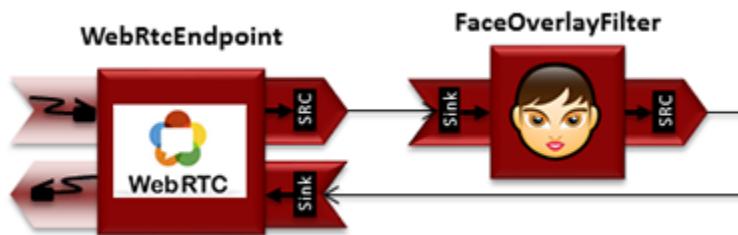


Fig. 6.18: WebRTC with filter in loopback Media Pipeline

To communicate the client with the Node.js application server we have designed a simple signaling protocol based on [JSON](#) messages over [WebSocket](#)'s. The normal sequence between client and server is as follows: i) Client starts the Magic Mirror. ii) Client stops the Magic Mirror.

If any exception happens, server sends an error message to the client. The detailed message sequence between client and application server is depicted in the following picture:

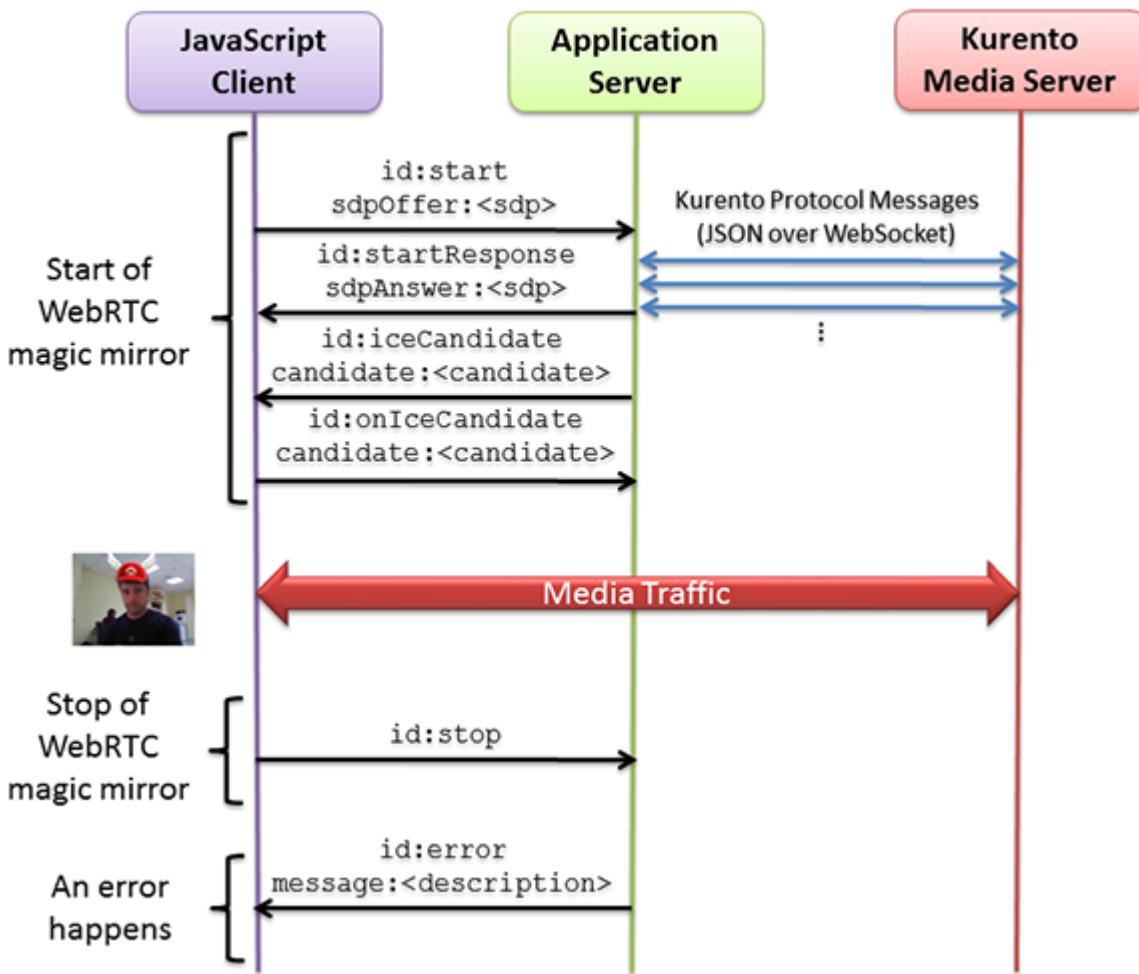


Fig. 6.19: One to one video call signaling protocol

As you can see in the diagram, an [SDP](#) and [ICE](#) candidates needs to be exchanged between client and server to establish the [WebRTC](#) session between the Kurento client and server. Specifically, the SDP negotiation connects the `WebRtcPeer` at the browser with the `WebRtcEndpoint` at the server. The complete source code of this demo can be found in [GitHub](#).

Application Server Logic

This demo has been developed using the `express` framework for Node.js, but express is not a requirement for Kurento. The main script of this demo is `server.js`.

In order to communicate the JavaScript client and the Node application server a [WebSocket](#) is used. The incoming messages to this [WebSocket](#) (variable `ws` in the code) are conveniently handled to implemented the signaling protocol depicted in the figure before (i.e. messages `start`, `stop`, `onIceCandidate`).

```
var ws = require('ws');

[...]

var wss = new ws.Server({
    server : server,
    path : '/magicmirror'
});

/*
 * Management of WebSocket messages
 */
wss.on('connection', function(ws) {
    var sessionId = null;
    var request = ws.upgradeReq;
    var response = {
        writeHead : {}
    };

    sessionHandler(request, response, function(err) {
        sessionId = request.session.id;
        console.log('Connection received with sessionId ' + sessionId);
    });
});

ws.on('error', function(error) {
    console.log('Connection ' + sessionId + ' error');
    stop(sessionId);
});

ws.on('close', function() {
    console.log('Connection ' + sessionId + ' closed');
    stop(sessionId);
});

ws.on('message', function(_message) {
    var message = JSON.parse(_message);
    console.log('Connection ' + sessionId + ' received message ', message);

    switch (message.id) {
        case 'start':
            sessionId = request.session.id;
            start(sessionId, ws, message.sdpOffer, function(error, sdpAnswer) {
                if (error) {
                    return ws.send(JSON.stringify({
                        id : 'error',
                        message : error
                    }));
                }
                ws.send(JSON.stringify({
                    id : 'startResponse',
                    sdpAnswer : sdpAnswer
                }));
            });
            break;

        case 'stop':
            stop(sessionId);
            break;
    }
});
```

```

        case 'onIceCandidate':
            onIceCandidate(sessionId, message.candidate);
            break;

        default:
            ws.send(JSON.stringify({
                id : 'error',
                message : 'Invalid message ' + message
            }));
            break;
    }

});
);

```

In order to control the media capabilities provided by the Kurento Media Server, we need an instance of the *KurentoClient* in the Node application server. In order to create this instance, we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it's located at *localhost* listening in port 8888.

```

var kurento = require('kurento-client');

var kurentoClient = null;

var argv = minimist(process.argv.slice(2), {
    default: {
        as_uri: 'https://localhost:8443/',
        ws_uri: 'ws://localhost:8888/kurento'
    }
});

[...]

function getKurentoClient(callback) {
    if (kurentoClient !== null) {
        return callback(null, kurentoClient);
    }

    kurento(argv.ws_uri, function(error, _kurentoClient) {
        if (error) {
            console.log("Could not find media server at address " + argv.ws_uri);
            return callback("Could not find media server at address" + argv.ws_uri
                           + ". Exiting with error " + error);
        }

        kurentoClient = _kurentoClient;
        callback(null, kurentoClient);
    });
}

```

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server. Our first operation is to create a *Media Pipeline*, then we need to create the *Media Elements* and connect them. In this example, we need a *WebRtcEndpoint* connected to a *FaceOverlayFilter*, which is connected to the sink of the same *WebRtcEndpoint*. These functions are called in the *start* function, which is fired when the *start* message is received:

```

function start(sessionId, ws, sdpOffer, callback) {
    if (!sessionId) {
        return callback('Cannot use undefined sessionId');
    }

    getKurentoClient(function(error, kurentoClient) {
        if (error) {
            return callback(error);
        }

        kurentoClient.create('MediaPipeline', function(error, pipeline) {
            if (error) {
                return callback(error);
            }

            createMediaElements(pipeline, ws, function(error, webRtcEndpoint) {
                if (error) {
                    pipeline.release();
                    return callback(error);
                }

                if (candidatesQueue[sessionId]) {
                    while(candidatesQueue[sessionId].length) {
                        var candidate = candidatesQueue[sessionId].shift();
                        webRtcEndpoint.addIceCandidate(candidate);
                    }
                }

                connectMediaElements(webRtcEndpoint, faceOverlayFilter, function(error) {
                    if (error) {
                        pipeline.release();
                        return callback(error);
                    }

                    webRtcEndpoint.on('OnIceCandidate', function(event) {
                        var candidate = kurento.getComplexType('IceCandidate')(event);
                        ws.send(JSON.stringify({
                            id : 'iceCandidate',
                            candidate : candidate
                        }));
                    });

                    webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
                        if (error) {
                            pipeline.release();
                            return callback(error);
                        }

                        sessions[sessionId] = {
                            'pipeline' : pipeline,
                            'webRtcEndpoint' : webRtcEndpoint
                        }
                        return callback(null, sdpAnswer);
                    });
                }

                webRtcEndpoint.gatherCandidates(function(error) {

```

```

                if (error) {
                    return callback(error);
                }
            });
        });
    });
}
}

function createMediaElements(pipeline, ws, callback) {
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            return callback(error);
        }

        return callback(null, webRtcEndpoint);
    });
}

function connectMediaElements(webRtcEndpoint, faceOverlayFilter, callback) {
    webRtcEndpoint.connect(faceOverlayFilter, function(error) {
        if (error) {
            return callback(error);
        }

        faceOverlayFilter.connect(webRtcEndpoint, function(error) {
            if (error) {
                return callback(error);
            }

            return callback(null);
        });
    });
}
}

```

As of Kurento Media Server 6.0, the WebRTC negotiation is done by exchanging *ICE* candidates between the WebRTC peers. To implement this protocol, the `webRtcEndpoint` receives candidates from the client in `onIceCandidate` function. These candidates are stored in a queue when the `webRtcEndpoint` is not available yet. Then these candidates are added to the media element by calling to the `addIceCandidate` method.

```

var candidatesQueue = {};

[...]

function onIceCandidate(sessionId, _candidate) {
    var candidate = kurento.getComplexType('IceCandidate')(_candidate);

    if (sessions[sessionId]) {
        console.info('Sending candidate');
        var webRtcEndpoint = sessions[sessionId].webRtcEndpoint;
        webRtcEndpoint.addIceCandidate(candidate);
    }
    else {
        console.info('Queueing candidate');
        if (!candidatesQueue[sessionId]) {
            candidatesQueue[sessionId] = [];
        }
    }
}

```

```

        candidatesQueue[sessionId].push(candidate);
    }
}

```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application. These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/magicmirror`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication.

```

var ws = new WebSocket('ws://' + location.host + '/magicmirror');
var webRtcPeer;

const I_CAN_START = 0;
const I_CAN_STOP = 1;
const I_AM_STARTING = 2;

[...]

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Error message from server: ' + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate);
            break;
        default:
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Unrecognized message', parsedMessage);
    }
}

```

In the function `start` the method `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to create the `webRtcPeer` object, which is used to handle the WebRTC communication.

```

videoInput = document.getElementById('videoInput');
videoOutput = document.getElementById('videoOutput');

```

```
[...]

function start() {
    console.log('Starting video call ...')

    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(videoInput, videoOutput);

    console.log('Creating WebRtcPeer and generating local sdp offer ...');

    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput,
        onicecandidate : onIceCandidate
    }

    webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
        if(error) return onError(error);
        this.generateOffer(onOffer);
    });
}

function onIceCandidate(candidate) {
    console.log('Local candidate' + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
    };
    sendMessage(message);
}

function onOffer(error, offerSdp) {
    if(error) return onError(error);

    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}
```

Dependencies

Server-side dependencies of this demo are managed using [npm](#). Our main dependency is the Kurento Client JavaScript (*kurento-client*). The relevant part of the `package.json` file for managing this dependency is:

```
"dependencies": {
    [...]
    "kurento-client" : "6.7.1"
}
```

At the client side, dependencies are managed using [Bower](#). Take a look to the `bower.json` file and pay attention to the following section:

```
"dependencies": {  
    [...]  
    "kurento-utils" : "6.7.1"  
}
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [npm](#) and [Bower](#).

6.3 RTP Receiver

This web application showcases reception of an incoming RTP or SRTP stream, and playback via a WebRTC connection.

6.3.1 Kurento Java Tutorial - RTP Receiver

This web application consists of a simple RTP stream pipeline: an *RtpEndpoint* is configured in KMS to listen for one incoming video stream. This stream must be generated by an external program. Visual feedback is provided in this page, by connecting the *RtpEndpoint* to a *WebRtcEndpoint* in receive-only mode.

The Java client application **connects to all events** emitted from KMS and prints log messages for each one, so this application is also a good reference to understand what are those events and their relationship with how KMS works. Check [Endpoint Events](#) for more information about events that can be emitted by KMS.

Note: This application uses HTTPS. It will work fine if you run it in `localhost` and accept a security exception in the browser, but you should secure your application if running remotely. For more info, check [Configure Java applications to use HTTPS](#).

Quick start

Follow these steps to run this demo application:

1. Install Kurento Media Server: [Installation Guide](#).
2. Run these commands:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git  
cd kurento-tutorial-java/kurento-hello-world  
git checkout 6.7.1  
mvn clean compile exec:java -Dkms.url=ws://localhost:8888/kurento
```

3. Open the demo page with a WebRTC-compliant browser (Chrome, Firefox): <https://localhost:8443/>
4. Click on *Start* to begin the demo.
5. Copy the KMS IP and Port information to the external streaming program.
6. As soon as the external streaming program starts sending RTP packets to the IP and Port where KMS is listening for incoming data, the video should appear in the page.
7. Click on *Stop* to finish the demo.

Understanding this example

To implement this behavior we have to create a [Media Pipeline](#), composed of an **RtpEndpoint** and a **WebRtcEndpoint**. The former acts as an RTP receiver, and the later is used to show the incoming video in the demo page.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier.

To communicate these entities, two WebSockets channels are used:

1. WebSocket is created between the browser client and the application server to implement a custom signaling protocol.
2. Another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server.

This communication takes place using the **Kurento Protocol**. For further information, see [Kurento Protocol](#).

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the [Spring Boot](#) framework, which embeds a Tomcat web server within the resulting program, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

This graph shows the class diagram of the client application:

Client-Side Logic

We use a specific Kurento JavaScript library called **kurento-utils.js** to simplify the WebRTC interaction between browser and server application. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences.

These libraries are linked in the *index.html* page, and are used in the *index.js* file.

6.4 WebRTC One-To-Many broadcast

Video broadcasting for [WebRTC](#). One peer transmits a video stream and N peers receive it.

6.4.1 Java - One to many video call

This web application consists on a one-to-many video call using [WebRTC](#) technology. In other words, it is an implementation of a video broadcasting web application.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

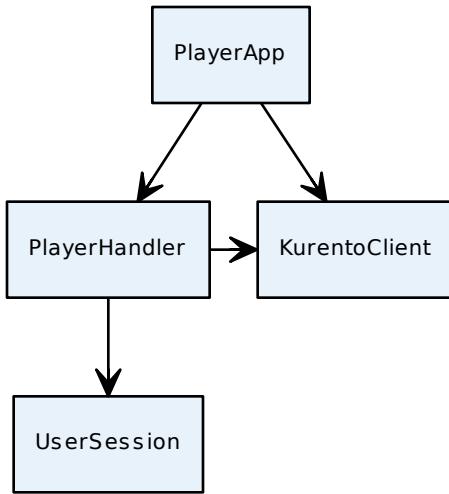


Fig. 6.20: Server-side class diagram of the client application

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-one2many-call
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

There will be two types of users in this application: 1 peer sending media (let's call it *Presenter*) and N peers receiving the media from the *Presenter* (let's call them *Viewers*). Thus, the Media Pipeline is composed by 1+N interconnected *WebRtcEndpoints*. The following picture shows an screenshot of the *Presenter*'s web GUI:

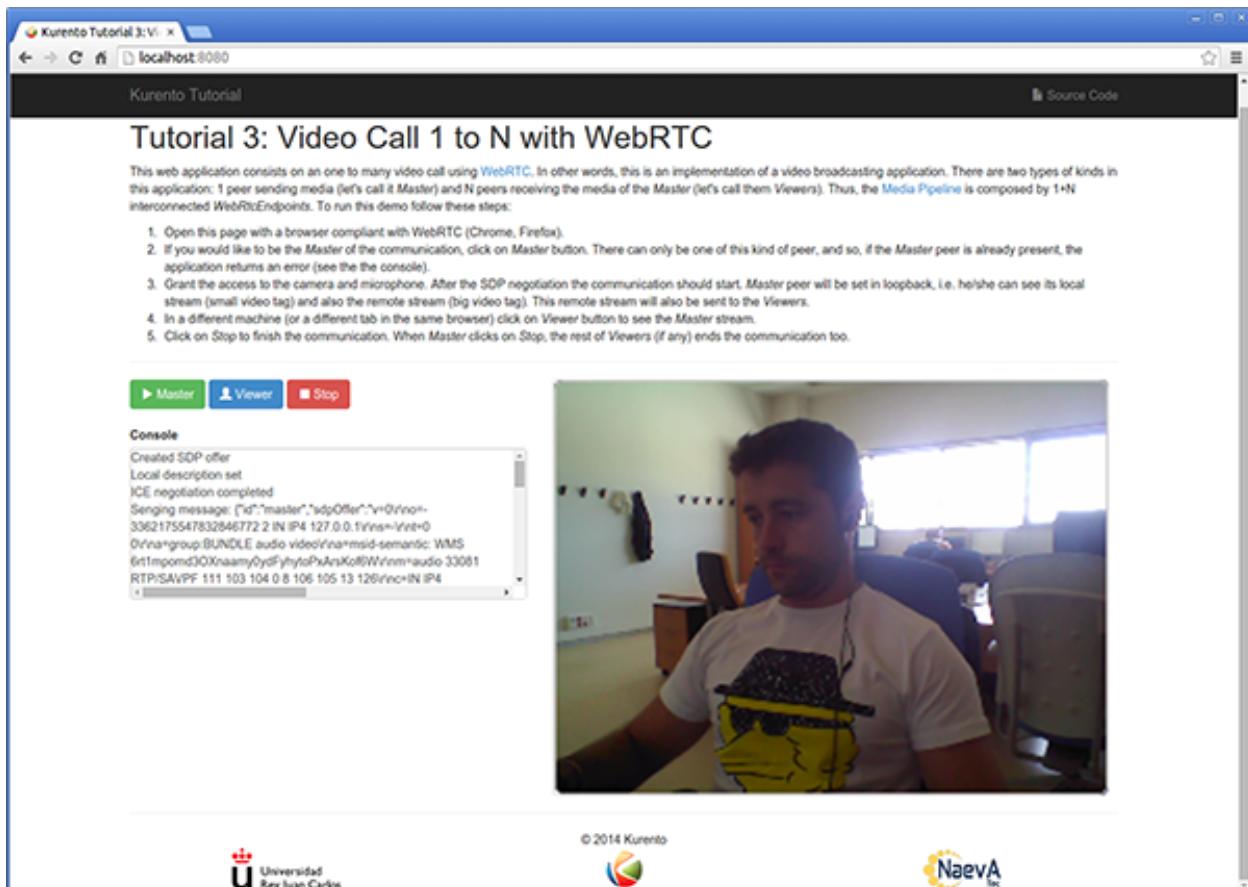


Fig. 6.21: One to many video call screenshot

To implement this behavior we have to create a *Media Pipeline* composed by 1+N **WebRtcEndpoints**. The *Presenter* peer sends its stream to the rest of the *Viewers*. *Viewers* are configured in receive-only mode. The implemented media pipeline is illustrated in the following picture:

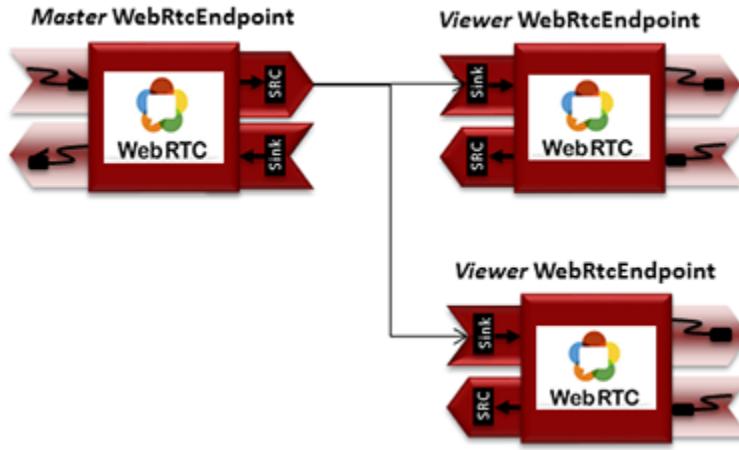


Fig. 6.22: *One to many video call Media Pipeline*

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities two WebSockets are used. First, a WebSocket is created between client and server-side to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication is implemented by the **Kurento Protocol**. For further information, please see this [page](#).

Client and application server communicate using a signaling protocol based on **JSON** messages over **WebSocket**'s. The normal sequence between client and server is as follows:

1. A *Presenter* enters in the system. There must be one and only one *Presenter* at any time. For that, if a *Presenter* has already present, an error message is sent if another user tries to become *Presenter*.
2. N *Viewers* connect to the presenter. If no *Presenter* is present, then an error is sent to the corresponding *Viewer*.
3. *Viewers* can leave the communication at any time.
4. When the *Presenter* finishes the session each connected *Viewer* receives an *stopCommunication* message and also terminates its session.

We can draw the following sequence diagram with detailed messages between clients and server:

As you can see in the diagram, **SDP** and **ICE** candidates need to be exchanged between client and server to establish the **WebRTC** connection between the Kurento client and server. Specifically, the SDP negotiation connects the **WebRtcPeer** in the browser with the **WebRtcEndpoint** in the server. The complete source code of this demo can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the **Spring Boot** framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For

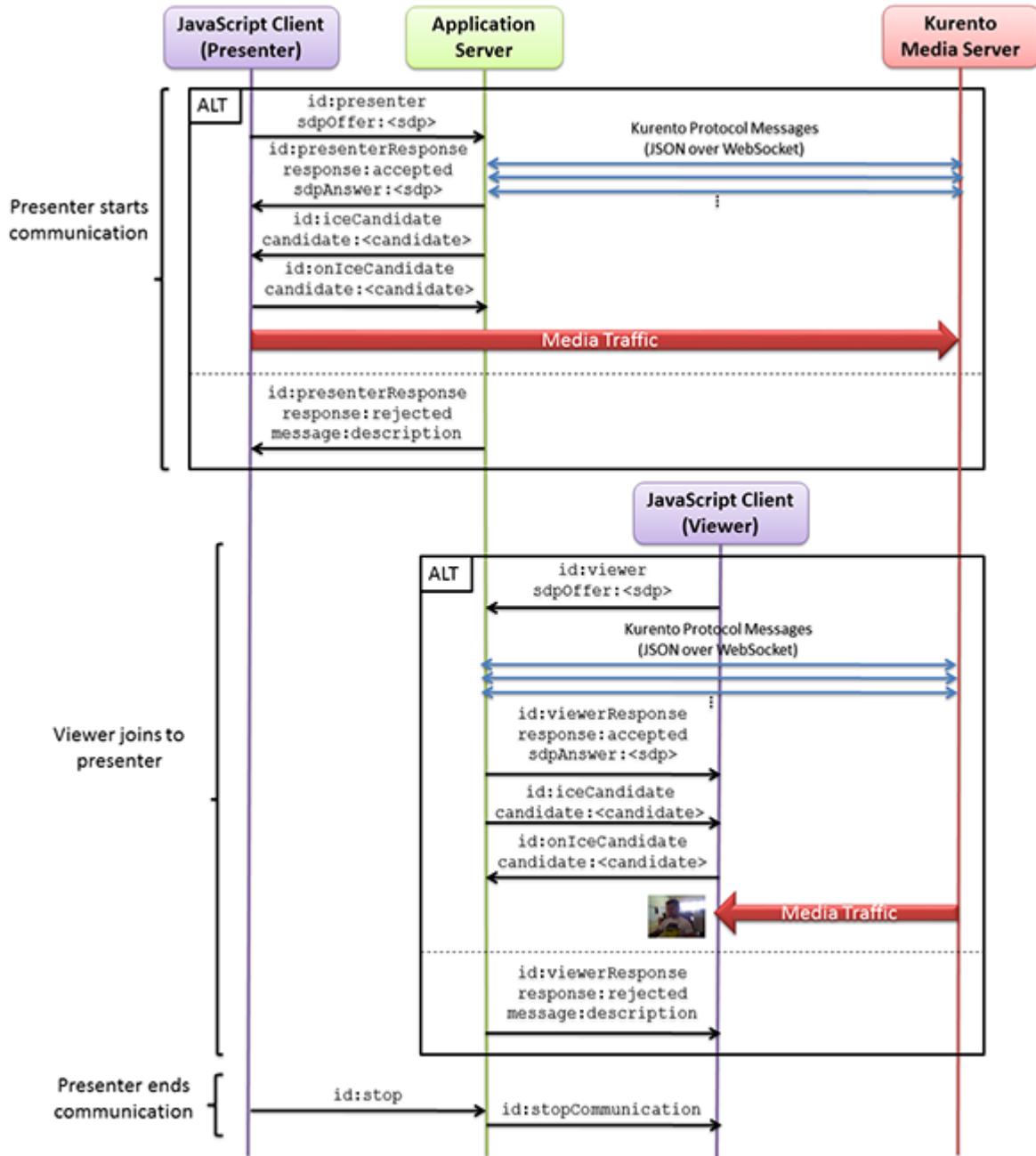


Fig. 6.23: One to many video call signaling protocol

example, a pure Java EE application, SIP Servlets, Play, Vertex, etc. We chose Spring Boot for convenience.

In the following, figure you can see a class diagram of the server side code:

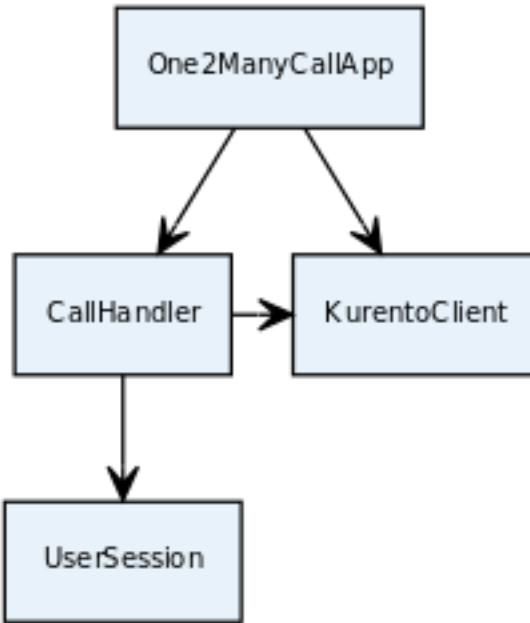


Fig. 6.24: Server-side class diagram of the One2Many app

The main class of this demo is named `One2ManyCallApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to your applications. In this instantiation we see that a WebSocket is used to connect with Kurento Media Server, by default in the *localhost* and listening in the port 8888.

```

@EnableWebSocket
@SpringBootApplication
public class One2ManyCallApp implements WebSocketConfigurer {

    @Bean
    public CallHandler callHandler() {
        return new CallHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(callHandler(), "/call");
    }

    public static void main(String[] args) throws Exception {
        new SpringApplication(One2ManyCallApp.class).run(args);
    }
}
  
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/call`.

`CallHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kind of incoming messages to the *Server*: `presenter`, `viewer`, `stop`, and `onIceCandidate`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```
public class CallHandler extends TextWebSocketHandler {

    private static final Logger log = LoggerFactory.getLogger(CallHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, UserSession> viewers = new
    ↪ConcurrentHashMap<String, UserSession>();

    @Autowired
    private KurentoClient kurento;

    private MediaPipeline pipeline;
    private UserSession presenterUserSession;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) ↪
    ↪throws Exception {
        JSONObject jsonMessage = gson.fromJson(message.getPayload(), JSONObject.class);
        log.debug("Incoming message from session '{}': {}", session.getId(), ↪
        ↪jsonMessage);

        switch (jsonMessage.get("id").getAsString()) {
            case "presenter":
                try {
                    presenter(session, jsonMessage);
                } catch (Throwable t) {
                    handleErrorResponse(t, session, "presenterResponse");
                }
                break;
            case "viewer":
                try {
                    viewer(session, jsonMessage);
                } catch (Throwable t) {
                    handleErrorResponse(t, session, "viewerResponse");
                }
                break;
            case "onIceCandidate":
                JsonObject candidate = jsonMessage.get("candidate").getAsJsonObject();

                UserSession user = null;
                if (presenterUserSession != null) {
                    if (presenterUserSession.getSession() == session) {
                        user = presenterUserSession;
                    } else {
                        user = viewers.get(session.getId());
                    }
                }
        }
    }
}
```

```

        if (user != null) {
            IceCandidate cand = new IceCandidate(candidate.get("candidate") .
→getAsString(),
                candidate.get("sdpMid").getAsString(), candidate.get("sdpMLineIndex"
→").getAsInt());
            user.addCandidate(cand);
        }
        break;
    }
    case "stop":
        stop(session);
        break;
    default:
        break;
}
}

private void handleErrorResponse(Throwable t, WebSocketSession session,
    String responseId) throws IOException {
    stop(session);
    log.error(t.getMessage(), t);
    JSONObject response = new JSONObject();
    response.addProperty("id", responseId);
    response.addProperty("response", "rejected");
    response.addProperty("message", t.getMessage());
    session.sendMessage(new TextMessage(response.toString()));
}

private synchronized void presenter(final WebSocketSession session, JSONObject_
→jsonMessage) throws IOException {
    ...
}

private synchronized void viewer(final WebSocketSession session, JSONObject_
→jsonMessage) throws IOException {
    ...
}

private synchronized void stop(WebSocketSession session) throws IOException {
    ...
}

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status)_
throws Exception {
    stop(session);
}

}

```

In the following snippet, we can see the `presenter` method. It creates a Media Pipeline and the WebRtcEndpoint for `presenter`:

```

private synchronized void presenter(final WebSocketSession session, JSONObject_
→jsonMessage) throws IOException {
    if (presenterUserSession == null) {
        presenterUserSession = new UserSession(session);

```

```

        pipeline = kurento.createMediaPipeline();
        presenterUserSession.setWebRtcEndpoint(new WebRtcEndpoint.Builder(pipeline).
        ↪build());
    }

    WebRtcEndpoint presenterWebRtc = presenterUserSession.getWebRtcEndpoint();

    presenterWebRtc.addIceCandidateFoundListener(new EventListener
    ↪<IceCandidateFoundEvent>() {

        @Override
        public void onEvent(IceCandidateFoundEvent event) {
            JSONObject response = new JSONObject();
            response.addProperty("id", "iceCandidate");
            response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
            try {
                synchronized (session) {
                    session.sendMessage(new TextMessage(response.toString()));
                }
            } catch (IOException e) {
                log.debug(e.getMessage());
            }
        }
    });
}

String sdpOffer = jsonMessage.getAsJsonPrimitive("sdpOffer").getAsString();
String sdpAnswer = presenterWebRtc.processOffer(sdpOffer);

JSONObject response = new JSONObject();
response.addProperty("id", "presenterResponse");
response.addProperty("response", "accepted");
response.addProperty("sdpAnswer", sdpAnswer);

synchronized (session) {
    presenterUserSession.sendMessage(response);
}
presenterWebRtc.gatherCandidates();

} else {
    JSONObject response = new JSONObject();
    response.addProperty("id", "presenterResponse");
    response.addProperty("response", "rejected");
    response.addProperty("message", "Another user is currently acting as sender."+
    ↪Try again later ...);
    session.sendMessage(new TextMessage(response.toString()));
}
}
}

```

The viewer method is similar, but note the *Presenter* WebRtcEndpoint is connected to each of the viewers WebRtcEndpoints, otherwise an error is sent back to the client.

```

private synchronized void viewer(final WebSocketSession session, JSONObject
    ↪jsonMessage) throws IOException {
    if (presenterUserSession == null || presenterUserSession.getWebRtcEndpoint() ==
    ↪null) {
        JSONObject response = new JSONObject();
        response.addProperty("id", "viewerResponse");
        response.addProperty("response", "rejected");
        response.addProperty("message", "No active sender now. Become sender or . Try" +
        ↪again later ...);
    }
}

```

```

        session.sendMessage(new TextMessage(response.toString()));
    } else {
        if (viewers.containsKey(session.getId())) {
            JSONObject response = new JSONObject();
            response.addProperty("id", "viewerResponse");
            response.addProperty("response", "rejected");
            response.addProperty("message",
                "You are already viewing in this session. Use a different browser to"
                + "add additional viewers.");
            session.sendMessage(new TextMessage(response.toString()));
            return;
        }
        UserSession viewer = new UserSession(session);
        viewers.put(session.getId(), viewer);

        String sdpOffer = jsonMessage.getAsJsonPrimitive("sdpOffer").getAsString();

        WebRtcEndpoint nextWebRtc = new WebRtcEndpoint.Builder(pipeline).build();

        nextWebRtc.addIceCandidateFoundListener(new EventListener
        <IceCandidateFoundEvent>() {

            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JSONObject response = new JSONObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {
                    synchronized (session) {
                        session.sendMessage(new TextMessage(response.toString()));
                    }
                } catch (IOException e) {
                    log.debug(e.getMessage());
                }
            }
        });

        viewer.setWebRtcEndpoint(nextWebRtc);
        presenterUserSession.getWebRtcEndpoint().connect(nextWebRtc);
        String sdpAnswer = nextWebRtc.processOffer(sdpOffer);

        JSONObject response = new JSONObject();
        response.addProperty("id", "viewerResponse");
        response.addProperty("response", "accepted");
        response.addProperty("sdpAnswer", sdpAnswer);

        synchronized (session) {
            viewer.sendMessage(response);
        }
        nextWebRtc.gatherCandidates();
    }
}

```

Finally, the stop message finishes the communication. If this message is sent by the *Presenter*, a stopCommunication message is sent to each connected *Viewer*:

```

private synchronized void stop(WebSocketSession session) throws IOException {
    String sessionId = session.getId();

```

```
if (presenterUserSession != null && presenterUserSession.getSession().getId() == sessionId) {
    for (UserSession viewer : viewers.values()) {
        JSONObject response = new JSONObject();
        response.addProperty("id", "stopCommunication");
        viewer.sendMessage(response);
    }
}

log.info("Releasing media pipeline");
if (pipeline != null) {
    pipeline.release();
}
pipeline = null;
presenterUserSession = null;
} else if (viewers.containsKey(sessionId)) {
    if (viewers.get(sessionId).getWebRtcEndpoint() != null) {
        viewers.get(sessionId).getWebRtcEndpoint().release();
    }
    viewers.remove(sessionId);
}
}
```

Client-Side

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/call`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are four incoming messages to client: `presenterResponse`, `viewerResponse`, `iceCandidate`, and `stopCommunication`. Convenient actions are taken to implement each step in the communication. For example, in the function `presenter` the function `WebRtcPeer.WebRtcPeerSendonly` of *kurento-utils.js* is used to start a WebRTC communication. Then, `WebRtcPeer.WebRtcPeerRecvonly` is used in the `viewer` function.

```
var ws = new WebSocket('ws://' + location.host + '/call');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'presenterResponse':
            presenterResponse(parsedMessage);
            break;
        case 'viewerResponse':
            viewerResponse(parsedMessage);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function (error) {
                if (!error) return;
                console.error("Error adding candidate: " + error);
            });
    }
}
```

```

        break;
    case 'stopCommunication':
        dispose();
        break;
    default:
        console.error('Unrecognized message', parsedMessage);
    }
}

function presenter() {
    if (!webRtcPeer) {
        spinner(video);

        var options = {
            localVideo: video,
            onicecandidate: onIceCandidate
        }
        webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendonly(options,
            function (error) {
                if(error) {
                    return console.error(error);
                }
                webRtcPeer.generateOffer(onOfferPresenter);
            });
    }
}

function viewer() {
    if (!webRtcPeer) {
        spinner(video);

        var options = {
            remoteVideo: video,
            onicecandidate: onIceCandidate
        }
        webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
            function (error) {
                if(error) {
                    return console.error(error);
                }
                this.generateOffer(onOfferViewer);
            });
    }
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>

```

```

</dependency>
<dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-utils-js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>bootstrap</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>demo-console</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>adapter.js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```

<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>

```

6.4.2 Node.js - One to many video call

This web application consists on one-to-many video call using **WebRTC** technology. In other words, it is an implementation of a video broadcasting web application.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information.

Be sure to have installed **Node.js** and **Bower** in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-one2many-call
git checkout 6.7.1
npm install
npm start
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules
npm cache clean
```

Access the application connecting to the URL <https://localhost:8443> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

There will be two types of users in this application: 1 peer sending media (let's call it *Presenter*) and N peers receiving the media from the *Presenter* (let's call them *Viewers*). Thus, the Media Pipeline is composed by 1+N interconnected *WebRtcEndpoints*. The following picture shows an screenshot of the *Presenter*'s web GUI:

To implement this behavior we have to create a *Media Pipeline* composed by 1+N **WebRtcEndpoints**. The *Presenter* peer sends its stream to the rest of the *Viewers*. *Viewers* are configured in receive-only mode. The implemented media pipeline is illustrated in the following picture:

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side we use the **Kurento JavaScript Client** in order to reach the **Kurento Media Server**. All in all, the high level architecture of this demo is three-tier. To communicate these entities two WebSockets are used. The first is created between the client browser and a Node.js application server to transport signaling messages. The second is used to communicate the Kurento JavaScript Client executing at Node.js and the Kurento Media Server. This communication is implemented by the **Kurento Protocol**. For further information, please see this [page](#).

Client and application server communicate using a signaling protocol based on **JSON** messages over **WebSocket**'s. The normal sequence between client and server is as follows:

1. A *Presenter* enters in the system. There must be one and only one *Presenter* at any time. For that, if a *Presenter* has already present, an error message is sent if another user tries to become *Presenter*.
2. N *Viewers* connect to the presenter. If no *Presenter* is present, then an error is sent to the corresponding *Viewer*.

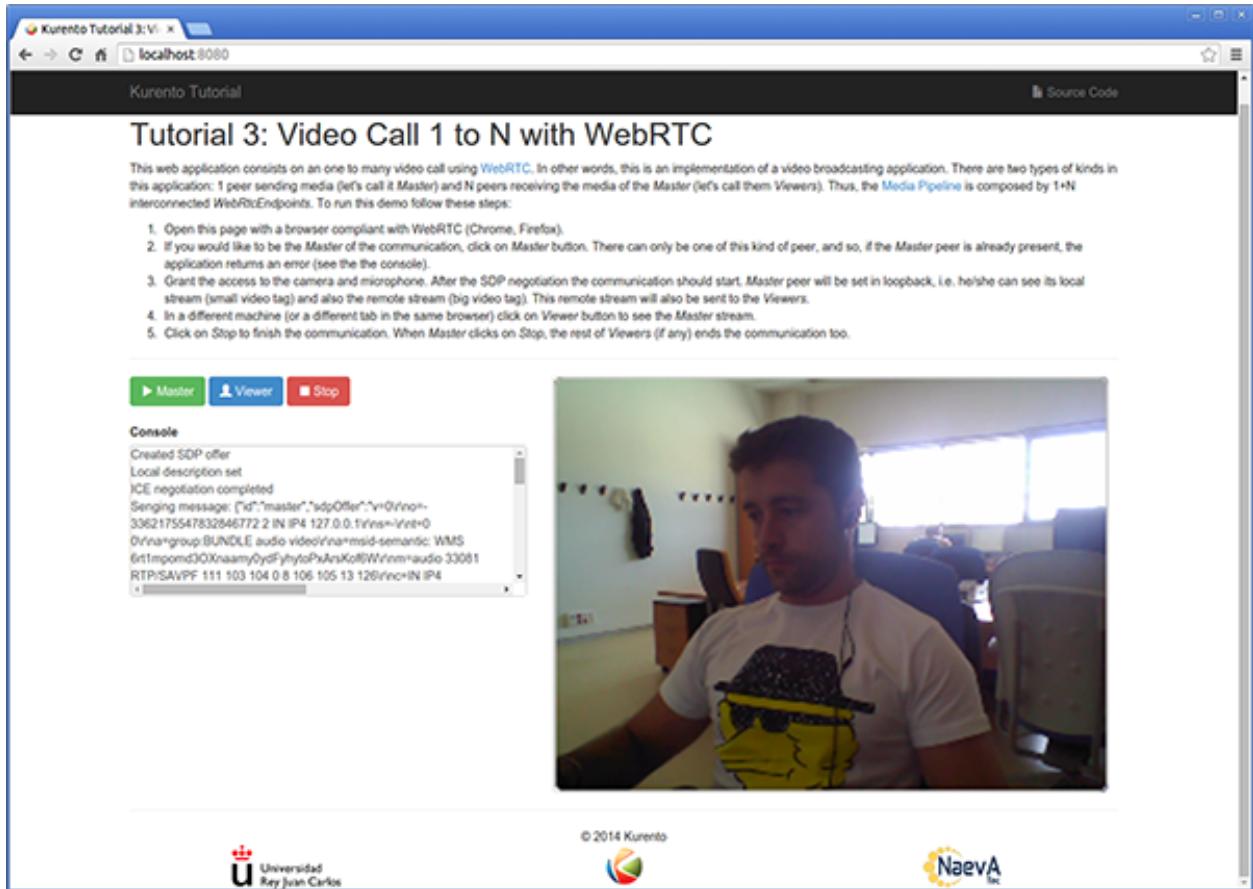


Fig. 6.25: *One to many video call screenshot*

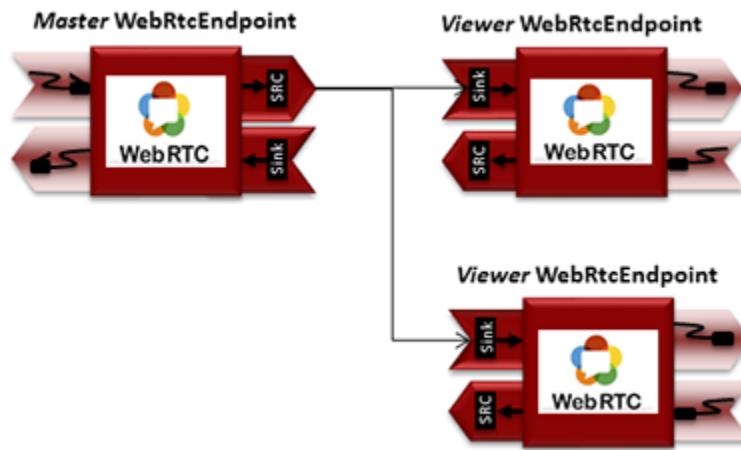


Fig. 6.26: One to many video call Media Pipeline

3. *Viewers* can leave the communication at any time.
4. When the *Presenter* finishes the session each connected *Viewer* receives an *stopCommunication* message and also terminates its session.

We can draw the following sequence diagram with detailed messages between clients and server:

As you can see in the diagram, *SDP* and *ICE* candidates need to be exchanged between client and server to establish the *WebRTC* connection between the Kurento client and server. Specifically, the SDP negotiation connects the WebRtcPeer in the browser with the WebRtcEndpoint in the server. The complete source code of this demo can be found in [GitHub](#).

Application Server Logic

This demo has been developed using the **express** framework for Node.js, but express is not a requirement for Kurento. The main script of this demo is `server.js`.

In order to communicate the JavaScript client and the Node application server a WebSocket is used. The incoming messages to this WebSocket (variable `ws` in the code) are conveniently handled to implemented the signaling protocol depicted in the figure before (i.e. messages `presenter`, `viewer`, `stop`, and `onIceCandidate`).

```
var ws = require('ws');

[...]

var wss = new ws.Server({
  server : server,
  path : '/one2many'
});

/*
 * Management of WebSocket messages
 */
wss.on('connection', function(ws) {

  var sessionId = nextUniqueId();
  console.log('Connection received with sessionId ' + sessionId);

  ws.on('error', function(error) {
    console.log('Connection ' + sessionId + ' error');
    stop(sessionId);
  });

  ws.on('close', function() {
    console.log('Connection ' + sessionId + ' closed');
    stop(sessionId);
  });

  ws.on('message', function(_message) {
    var message = JSON.parse(_message);
    console.log('Connection ' + sessionId + ' received message ', message);

    switch (message.id) {
      case 'presenter':
        startPresenter(sessionId, ws, message.sdpOffer, function(error, sdpAnswer) {
          if (error) {
            return ws.send(JSON.stringify({
              id : 'presenterResponse',
              response : 'rejected',
            }));
          }
        });
    }
  });
});
```

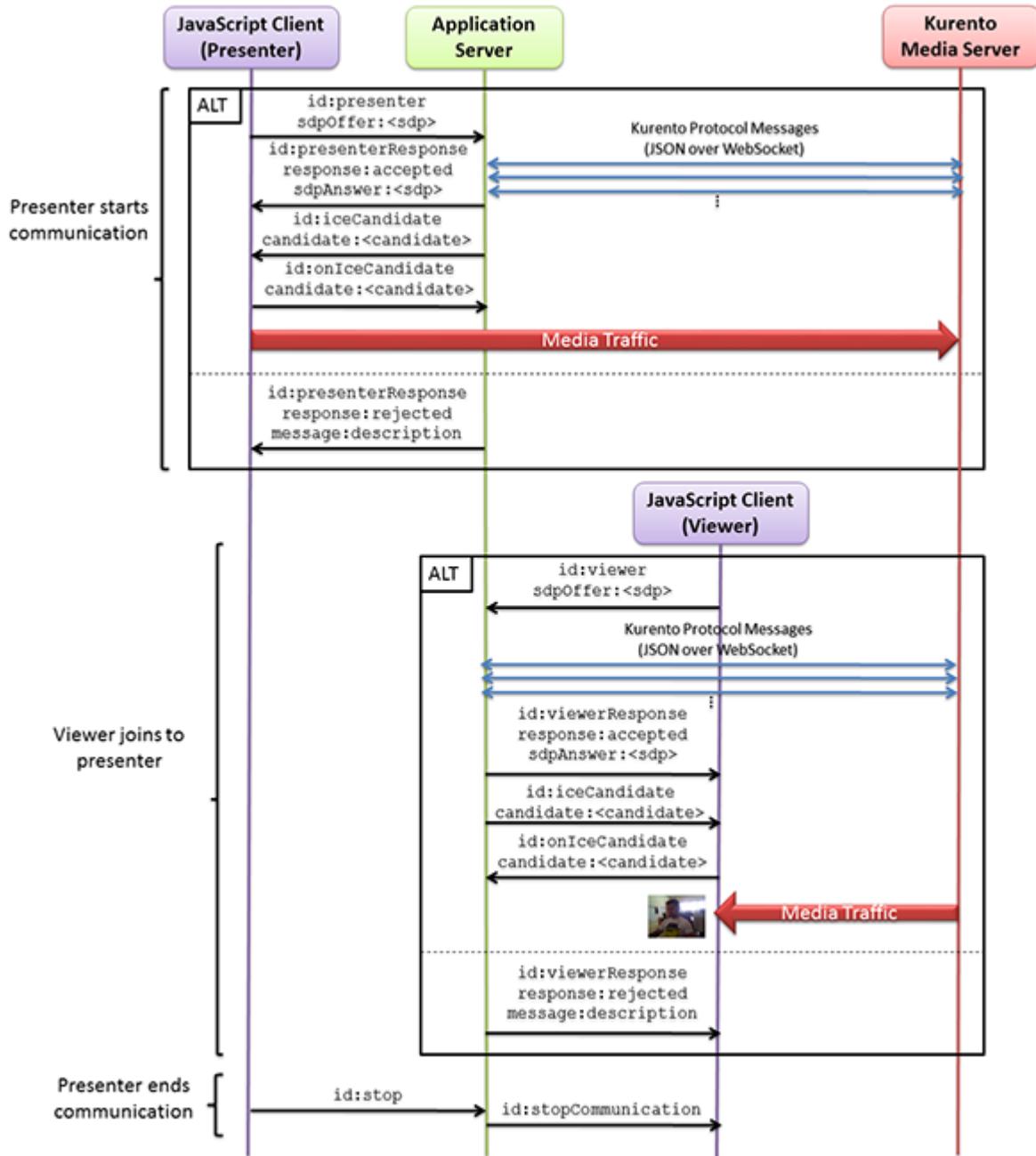


Fig. 6.27: One to many video call signaling protocol

```

        message : error
    }));
}
ws.send(JSON.stringify({
    id : 'presenterResponse',
    response : 'accepted',
    sdpAnswer : sdpAnswer
)));
});
break;

case 'viewer':
startViewer(sessionId, ws, message.sdpOffer, function(error, sdpAnswer) {
    if (error) {
        return ws.send(JSON.stringify({
            id : 'viewerResponse',
            response : 'rejected',
            message : error
        }));
    }
    ws.send(JSON.stringify({
        id : 'viewerResponse',
        response : 'accepted',
        sdpAnswer : sdpAnswer
    )));
});
break;

case 'stop':
stop(sessionId);
break;

case 'onIceCandidate':
onIceCandidate(sessionId, message.candidate);
break;

default:
ws.send(JSON.stringify({
    id : 'error',
    message : 'Invalid message ' + message
}));
break;
}
});
});

```

In order to control the media capabilities provided by the Kurento Media Server, we need an instance of the *KurentoClient* in the Node application server. In order to create this instance, we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it's located at *localhost* listening in port 8888.

```

var kurento = require('kurento-client');

var kurentoClient = null;

var argv = minimist(process.argv.slice(2), {
    default: {
        as_uri: 'https://localhost:8443/',

```

```

        ws_uri: 'ws://localhost:8888/kurento'
    }
});

[...]

function getKurentoClient(callback) {
    if (kurentoClient !== null) {
        return callback(null, kurentoClient);
    }

    kurento(argv.ws_uri, function(error, _kurentoClient) {
        if (error) {
            console.log("Could not find media server at address " + argv.ws_uri);
            return callback("Could not find media server at address" + argv.ws_uri
                + ". Exiting with error " + error);
        }

        kurentoClient = _kurentoClient;
        callback(null, kurentoClient);
    });
}
}

```

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server. Our first operation is to create a *Media Pipeline*, then we need to create the *Media Elements* and connect them. In this example, we need a *WebRtcEndpoint* (in send-only mode) for the presenter connected to N *WebRtcEndpoint* (in receive-only mode) for the viewers. These functions are called in the `startPresenter` and `startViewer` function, which is fired when the `presenter` and `viewer` message are received respectively:

```

function startPresenter(sessionId, ws, sdpOffer, callback) {
    clearCandidatesQueue(sessionId);

    if (presenter !== null) {
        stop(sessionId);
        return callback("Another user is currently acting as presenter. Try again later\u2192");
    }

    presenter = {
        id : sessionId,
        pipeline : null,
        webRtcEndpoint : null
    }

    getKurentoClient(function(error, kurentoClient) {
        if (error) {
            stop(sessionId);
            return callback(error);
        }

        if (presenter === null) {
            stop(sessionId);
            return callback(noPresenterMessage);
        }

        kurentoClient.create('MediaPipeline', function(error, pipeline) {
            if (error) {
                stop(sessionId);
            }
        });
    });
}

```

```

        return callback(error);
    }

    if (presenter === null) {
        stop(sessionId);
        return callback(noPresenterMessage);
    }

    presenter.pipeline = pipeline;
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            stop(sessionId);
            return callback(error);
        }

        if (presenter === null) {
            stop(sessionId);
            return callback(noPresenterMessage);
        }

        presenter.webRtcEndpoint = webRtcEndpoint;

        if (candidatesQueue[sessionId]) {
            while(candidatesQueue[sessionId].length) {
                var candidate = candidatesQueue[sessionId].shift();
                webRtcEndpoint.addIceCandidate(candidate);
            }
        }

        webRtcEndpoint.on('OnIceCandidate', function(event) {
            var candidate = kurento.getComplexType('IceCandidate')(event.
candidate);
            ws.send(JSON.stringify({
                id : 'iceCandidate',
                candidate : candidate
            }));
        });
    });

    webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
        if (error) {
            stop(sessionId);
            return callback(error);
        }

        if (presenter === null) {
            stop(sessionId);
            return callback(noPresenterMessage);
        }

        callback(null, sdpAnswer);
    });

    webRtcEndpoint.gatherCandidates(function(error) {
        if (error) {
            stop(sessionId);
            return callback(error);
        }
    });
}

```

```

        });
    });
});
}

function startViewer(sessionId, ws, sdpOffer, callback) {
    clearCandidatesQueue(sessionId);

    if (presenter === null) {
        stop(sessionId);
        return callback(noPresenterMessage);
    }

    presenter.pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            stop(sessionId);
            return callback(error);
        }
        viewers[sessionId] = {
            "webRtcEndpoint" : webRtcEndpoint,
            "ws" : ws
        }

        if (presenter === null) {
            stop(sessionId);
            return callback(noPresenterMessage);
        }

        if (candidatesQueue[sessionId]) {
            while(candidatesQueue[sessionId].length) {
                var candidate = candidatesQueue[sessionId].shift();
                webRtcEndpoint.addIceCandidate(candidate);
            }
        }

        webRtcEndpoint.on('OnIceCandidate', function(event) {
            var candidate = kurento.getComplexType('IceCandidate')(event.candidate);
            ws.send(JSON.stringify({
                id : 'iceCandidate',
                candidate : candidate
            }));
        });
    });

    webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
        if (error) {
            stop(sessionId);
            return callback(error);
        }
        if (presenter === null) {
            stop(sessionId);
            return callback(noPresenterMessage);
        }

        presenter.webRtcEndpoint.connect(webRtcEndpoint, function(error) {
            if (error) {
                stop(sessionId);
                return callback(error);
            }
        });
    });
}

```

```
        if (presenter === null) {
            stop(sessionId);
            return callback(noPresenterMessage);
        }

        callback(null, sdpAnswer);
        webRtcEndpoint.gatherCandidates(function(error) {
            if (error) {
                stop(sessionId);
                return callback(error);
            }
        });
    });
});
```

As of Kurento Media Server 6.0, the WebRTC negotiation is done by exchanging *ICE* candidates between the WebRTC peers. To implement this protocol, the `webRtcEndpoint` receives candidates from the client in `OnIceCandidate` function. These candidates are stored in a queue when the `webRtcEndpoint` is not available yet. Then these candidates are added to the media element by calling to the `addIceCandidate` method.

```
var candidatesQueue = {};  
[...]  
  
function onIceCandidate(sessionId, _candidate) {  
    var candidate = kurento.getComplexType('IceCandidate')(_candidate);  
  
    if (presenter && presenter.id === sessionId && presenter.webRtcEndpoint) {  
        console.info('Sending presenter candidate');  
        presenter.webRtcEndpoint.addIceCandidate(candidate);  
    }  
    else if (viewers[sessionId] && viewers[sessionId].webRtcEndpoint) {  
        console.info('Sending viewer candidate');  
        viewers[sessionId].webRtcEndpoint.addIceCandidate(candidate);  
    }  
    else {  
        console.info('Queueing candidate');  
        if (!candidatesQueue[sessionId]) {  
            candidatesQueue[sessionId] = [];  
        }  
        candidatesQueue[sessionId].push(candidate);  
    }  
}  
  
function clearCandidatesQueue(sessionId) {  
    if (candidatesQueue[sessionId]) {  
        delete candidatesQueue[sessionId];  
    }  
}
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called **kurento-utils.js** to

simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application. These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/one2many`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `presenterResponse`, `viewerResponse`, “`stopCommunication`”, and `iceCandidate`. Convenient actions are taken to implement each step in the communication.

On the one hand, the function `presenter` uses the method `WebRtcPeer.WebRtcPeerSendonly` of *kurento-utils.js* to start a WebRTC communication in send-only mode. On the other hand, the function `viewer` uses the method `WebRtcPeer.WebRtcPeerRecvonly` of *kurento-utils.js* to start a WebRTC communication in receive-only mode.

```
var ws = new WebSocket('ws://' + location.host + '/one2many');
var webRtcPeer;

const I_CAN_START = 0;
const I_CAN_STOP = 1;
const I_AM_STARTING = 2;

[...]

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'presenterResponse':
            presenterResponse(parsedMessage);
            break;
        case 'viewerResponse':
            viewerResponse(parsedMessage);
            break;
        case 'stopCommunication':
            dispose();
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate)
            break;
        default:
            console.error('Unrecognized message', parsedMessage);
    }
}

function presenterResponse(message) {
    if (message.response != 'accepted') {
        var errorMsg = message.message ? message.message : 'Unknown error';
        console.warn('Call not accepted for the following reason: ' + errorMsg);
        dispose();
    } else {
        webRtcPeer.processAnswer(message.sdpAnswer);
    }
}

function viewerResponse(message) {
    if (message.response != 'accepted') {
        var errorMsg = message.message ? message.message : 'Unknown error';
        console.warn('Call not accepted for the following reason: ' + errorMsg);
    }
}
```

```
        dispose();
    } else {
        webRtcPeer.processAnswer(message.sdpAnswer);
    }
}
```

On the one hand, the function `presenter` uses the method `WebRtcPeer.WebRtcPeerSendonly` of `kurento-utils.js` to start a WebRTC communication in send-only mode. On the other hand, the function `viewer` uses the method `WebRtcPeer.WebRtcPeerRecvonly` of `kurento-utils.js` to start a WebRTC communication in receive-only mode.

```

function presenter() {
    if (!webRtcPeer) {
        showSpinner(video);

        var options = {
            localVideo: video,
            onicecandidate : onIceCandidate
        }

        webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendonly(options, onSuccess);
        function(error) {
            if(error) return onError(error);

            this.generateOffer(onOfferPresenter);
        });
    }
}

function onOfferPresenter(error, offerSdp) {
    if (error) return onError(error);

    var message = {
        id : 'presenter',
        sdpOffer : offerSdp
    };
    sendMessage(message);
}

function viewer() {
    if (!webRtcPeer) {
        showSpinner(video);

        var options = {
            remoteVideo: video,
            onicecandidate : onIceCandidate
        }

        webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options, onSuccess);
        function(error) {
            if(error) return onError(error);

            this.generateOffer(onOfferViewer);
        });
    }
}

function onOfferViewer(error, offerSdp) {

```

```

if (error) return onError(error)

var message = {
  id : 'viewer',
  sdpOffer : offerSdp
}
sendMessage(message);
}

```

Dependencies

Server-side dependencies of this demo are managed using [npm](#). Our main dependency is the Kurento Client JavaScript ([kurento-client](#)). The relevant part of the `package.json` file for managing this dependency is:

```

"dependencies": {
  [...]
  "kurento-client" : "6.7.1"
}

```

At the client side, dependencies are managed using [Bower](#). Take a look to the `bower.json` file and pay attention to the following section:

```

"dependencies": {
  [...]
  "kurento-utils" : "6.7.1"
}

```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [npm](#) and [Bower](#).

6.5 WebRTC One-To-One video call

This web application is a videophone (call one to one) based on [WebRTC](#).

6.5.1 Java - One to one video call

This web application consists on a one-to-one video call using [WebRTC](#) technology. In other words, this application provides a simple video softphone.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-one2one-call
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

The following picture shows an screenshot of this demo running in a web browser:

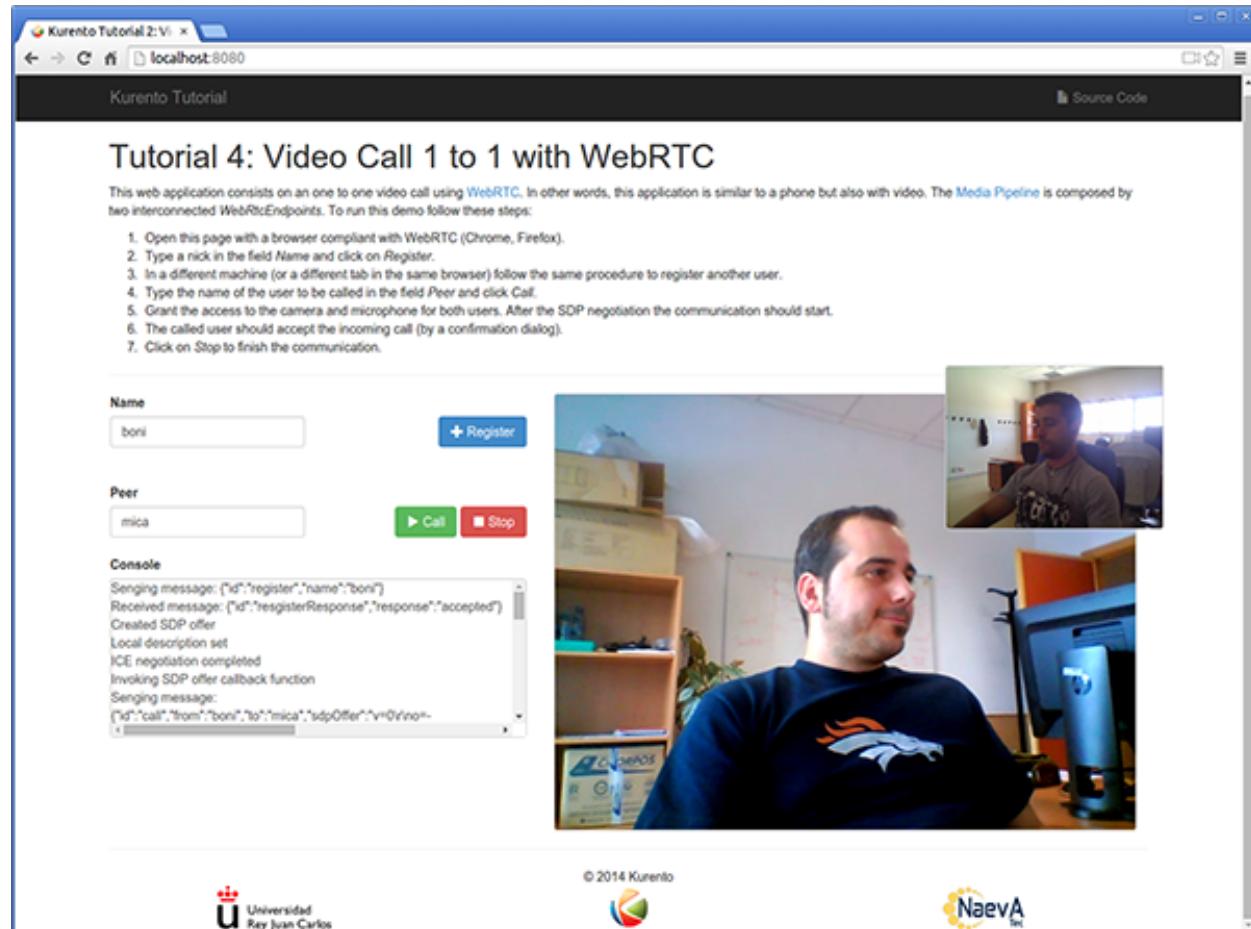


Fig. 6.28: One to one video call screenshot

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the local stream and other for the remote peer stream). If two users, A and B, are using the application, the media flow goes this way:

The video camera stream of user A is sent to the Kurento Media Server, which sends it to user B. In the same way, B sends to Kurento Media Server, which forwards it to A. This means that KMS is providing a B2B (back-to-back) call service.

To implement this behavior, create a [Media Pipeline](#) composed by two WebRtc endpoints connected in B2B. The implemented media pipeline is illustrated in the following picture:

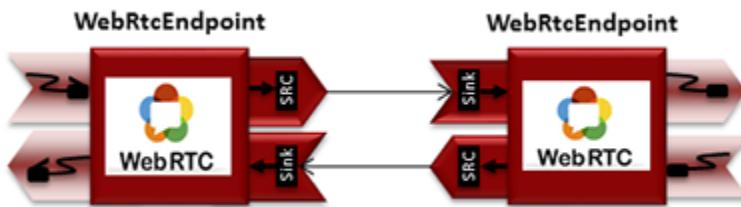


Fig. 6.29: One to one video call Media Pipeline

The client and the server communicate through a signaling protocol based on [JSON](#) messages over [WebSocket](#)'s. The normal sequence between client and server would be as follows:

1. User A is registered in the server with his name
2. User B is registered in the server with her name
3. User A wants to call to User B
4. User B accepts the incoming call
5. The communication is established and media is flowing between User A and User B
6. One of the users finishes the video communication

The detailed message flow in a call are shown in the picture below:

As you can see in the diagram, [SDP](#) and [ICE](#) candidates need to be interchanged between client and server to establish the [WebRTC](#) connection between the Kurento client and server. Specifically, the SDP negotiation connects the WebRtcPeer in the browser with the WebRtcEndpoint in the server.

The following sections describe in detail the server-side, the client-side, and how to run the demo. The complete source code of this demo can be found in [GitHub](#).

Application Server Logic

This demo has been developed using [Java](#) in the server-side, based on the [Spring Boot](#) framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vertx, etc. We have choose Spring Boot for convenience.

In the following figure you can see a class diagram of the server side code:

The main class of this demo is named [One2OneCallApp](#). As you can see, the [KurentoClient](#) is instantiated in this class as a Spring Bean.

```

@EnableWebSocket
@SpringBootApplication
public class One2OneCallApp implements WebSocketConfigurer {

```

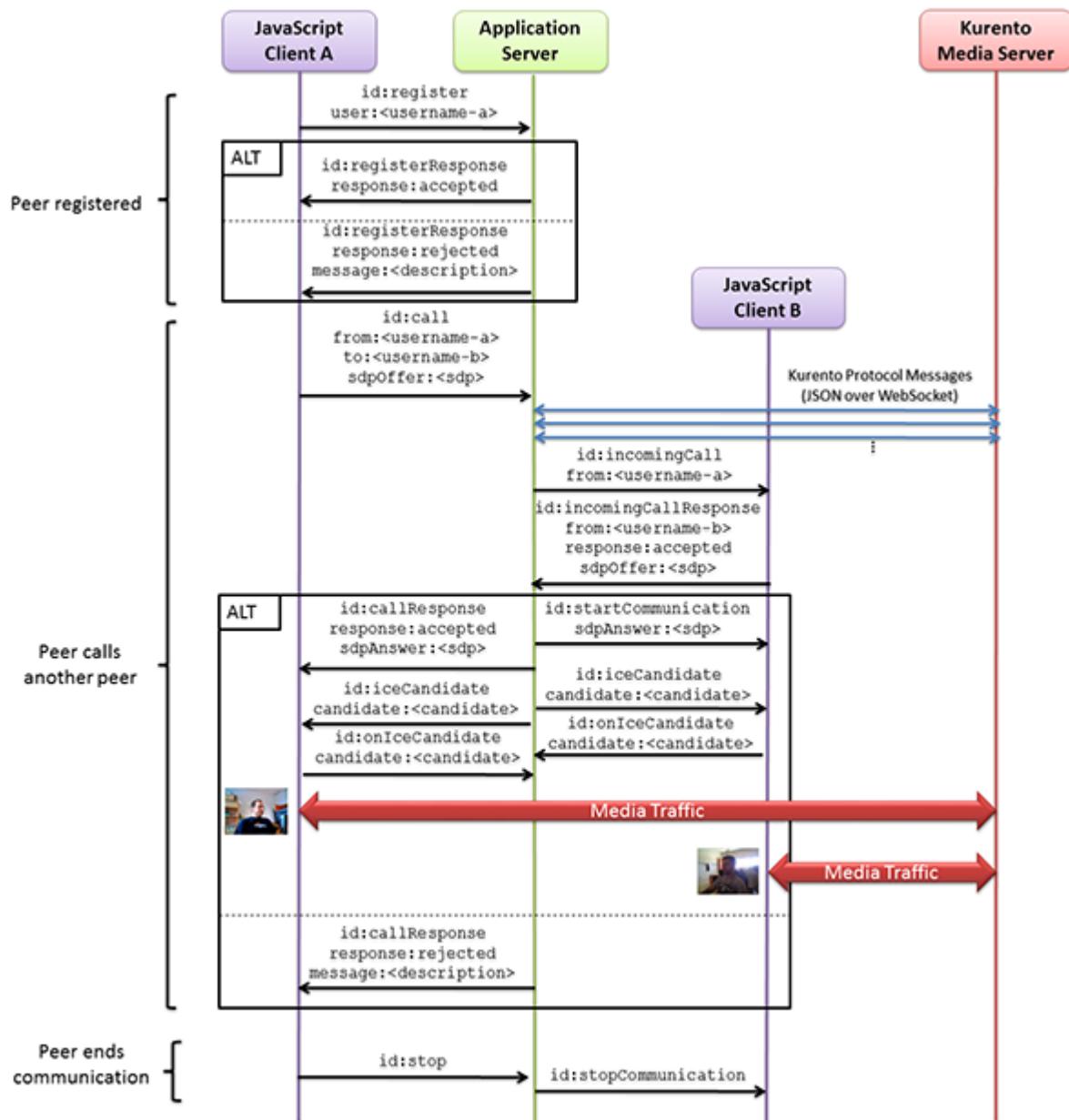


Fig. 6.30: One to many one call signaling protocol

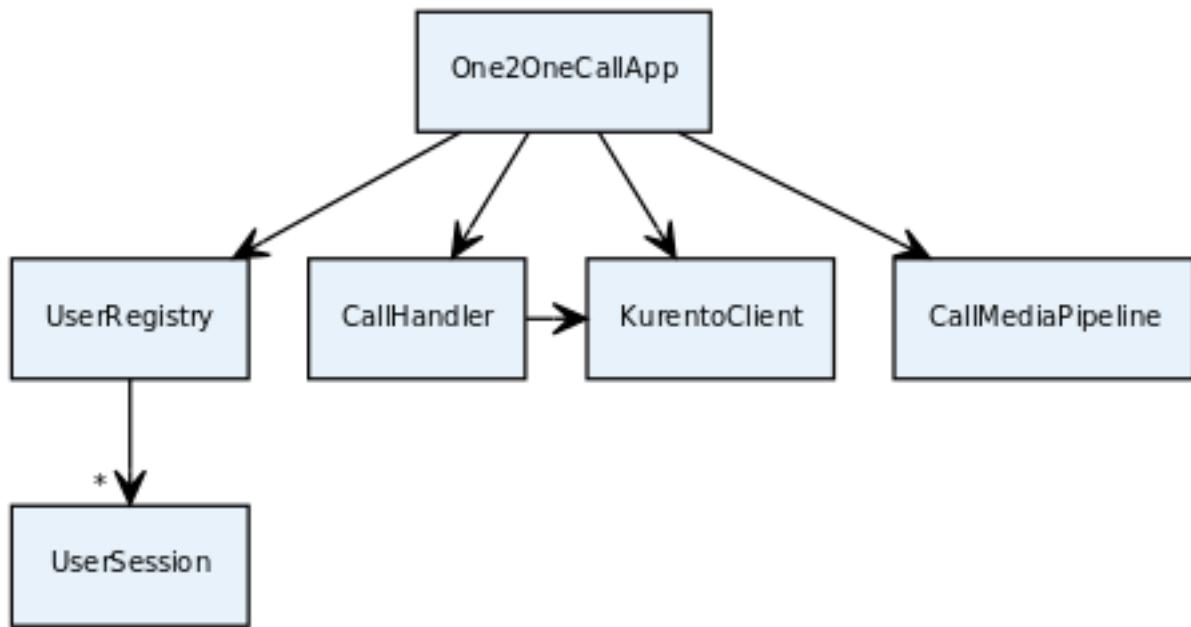


Fig. 6.31: Server-side class diagram of the one to one video call app

```

@Bean
public CallHandler callHandler() {
    return new CallHandler();
}

@Bean
public UserRegistry registry() {
    return new UserRegistry();
}

@Bean
public KurentoClient kurentoClient() {
    return KurentoClient.create();
}

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(callHandler(), "/call");
}

public static void main(String[] args) throws Exception {
    new SpringApplication(One2OneCallApp.class).run(args);
}
}

```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/call`.

`CallHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning

responses through the WebSocket. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are five different kind of incoming messages to the application server: register, call, incomingCallResponse, onIceCandidate and stop. These messages are treated in the *switch* clause, taking the proper steps in each case.

```
public class CallHandler extends TextWebSocketHandler {

    private static final Logger log = LoggerFactory.getLogger(CallHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, CallMediaPipeline> pipelines = new
    ↪ConcurrentHashMap<String, CallMediaPipeline>();

    @Autowired
    private KurentoClient kurento;

    @Autowired
    private UserRegistry registry;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) ↪
    throws Exception {
        JsonObject jsonMessage = gson.fromJson(message.getPayload(), JsonObject.class);
        UserSession user = registry.getBySession(session);

        if (user != null) {
            log.debug("Incoming message from user '{}': {}", user.getName(), ↪
        jsonMessage);
        } else {
            log.debug("Incoming message from new user: {}", jsonMessage);
        }

        switch (jsonMessage.get("id").getAsString()) {
            case "register":
                try {
                    register(session, jsonMessage);
                } catch (Throwable t) {
                    handleErrorResponse(t, session, "registerResponse");
                }
                break;
            case "call":
                try {
                    call(user, jsonMessage);
                } catch (Throwable t) {
                    handleErrorResponse(t, session, "callResponse");
                }
                break;
            case "incomingCallResponse":
                incomingCallResponse(user, jsonMessage);
                break;
            case "onIceCandidate":
                JsonObject candidate = jsonMessage.get("candidate").getAsJsonObject();
                if (user != null) {
                    IceCandidate cand = new IceCandidate(candidate.get("candidate") . ↪
                getAsString(),
                    candidate.get("sdpMid").getAsString(), candidate.get("sdpMLineIndex" ↪
                ).getAsInt());
                }
        }
    }
}
```

```

        user.addCandidate(cand);
    }
    break;
}
case "stop":
    stop(session);
    break;
default:
    break;
}

private void handleErrorResponse(Throwable t, WebSocketSession session,
    String responseId) throws IOException {
    stop(session);
    log.error(t.getMessage(), t);
    JSONObject response = new JSONObject();
    response.addProperty("id", responseId);
    response.addProperty("response", "rejected");
    response.addProperty("message", t.getMessage());
    session.sendMessage(new TextMessage(response.toString()));
}

private void register(WebSocketSession session, JSONObject jsonMessage) throws
IOException {
    ...
}

private void call(UserSession caller, JSONObject jsonMessage) throws IOException {
    ...
}

private void incomingCallResponse(final UserSession callee, JSONObject
jsonMessage) throws IOException {
    ...
}

public void stop(WebSocketSession session) throws IOException {
    ...
}

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status)
throws Exception {
    stop(session);
    registry.removeBySession(session);
}
}

```

In the following snippet, we can see the `register` method. Basically, it obtains the `name` attribute from `register` message and check if there are a registered user with that name. If not, the new user is registered and an acceptance message is sent to it.

```

private void register(WebSocketSession session, JSONObject jsonMessage) throws
IOException {
    String name = jsonMessage.getAsJsonPrimitive("name").getAsString();

```

```
UserSession caller = new UserSession(session, name);
String responseMsg = "accepted";
if (name.isEmpty()) {
    responseMsg = "rejected: empty user name";
} else if (registry.exists(name)) {
    responseMsg = "rejected: user '" + name + "' already registered";
} else {
    registry.register(caller);
}

JsonObject response = new JsonObject();
response.addProperty("id", "registerResponse");
response.addProperty("response", responseMsg);
caller.sendMessage(response);
}
```

In the `call` method, the server checks if there is a registered user with the name specified in `to` message attribute, and sends an `incomingCall` message. If there is no user with that name, a `callResponse` message is sent to caller rejecting the call.

```
private void call(UserSession caller, JsonObject jsonMessage) throws IOException {
    String to = jsonMessage.get("to").getAsString();
    String from = jsonMessage.get("from").getAsString();
    JsonObject response = new JsonObject();

    if (registry.exists(to)) {
        UserSession callee = registry.getByName(to);
        caller.setSdpOffer(jsonMessage.getAsJsonPrimitive("sdpOffer").getAsString());
        caller.setCallingTo(to);

        response.addProperty("id", "incomingCall");
        response.addProperty("from", from);

        callee.sendMessage(response);
        callee.setCallingFrom(from);
    } else {
        response.addProperty("id", "callResponse");
        response.addProperty("response", "rejected: user '" + to + "' is not registered");
    }

    caller.sendMessage(response);
}
```

The `stop` method ends the video call. It can be called both by caller and callee in the communication. The result is that both peers release the Media Pipeline and ends the video communication:

```
public void stop(WebSocketSession session) throws IOException {
    String sessionId = session.getId();
    if (pipelines.containsKey(sessionId)) {
        pipelines.get(sessionId).release();
        CallMediaPipeline pipeline = pipelines.remove(sessionId);
        pipeline.release();

        // Both users can stop the communication. A 'stopCommunication'
        // message will be sent to the other peer.
        UserSession stopperUser = registry.getBySession(session);
        if (stopperUser != null) {
```

```
UserSession stoppedUser = (stopperUser.getCallingFrom() != null)
    ? registry.getByName(stopperUser.getCallingFrom())
    : stopperUser.getCallingTo() != null
        ? registry.getByName(stopperUser.getCallingTo())
        : null;

if (stoppedUser != null) {
    JsonObject message = new JsonObject();
    message.addProperty("id", "stopCommunication");
    stoppedUser.sendMessage(message);
    stoppedUser.clear();
}

stopperUser.clear();
}

}
```

In the `incomingCallResponse` method, if the callee user accepts the call, it is established and the media elements are created to connect the caller with the callee in a B2B manner. Basically, the server creates a `CallMediaPipeline` object, to encapsulate the media pipeline creation and management. Then, this object is used to negotiate media interchange with user's browsers.

The negotiation between WebRTC peer in the browser and WebRtcEndpoint in Kurento Media Server is made by means of *SDP* generation at the client (offer) and SDP generation at the server (answer). The SDP answers are generated with the Kurento Java Client inside the class `CallMediaPipeline` (as we see in a moment). The methods used to generate SDP are `generateSdpAnswerForCallee(calleeSdpOffer)` and `generateSdpAnswerForCaller(callerSdpOffer)`:

```
private void incomingCallResponse(final UserSession callee, JSONObject jsonMessage) {
    throws IOException {
    String callResponse = jsonMessage.get("callResponse").getAsString();
    String from = jsonMessage.get("from").getAsString();
    final UserSession calleer = registry.getByName(from);
    String to = calleer.getCallingTo();

    if ("accept".equals(callResponse)) {
        log.debug("Accepted call from '{}' to '{}'", from, to);

        CallMediaPipeline pipeline = null;
        try {
            pipeline = new CallMediaPipeline(kurento);
            pipelines.put(calleer.getSessionId(), pipeline);
            pipelines.put(callee.getSessionId(), pipeline);

            String calleeSdpOffer = jsonMessage.get("sdpOffer").getAsString();
            callee.setWebRtcEndpoint(pipeline.getcalleeWebRtcEP());
            pipeline.getcalleeWebRtcEP().addIceCandidateFoundListener(new EventListener
    <IceCandidateFoundEvent>() {
        @Override
        public void onEvent(IceCandidateFoundEvent event) {
            JSONObject response = new JSONObject();
            response.addProperty("id", "iceCandidate");
            response.add("candidate", JsonUtils.toJsonObject(event.
    getcandidate()));
            try {
                synchronized (callee.getSession()) {
                    callee.getSession().sendMessage(new TextMessage(response.
    toString()));
                }
            } catch (IOException e) {
                log.error("Error sending ice candidate message to callee: " + e.getMessage());
            }
        }
    });
    }
}
```

```

        }

    } catch (IOException e) {
        log.debug(e.getMessage());
    }
}

});

String calleeSdpAnswer = pipeline.generateSdpAnswerForCallee(calleeSdpOffer);
String callerSdpOffer = registry.getByName(from).getSdpOffer();
calleeer.setWebRtcEndpoint(pipeline.getCallerWebRtcEP());
pipeline.getCallerWebRtcEP().addIceCandidateFoundListener(new EventListener
→<IceCandidateFoundEvent>() {

    @Override
    public void onEvent(IceCandidateFoundEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils.toJsonObject(event.
→getCandidate()));
        try {
            synchronized (calleeer.getSession()) {
                calleeer.getSession().sendMessage(new TextMessage(response.
→toString()));
            }
        } catch (IOException e) {
            log.debug(e.getMessage());
        }
    }
});

String callerSdpAnswer = pipeline.generateSdpAnswerForCaller(callerSdpOffer);

JsonObject startCommunication = new JsonObject();
startCommunication.addProperty("id", "startCommunication");
startCommunication.addProperty("sdpAnswer", calleeSdpAnswer);

synchronized (callee) {
    callee.sendMessage(startCommunication);
}

pipeline.getcalleeWebRtcEP().gatherCandidates();

JsonObject response = new JsonObject();
response.addProperty("id", "callResponse");
response.addProperty("response", "accepted");
response.addProperty("sdpAnswer", callerSdpAnswer);

synchronized (calleeer) {
    calleer.sendMessage(response);
}

pipeline.getcallerWebRtcEP().gatherCandidates();

} catch (Throwable t) {
    log.error(t.getMessage(), t);

    if (pipeline != null) {
        pipeline.release();
    }
}
}

```

```
        }

        pipelines.remove(calleer.getSessionId());
        pipelines.remove(callee.getSessionId());

        JsonObject response = new JsonObject();
        response.addProperty("id", "callResponse");
        response.addProperty("response", "rejected");
        calleer.sendMessage(response);

        response = new JsonObject();
        response.addProperty("id", "stopCommunication");
        callee.sendMessage(response);
    }

} else {
    JsonObject response = new JsonObject();
    response.addProperty("id", "callResponse");
    response.addProperty("response", "rejected");
    calleer.sendMessage(response);
}

}
```

The media logic in this demo is implemented in the class `CallMediaPipeline`. As you can see, the media pipeline of this demo is quite simple: two `WebRtcEndpoint` elements directly interconnected. Please take note that the `WebRtcEndpoints` need to be connected twice, one for each media direction.

```
public class CallMediaPipeline {  
  
    private MediaPipeline pipeline;  
    private WebRtcEndpoint callerWebRtcEP;  
    private WebRtcEndpoint calleeWebRtcEP;  
  
    public CallMediaPipeline(KurentoClient kurento) {  
        try {  
            this.pipeline = kurento.createMediaPipeline();  
            this.callerWebRtcEP = new WebRtcEndpoint.Builder(pipeline).build();  
            this.calleeWebRtcEP = new WebRtcEndpoint.Builder(pipeline).build();  
  
            this.callerWebRtcEP.connect(this.calleeWebRtcEP);  
            this.calleeWebRtcEP.connect(this.callerWebRtcEP);  
        } catch (Throwable t) {  
            if (this.pipeline != null) {  
                pipeline.release();  
            }  
        }  
    }  
  
    public String generateSdpAnswerForCaller(String sdpOffer) {  
        return callerWebRtcEP.processOffer(sdpOffer);  
    }  
  
    public String generateSdpAnswerForCallee(String sdpOffer) {  
        return calleeWebRtcEP.processOffer(sdpOffer);  
    }  
  
    public void release() {  
        if (pipeline != null) {  
            pipeline.release();  
        }  
    }  
}
```

```
        pipeline.release();
    }
}

public WebRtcEndpoint getCallerWebRtcEP() {
    return callerWebRtcEP;
}

public WebRtcEndpoint getCalleeWebRtcEP() {
    return calleeWebRtcEP;
}

}
```

In this class we can see the implementation of methods `generateSdpAnswerForCaller` and `generateSdpAnswerForCallee`. These methods delegate to WebRtc endpoints to create the appropriate answer.

Client-Side

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`.

In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/call`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are five incoming messages to client: `registerResponse`, `callResponse`, `incomingCall`, `iceCandidate` and `startCommunication`. Convenient actions are taken to implement each step in the communication. For example, in functions `call` and `incomingCall` (for caller and callee respectively), the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```
var ws = new WebSocket('ws://' + location.host + '/call');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'registerResponse':
            registerResponse(parsedMessage);
            break;
        case 'callResponse':
            callResponse(parsedMessage);
            break;
        case 'incomingCall':
            incomingCall(parsedMessage);
            break;
        case 'startCommunication':
            startCommunication(parsedMessage);
            break;
        case 'stopCommunication':
            console.info("Communication ended by remote peer");
            stop(true);
    }
}
```

```

        break;
    case 'iceCandidate':
        webRtcPeer.addIceCandidate(parsedMessage.candidate, function (error) {
            if (!error) return;
            console.error("Error adding candidate: " + error);
        });
        break;
    default:
        console.error('Unrecognized message', parsedMessage);
    }
}

function incomingCall(message) {
    //If bussy just reject without disturbing user
    if (callState != NO_CALL) {
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'reject',
            message : 'bussy'
        };
        return sendMessage(response);
    }

    setCallState(PREPROCESSING_CALL);
    if (confirm('User ' + message.from
        + ' is calling you. Do you accept the call?')) {
        showSpinner(videoInput, videoOutput);

        from = message.from;
        var options = {
            localVideo: videoInput,
            remoteVideo: videoOutput,
            onicecandidate: onIceCandidate,
            onerror: onError
        }
        webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
            function (error) {
                if(error) {
                    return console.error(error);
                }
                webRtcPeer.generateOffer (onOfferIncomingCall);
            });
    } else {
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'reject',
            message : 'user declined'
        };
        sendMessage(response);
        stop();
    }
}

function call() {
    if (document.getElementById('peer').value == '') {

```

```

        window.alert("You must specify the peer name");
        return;
    }
    setCallState(PROCESSING_CALL);
    showSpinner(videoInput, videoOutput);

    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput,
        onicecandidate: onIceCandidate,
        onerror: onError
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function (error) {
            if(error) {
                return console.error(error);
            }
            webRtcPeer.generateOffer (onOfferCall);
        });
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side. Other client libraries are managed with webjars:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>draggabilly</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>

```

```

</dependency>
<dependency>
  <groupId>org.webjars.bower</groupId>
  <artifactId>jquery</artifactId>
</dependency>
<dependency>
  <groupId>org.webjars.bower</groupId>
  <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```

<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>

```

6.5.2 Node.js - One to one video call

This web application consists on a one-to-one video call using *WebRTC* technology. In other words, this application provides a simple video softphone.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information.

Be sure to have installed *Node.js* and *Bower* in your system. In an Ubuntu machine, you can install both as follows:

```

curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower

```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```

git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-one2one-call
git checkout 6.7.1
npm install
npm start

```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```

rm -r node_modules
npm cache clean

```

Access the application connecting to the URL <https://localhost:8443> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

The following picture shows an screenshot of this demo running in a web browser:

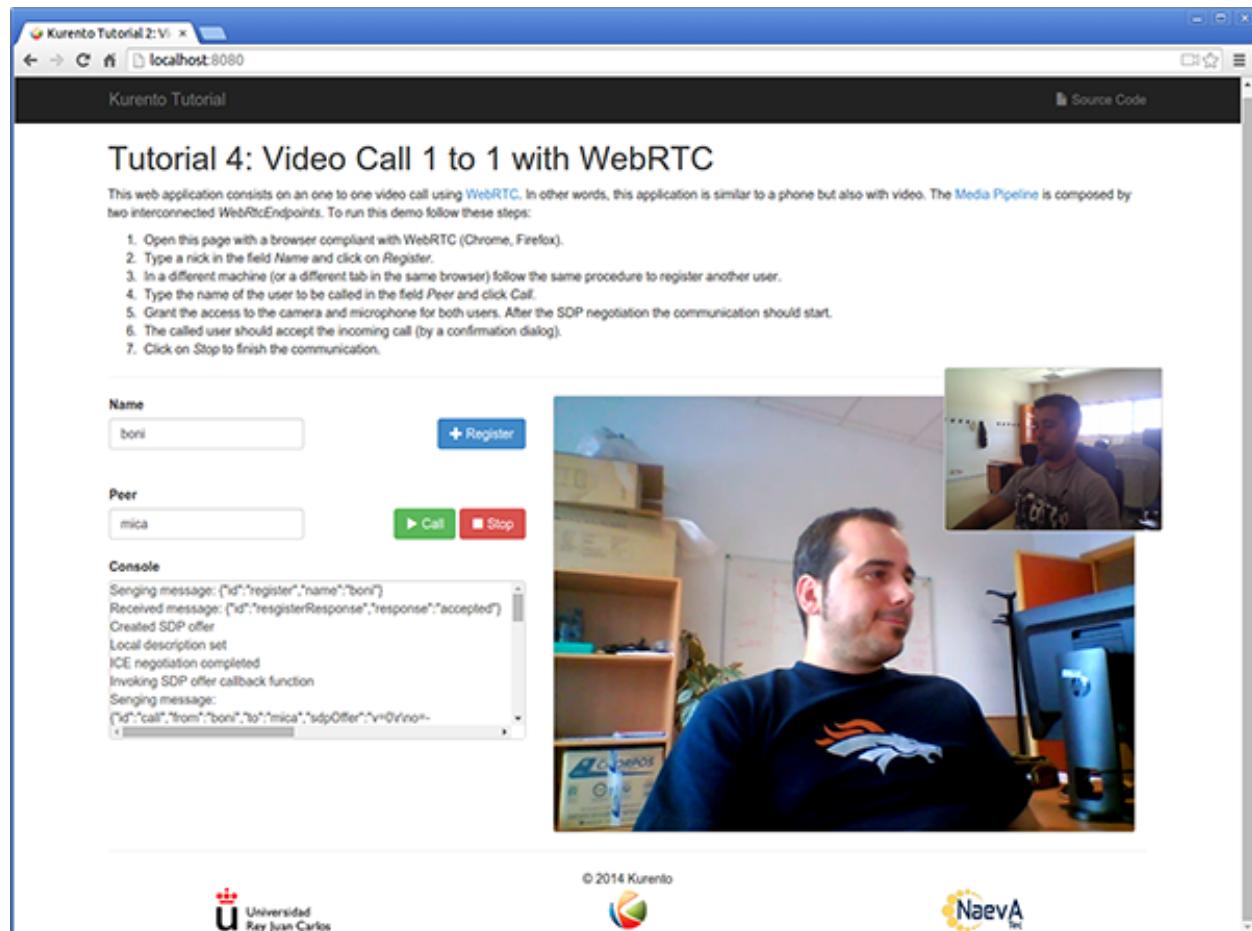


Fig. 6.32: One to one video call screenshot

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the local stream and other for the remote peer stream). If two users, A and B, are using the application, the media flow goes this way: The video camera stream of user A is sent to the Kurento Media Server, which sends it to user B. In the same way, B

sends to Kurento Media Server, which forwards it to A. This means that KMS is providing a B2B (back-to-back) call service.

To implement this behavior create a *Media Pipeline* composed by two WebRtc endpoints connected in B2B. The implemented media pipeline is illustrated in the following picture:

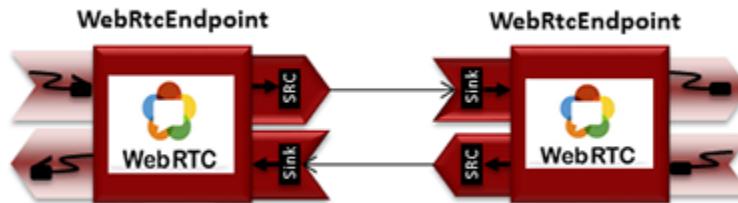


Fig. 6.33: One to one video call Media Pipeline

The client and the server communicate through a signaling protocol based on *JSON* messages over *WebSocket*'s. The normal sequence between client and application server logic is as follows:

1. User A is registered in the application server with his name
2. User B is registered in the application server with her name
3. User A issues a call to User B
4. User B accepts the incoming call
5. The communication is established and media flows between User A and User B
6. One of the users finishes the video communication

The detailed message flow in a call are shown in the picture below:

As you can see in the diagram, *SDP* and *ICE* candidates need to be exchanged between client and server to establish the *WebRTC* connection between the Kurento client and server. Specifically, the SDP negotiation connects the *WebRtcPeer* in the browser with the *WebRtcEndpoint* in the server. The complete source code of this demo can be found in [GitHub](#).

Application Server Logic

This demo has been developed using the **express** framework for Node.js, but express is not a requirement for Kurento. The main script of this demo is `server.js`.

In order to communicate the JavaScript client and the Node application server a *WebSocket* is used. The incoming messages to this *WebSocket* (variable `ws` in the code) are conveniently handled to implemented the signaling protocol depicted in the figure before (i.e. messages `register`, `call`, `incomingCallResponse`, `stop`, and `onIceCandidate`).

```
var ws = require('ws');

[...]

var wss = new ws.Server({
  server : server,
  path : '/one2one'
});

wss.on('connection', function(ws) {
  var sessionId = nextUniqueId();
```

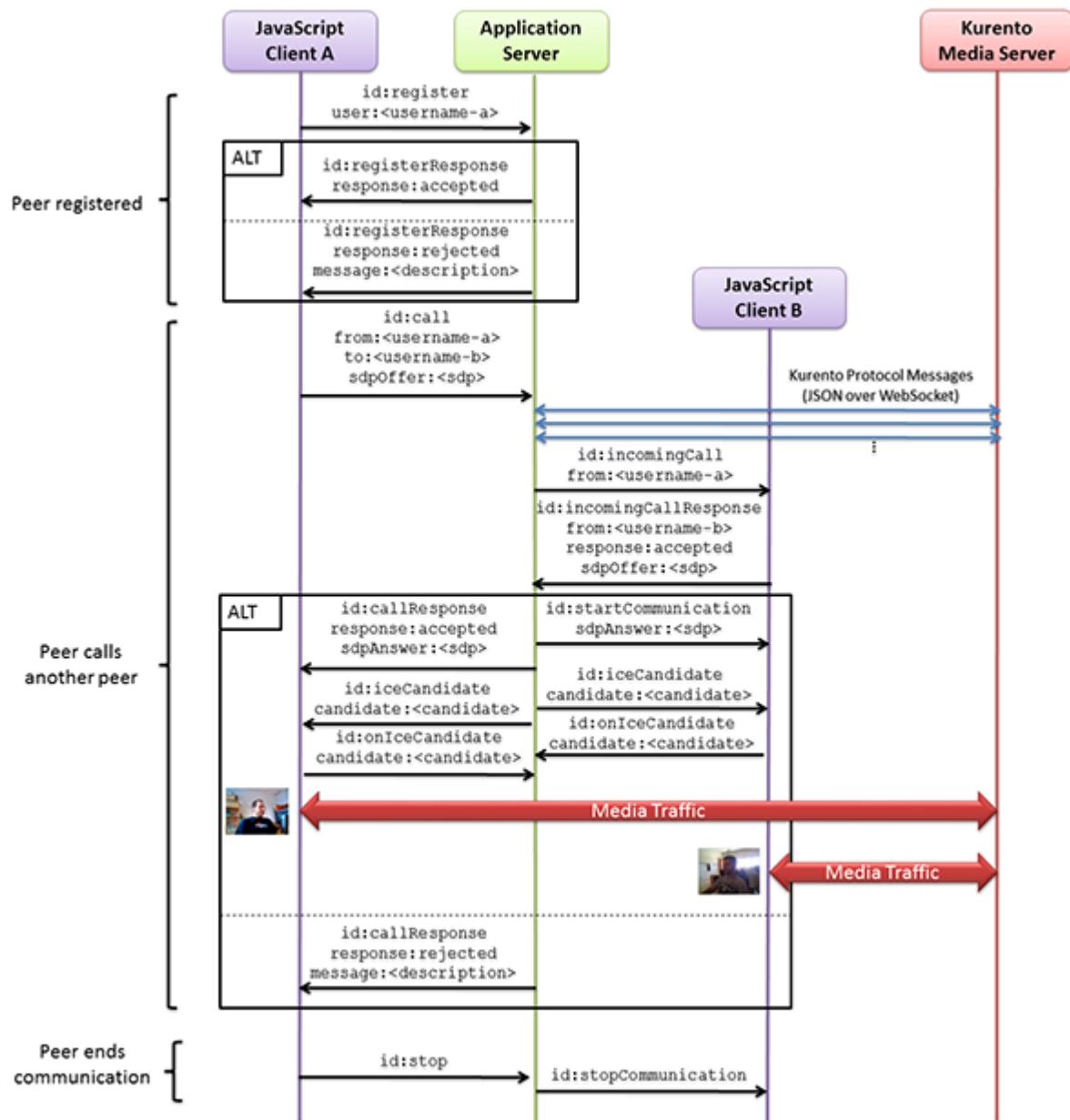


Fig. 6.34: One to many one call signaling protocol

```

console.log('Connection received with sessionId ' + sessionId);

ws.on('error', function(error) {
    console.log('Connection ' + sessionId + ' error');
    stop(sessionId);
});

ws.on('close', function() {
    console.log('Connection ' + sessionId + ' closed');
    stop(sessionId);
    userRegistry.unregister(sessionId);
});

ws.on('message', function(_message) {
    var message = JSON.parse(_message);
    console.log('Connection ' + sessionId + ' received message ', message);

    switch (message.id) {
        case 'register':
            register(sessionId, message.name, ws);
            break;

        case 'call':
            call(sessionId, message.to, message.from, message.sdpOffer);
            break;

        case 'incomingCallResponse':
            incomingCallResponse(sessionId, message.from, message.callResponse,
            ↵message.sdpOffer, ws);
            break;

        case 'stop':
            stop(sessionId);
            break;

        case 'onIceCandidate':
            onIceCandidate(sessionId, message.candidate);
            break;

        default:
            ws.send(JSON.stringify({
                id : 'error',
                message : 'Invalid message ' + message
            }));
            break;
    }
});

});
});

```

In order to perform a call, each user (the caller and the callee) must be register in the system. For this reason, in the server-side there is a class named `UserRegistry` to store and locate users. Then, the `register` message fires the execution of the following function:

```

// Represents registrar of users
function UserRegistry() {
    this.usersById = {};
    this.usersByName = {};

```

```

}

UserRegistry.prototype.register = function(user) {
    this.usersById[user.id] = user;
    this.usersByName[user.name] = user;
}

UserRegistry.prototype.unregister = function(id) {
    var user = this.getById(id);
    if (user) delete this.usersById[id];
    if (user && this.getByName(user.name)) delete this.usersByName[user.name];
}

UserRegistry.prototype.getById = function(id) {
    return this.usersById[id];
}

UserRegistry.prototype.getByName = function(name) {
    return this.usersByName[name];
}

UserRegistry.prototype.removeById = function(id) {
    var userSession = this.usersById[id];
    if (!userSession) return;
    delete this.usersById[id];
    delete this.usersByName[userSession.name];
}

function register(id, name, ws, callback) {
    function onError(error) {
        ws.send(JSON.stringify({id:'registerResponse', response : 'rejected ',  
message: error}));
    }

    if (!name) {
        return onError("empty user name");
    }

    if (userRegistry.getByName(name)) {
        return onError("User " + name + " is already registered");
    }

    userRegistry.register(new UserSession(id, name, ws));
    try {
        ws.send(JSON.stringify({id: 'registerResponse', response: 'accepted'}));
    } catch(exception) {
        onError(exception);
    }
}

```

In order to control the media capabilities provided by the Kurento Media Server, we need an instance of the *KurentoClient* in the Node application server. In order to create this instance, we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it's located at *localhost* listening on port 8888.

```

var kurento = require('kurento-client');

var kurentoClient = null;

```

```

var argv = minimist(process.argv.slice(2), {
  default: {
    as_uri: 'https://localhost:8443/',
    ws_uri: 'ws://localhost:8888/kurento'
  }
});

[...]

function getKurentoClient(callback) {
  if (kurentoClient !== null) {
    return callback(null, kurentoClient);
  }

  kurento(argv.ws_uri, function(error, _kurentoClient) {
    if (error) {
      console.log("Could not find media server at address " + argv.ws_uri);
      return callback("Could not find media server at address" + argv.ws_uri
        + ". Exiting with error " + error);
    }

    kurentoClient = _kurentoClient;
    callback(null, kurentoClient);
  });
}

```

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server. Our first operation is to create a *Media Pipeline*, then we need to create the *Media Elements* and connect them. In this example, we need two WebRtcEndpoints, i.e. one peer caller and other one for the callee. This media logic is implemented in the class CallMediaPipeline. Note that the WebRtcEndpoints need to be connected twice, one for each media direction. This object is created in the function incomingCallResponse which is fired in the callee peer, after the caller executes the function call:

```

function call(callerId, to, from, sdpOffer) {
  clearCandidatesQueue(callerId);

  var caller = userRegistry.getById(callerId);
  var rejectCause = 'User ' + to + ' is not registered';
  if (userRegistry.getByName(to)) {
    var callee = userRegistry.getByName(to);
    caller.sdpOffer = sdpOffer
    callee.peer = from;
    caller.peer = to;
    var message = {
      id: 'incomingCall',
      from: from
    };
    try{
      return callee.sendMessage(message);
    } catch(exception) {
      rejectCause = "Error " + exception;
    }
  }
  var message = {
    id: 'callResponse',
    response: 'rejected: ',
    message: rejectCause
  }
}

```

```

};

caller.sendMessage(message);
}

function incomingCallResponse(calleeId, from, callResponse, calleeSdp, ws) {
clearCandidatesQueue(calleeId);

function onError(callerReason, calleeReason) {
    if (pipeline) pipeline.release();
    if (caller) {
        var callerMessage = {
            id: 'callResponse',
            response: 'rejected'
        }
        if (callerReason) callerMessage.message = callerReason;
        caller.sendMessage(callerMessage);
    }

    var calleeMessage = {
        id: 'stopCommunication'
    };
    if (calleeReason) calleeMessage.message = calleeReason;
    callee.sendMessage(calleeMessage);
}

var callee = userRegistry.getById(calleeId);
if (!from || !userRegistry.getByName(from)) {
    return onError(null, 'unknown from = ' + from);
}
var caller = userRegistry.getByName(from);

if (callResponse === 'accept') {
    var pipeline = new CallMediaPipeline();
    pipelines[caller.id] = pipeline;
    pipelines[callee.id] = pipeline;

    pipeline.createPipeline(caller.id, callee.id, ws, function(error) {
        if (error) {
            return onError(error, error);
        }

        pipeline.generateSdpAnswer(caller.id, caller.sdpOffer, function(error, ↵
callerSdpAnswer) {
            if (error) {
                return onError(error, error);
            }

            pipeline.generateSdpAnswer(callee.id, calleeSdp, function(error, ↵
calleeSdpAnswer) {
                if (error) {
                    return onError(error, error);
                }

                var message = {
                    id: 'startCommunication',
                    sdpAnswer: calleeSdpAnswer
                };
                callee.sendMessage(message);
            }
        })
    })
}
}

```

```

        message = {
            id: 'callResponse',
            response : 'accepted',
            sdpAnswer: callerSdpAnswer
        };
        caller.sendMessage(message);
    });
});
} else {
    var decline = {
        id: 'callResponse',
        response: 'rejected',
        message: 'user declined'
    };
    caller.sendMessage(decline);
}
}
}

```

As of Kurento Media Server 6.0, the WebRTC negotiation is done by exchanging *ICE* candidates between the WebRTC peers. To implement this protocol, the `webRtcEndpoint` receives candidates from the client in `OnIceCandidate` function. These candidates are stored in a queue when the `webRtcEndpoint` is not available yet. Then these candidates are added to the media element by calling to the `addIceCandidate` method.

```

var candidatesQueue = {};

[...]

function onIceCandidate(sessionId, _candidate) {
    var candidate = kurento.getComplexType('IceCandidate')(_candidate);
    var user = userRegistry.getById(sessionId);

    if (pipelines[user.id] && pipelines[user.id].webRtcEndpoint && pipelines[user.id].  
→webRtcEndpoint[user.id]) {
        var webRtcEndpoint = pipelines[user.id].webRtcEndpoint[user.id];
        webRtcEndpoint.addIceCandidate(candidate);
    }
    else {
        if (!candidatesQueue[user.id]) {
            candidatesQueue[user.id] = [];
        }
        candidatesQueue[sessionId].push(candidate);
    }
}

function clearCandidatesQueue(sessionId) {
    if (candidatesQueue[sessionId]) {
        delete candidatesQueue[sessionId];
    }
}

```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js`

to simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application. These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/oneZone`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer`.
`WebRtcPeerSendrecv` of *kurento-utils.js* is used to start a WebRTC communication.

```
var ws = new WebSocket('ws://' + location.host + '/oneZone');
var webRtcPeer;

[...]

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'registerResponse':
            registerResponse(parsedMessage);
            break;
        case 'callResponse':
            callResponse(parsedMessage);
            break;
        case 'incomingCall':
            incomingCall(parsedMessage);
            break;
        case 'startCommunication':
            startCommunication(parsedMessage);
            break;
        case 'stopCommunication':
            console.info("Communication ended by remote peer");
            stop(true);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate)
            break;
        default:
            console.error('Unrecognized message', parsedMessage);
    }
}
```

On the one hand, the function `call` is executed in the caller client-side, using the method `WebRtcPeer`.
`WebRtcPeerSendrecv` of *kurento-utils.js* to start a WebRTC communication in duplex mode. On the other hand, the function `incomingCall` in the callee client-side uses also the method `WebRtcPeer`.
`WebRtcPeerSendrecv` of *kurento-utils.js* to complete the WebRTC call.

```
function call() {
    if (document.getElementById('peer').value == '') {
        window.alert("You must specify the peer name");
        return;
    }

    setCallState(PROCESSING_CALL);

    showSpinner(videoInput, videoOutput);
```

```

var options = {
    localVideo : videoInput,
    remoteVideo : videoOutput,
    onicecandidate : onIceCandidate
}

webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(
    error) {
    if (error) {
        console.error(error);
        setCallState(NO_CALL);
    }

    this.generateOffer(function(error, offerSdp) {
        if (error) {
            console.error(error);
            setCallState(NO_CALL);
        }
        var message = {
            id : 'call',
            from : document.getElementById('name').value,
            to : document.getElementById('peer').value,
            sdpOffer : offerSdp
        };
        sendMessage(message);
    });
});
});

function incomingCall(message) {
    // If bussy just reject without disturbing user
    if (callState != NO_CALL) {
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'reject',
            message : 'bussy'

        };
        return sendMessage(response);
    }

    setCallState(PREPROCESSING_CALL);
    if (confirm('User ' + message.from
        + ' is calling you. Do you accept the call?')) {
        showSpinner(videoInput, videoOutput);

        var options = {
            localVideo : videoInput,
            remoteVideo : videoOutput,
            onicecandidate : onIceCandidate
        }

        webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
            function(error) {
                if (error) {
                    console.error(error);
                    setCallState(NO_CALL);
                }
            })
    }
}

```

```
}

    this.generateOffer(function(error, offerSdp) {
        if (error) {
            console.error(error);
            setCallState(NO_CALL);
        }
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'accept',
            sdpOffer : offerSdp
        };
        sendMessage(response);
    });
});

} else {
    var response = {
        id : 'incomingCallResponse',
        from : message.from,
        callResponse : 'reject',
        message : 'user declined'
    };
    sendMessage(response);
    stop(true);
}
}
```

Dependencies

Server-side dependencies of this demo are managed using [npm](#). Our main dependency is the Kurento Client JavaScript (*kurento-client*). The relevant part of the `package.json` file for managing this dependency is:

```
"dependencies": {
    [...]
    "kurento-client" : "6.7.1"
}
```

At the client side, dependencies are managed using [Bower](#). Take a look to the `bower.json` file and pay attention to the following section:

```
"dependencies": {
    [...]
    "kurento-utils" : "6.7.1"
}
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [npm](#) and [Bower](#).

6.6 WebRTC One-To-One video call with recording and filtering

This is an enhanced version of the the One-To-One application with video recording and Augmented Reality.

6.6.1 Java - Advanced One to one video call

This web application consists on an advanced one to one video call using *WebRTC* technology. It is an improved version of the [one 2 one call tutorial](#).

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-one2one-call-advanced
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This application incorporates the recording capability provided by the Kurento Media Server in a one to one video communication. In addition, a filter element (*FaceOverlayFilter*) is placed between the *WebRtcEndpoints* of the Media Pipeline. The following picture shows an screenshot of this demo running in a web browser:

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the local video camera stream (the caller stream, the smaller video in the picture) and other for the remote peer in the call (the callee stream, the bigger video in the picture). If two users, A and B, are using the application, the media flow goes this way: The video camera stream of user A is sent to the Kurento Media Server and sent again to the user B. On the other hand, user B sends its video camera stream to Kurento and then it is sent to user A.

This application is implemented by means of two *Media Pipeline*'s. First, the rich real-time WebRTC communication is performed two *WebRtcEndpoints* interconnected, and with a *FaceOverlayFilter* in between them. In addition and a *RecorderEndpoint* is used to store both streams in the file system of the Kurento Media Server. This media pipeline is illustrated in the following picture:

A second media pipeline is needed to play the previously recorded media. This pipeline is composed by a *PlayerEndpoint* which reads the files stored in the Kurento Media Server. This media element injects the media in a *WebRtcEndpoint* which is charge to transport the media to the HTML5 video tag in the browser:

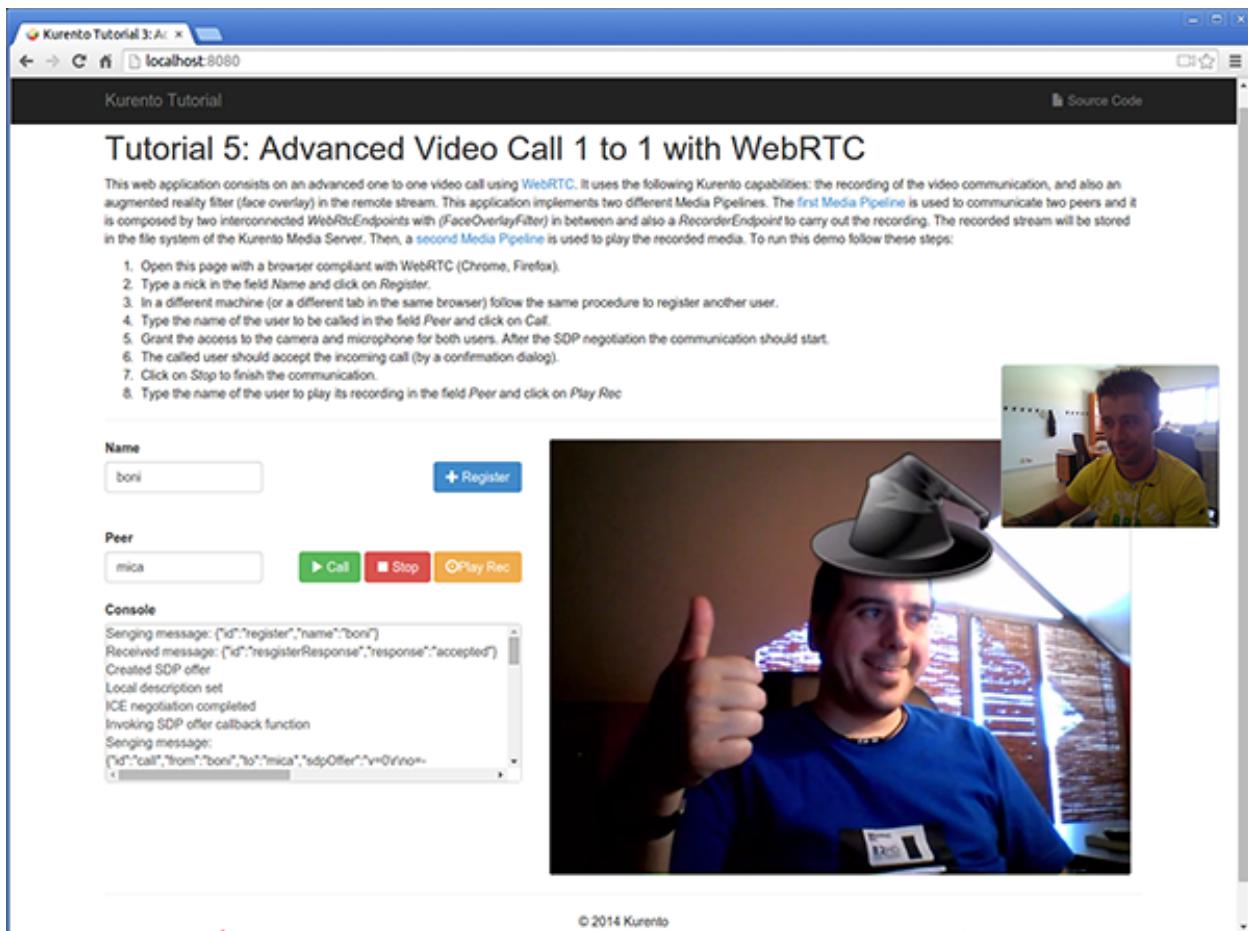


Fig. 6.35: Advanced one to one video call screenshot

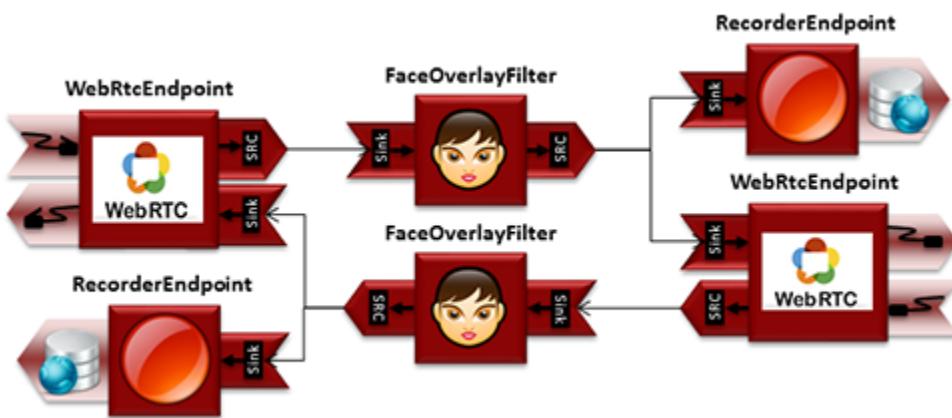


Fig. 6.36: Advanced one to one video call media pipeline (1)

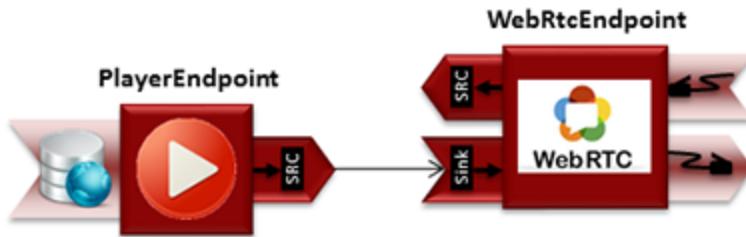


Fig. 6.37: Advanced one to one video call media pipeline (2)

Note: The playback of a static file can be done in several ways. In addition to this media pipeline (*PlayerEndpoint* -> *WebRtcEndpoint*) the recorded file could be served directly by an HTTP server.

To communicate the client with the server to manage calls we have designed a signaling protocol based on *JSON* messages over *WebSocket*'s. The normal sequence between client and server would be as follows:

1. User A is registered in the server with his name
2. User B is registered in the server with her name
3. User A wants to call to User B
4. User B accepts the incoming call
5. The communication is established and media is flowing between User A and User B
6. One of the users finishes the video communication
7. One of the users play the recorded media

This is very simple protocol designed to show a simple one to one call application implemented with Kurento. In a professional application it can be improved, for example implementing seeking user, ordered finish, among other functions.

Assuming that User A is using Client A and User B is using Client B, we can draw the following sequence diagram with detailed messages between clients and server. The following diagram shows the two parts of the signaling protocol: first the enhanced real-time communication is performed, and then the playback of the recorded file is carried out.

As you can see in the diagram, *SDP* and *ICE* candidates need to be interchanged between client and server to establish the *WebRTC* connection between the Kurento client and server. Specifically, the SDP negotiation connects the *WebRtcPeer* in the browser with the *WebRtcEndpoint* in the server.

The following sections describe in detail the server-side, the client-side, and how to run the demo. The complete source code of this demo can be found in [GitHub](#).

Application Server Logic

As in the *Magic Mirror tutorial*, this demo has been developed using **Java** and *Spring Boot*.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vertex, etc. We have choose Spring Boot for convenience.

In the following figure you can see a class diagram of the server side code:

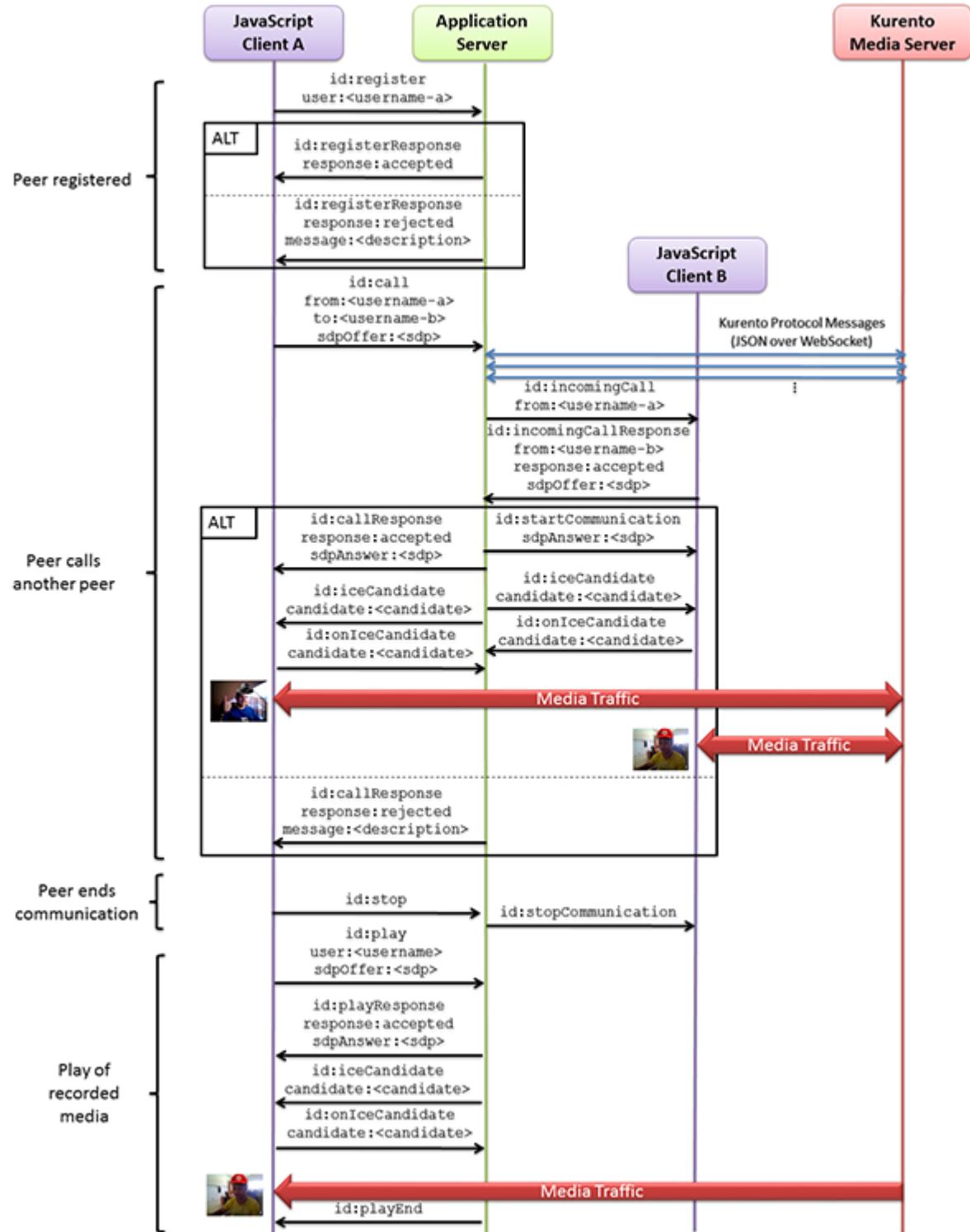


Fig. 6.38: Advanced one to one video call signaling protocol

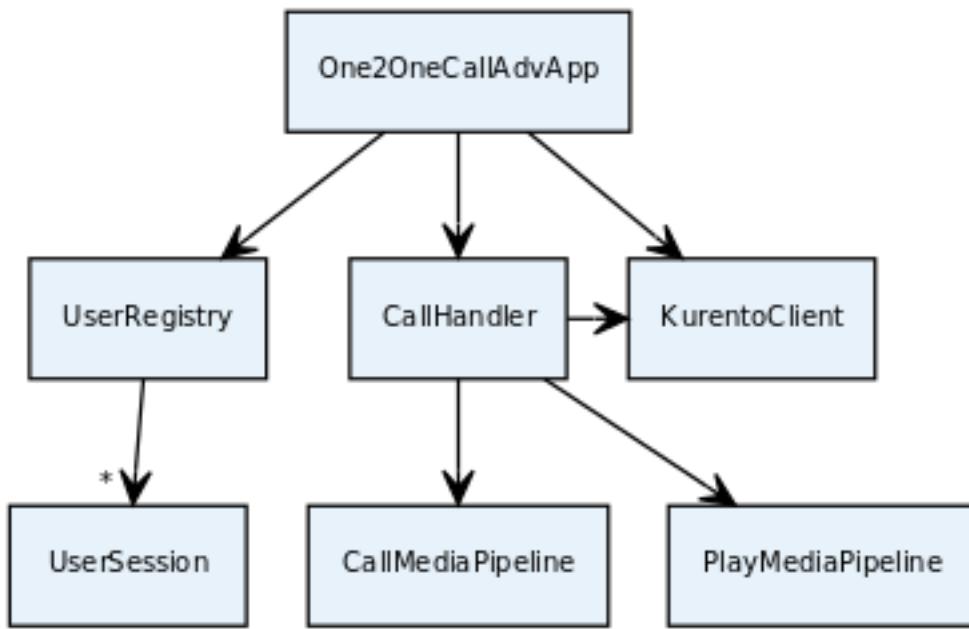


Fig. 6.39: Server-side class diagram of the advanced one to one video call app

The main class of this demo is named `One2OneCallAdvApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean.

```

@EnableWebSocket
@SpringBootApplication
public class One2OneCallAdvApp implements WebSocketConfigurer {

    final static String DEFAULT_APP_SERVER_URL = "https://localhost:8443";

    @Bean
    public CallHandler callHandler() {
        return new CallHandler();
    }

    @Bean
    public UserRegistry registry() {
        return new UserRegistry();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(callHandler(), "/call");
    }

    public static void main(String[] args) throws Exception {
        new SpringApplication(One2OneCallAdvApp.class).run(args);
    }
}
  
```

```
}
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/call`.

`CallHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are five different kind of incoming messages to the *Server* : `register`, `call`, `incomingCallResponse`, `onIceCandidate` and `play`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```
public class CallHandler extends TextWebSocketHandler {

    private static final Logger log = LoggerFactory
        .getLogger(CallHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, MediaPipeline> pipelines = new
    ↵ConcurrentHashMap<String, MediaPipeline>();

    @Autowired
    private KurentoClient kurento;

    @Autowired
    private UserRegistry registry;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message)
        throws Exception {
        JsonObject jsonMessage = gson.fromJson(message.getPayload(),
            JsonObject.class);
        UserSession user = registry.getBySession(session);

        if (user != null) {
            log.debug("Incoming message from user '{}': {}", user.getName(),
                jsonMessage);
        } else {
            log.debug("Incoming message from new user: {}", jsonMessage);
        }

        switch (jsonMessage.get("id").getAsString()) {
            case "register":
                register(session, jsonMessage);
                break;
            case "call":
                call(user, jsonMessage);
                break;
            case "incomingCallResponse":
                incomingCallResponse(user, jsonMessage);
                break;
            case "play":
                play(user, jsonMessage);
                break;
            case "onIceCandidate": {
                onIceCandidate(user, jsonMessage);
            }
        }
    }
}
```

```

JsonObject candidate = jsonMessage.get("candidate")
    .getAsJsonObject();

    if (user != null) {
        IceCandidate cand = new IceCandidate(candidate.get("candidate")
            .getAsString(), candidate.get("sdpMid").getAsString(),
            candidate.get("sdpMLineIndex").getAsInt());
        user.addCandidate(cand);
    }
    break;
}
case "stop":
    stop(session);
    releasePipeline(user);
case "stopPlay":
    releasePipeline(user);
default:
    break;
}
}

private void register(WebSocketSession session, JsonObject jsonMessage)
    throws IOException {
    ...
}

private void call(UserSession caller, JsonObject jsonMessage)
    throws IOException {
    ...
}

private void incomingCallResponse(final UserSession callee,
    JsonObject jsonMessage) throws IOException {
    ...
}

public void stop(WebSocketSession session) throws IOException {
    ...
}

public void releasePipeline(UserSession session) throws IOException {
    ...
}

private void play(final UserSession session, JsonObject jsonMessage)
    throws IOException {
    ...
}

@Override
public void afterConnectionClosed(WebSocketSession session,
    CloseStatus status) throws Exception {
    stop(session);
    registry.removeBySession(session);
}
}

```

In the following snippet, we can see the `register` method. Basically, it obtains the `name` attribute from `register` message and check if there are a registered user with that name. If not, the new user is registered and an acceptance message is sent to it.

```
private void register(WebSocketSession session, JsonObject jsonMessage)
    throws IOException {
String name = jsonMessage.getAsJsonPrimitive("name").getAsString();

UserSession caller = new UserSession(session, name);
String responseMsg = "accepted";
if (name.isEmpty()) {
    responseMsg = "rejected: empty user name";
} else if (registry.exists(name)) {
    responseMsg = "rejected: user '" + name + "' already registered";
} else {
    registry.register(caller);
}

JsonObject response = new JsonObject();
response.addProperty("id", "registerResponse");
response.addProperty("response", responseMsg);
caller.sendMessage(response);
}
```

In the `call` method, the server checks if there are a registered user with the name specified in `to` message attribute and send an `incomingCall` message to it. Or, if there isn't any user with that name, a `callResponse` message is sent to caller rejecting the call.

```
private void call(UserSession caller, JsonObject jsonMessage)
    throws IOException {
String to = jsonMessage.get("to").getAsString();
String from = jsonMessage.get("from").getAsString();
JsonObject response = new JsonObject();

if (registry.exists(to)) {
    UserSession callee = registry.getByName(to);
    caller.setSdpOffer(jsonMessage.getAsJsonPrimitive("sdpOffer")
        .getAsString());
    caller.setCallingTo(to);

    response.addProperty("id", "incomingCall");
    response.addProperty("from", from);

    callee.sendMessage(response);
    callee.setCallingFrom(from);
} else {
    response.addProperty("id", "callResponse");
    response.addProperty("response", "rejected");
    response.addProperty("message", "user '" + to
        + "' is not registered");

    caller.sendMessage(response);
}
}
```

In the `incomingCallResponse` method, if the callee user accepts the call, it is established and the media elements are created to connect the caller with the callee. Basically, the server creates a `CallMediaPipeline` object, to encapsulate the media pipeline creation and management. Then, this object is used to negotiate media interchange

with user's browsers.

As explained in the [Magic Mirror tutorial](#), the negotiation between WebRTC peer in the browser and WebRtcEndpoint in Kurento Server is made by means of [SDP](#) generation at the client (offer) and SDP generation at the server (answer). The SDP answers are generated with the Kurento Java Client inside the class CallMediaPipeline (as we see in a moment). The methods used to generate SDP are generateSdpAnswerForCallee(calleeSdpOffer) and generateSdpAnswerForCaller(callerSdpOffer):

```
private void incomingCallResponse(final UserSession callee,
    JSONObject jsonMessage) throws IOException {
    String callResponse = jsonMessage.get("callResponse").getAsString();
    String from = jsonMessage.get("from").getAsString();
    final UserSession calleer = registry.getByName(from);
    String to = calleer.getCallingTo();

    if ("accept".equals(callResponse)) {
        log.debug("Accepted call from '{}' to '{}'", from, to);

        CallMediaPipeline callMediaPipeline = new CallMediaPipeline(
            kurento, from, to);
        pipelines.put(calleer.getSessionId(),
            callMediaPipeline.getPipeline());
        pipelines.put(callee.getSessionId(),
            callMediaPipeline.getPipeline());

        String calleeSdpOffer = jsonMessage.get("sdpOffer").getAsString();
        String calleeSdpAnswer = callMediaPipeline
            .generateSdpAnswerForCallee(calleeSdpOffer);

        callee.setWebRtcEndpoint(callMediaPipeline.getcalleeWebRtcEP());
        callMediaPipeline.getcalleeWebRtcEP().addIceCandidateFoundListener(
            new EventListener<IceCandidateFoundEvent>() {

                @Override
                public void onEvent(IceCandidateFoundEvent event) {
                    JSONObject response = new JSONObject();
                    response.addProperty("id", "iceCandidate");
                    response.add("candidate", JsonUtils
                        .toJsonObject(event.getCandidate()));
                    try {
                        synchronized (callee.getSession()) {
                            callee.getSession()
                                .sendMessage(
                                    new TextMessage(response
                                        .toString()));
                        }
                    } catch (IOException e) {
                        log.debug(e.getMessage());
                    }
                }
            });
    }

    JSONObject startCommunication = new JSONObject();
    startCommunication.addProperty("id", "startCommunication");
    startCommunication.addProperty("sdpAnswer", calleeSdpAnswer);

    synchronized (callee) {
        callee.sendMessage(startCommunication);
    }
}
```

```

callMediaPipeline.getcalleeWebRtcEP().gatherCandidates();

String callerSdpOffer = registry.getByName(from).getSdpOffer();

calleer.setWebRtcEndpoint(callMediaPipeline.getCallerWebRtcEP());
callMediaPipeline.getCallerWebRtcEP().addIceCandidateFoundListener(
    new EventListener<IceCandidateFoundEvent>() {

        @Override
        public void onEvent(IceCandidateFoundEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "iceCandidate");
            response.add("candidate", JsonUtils
                .toJsonObject(event.getCandidate()));
            try {
                synchronized (calleer.getSession()) {
                    calleer.getSession()
                        .sendMessage(
                            new TextMessage(response
                                .toString()));
                }
            } catch (IOException e) {
                log.debug(e.getMessage());
            }
        }
    });
}

String callerSdpAnswer = callMediaPipeline
    .generateSdpAnswerForCaller(callerSdpOffer);

JsonObject response = new JsonObject();
response.addProperty("id", "callResponse");
response.addProperty("response", "accepted");
response.addProperty("sdpAnswer", callerSdpAnswer);

synchronized (calleer) {
    calleer.sendMessage(response);
}

callMediaPipeline.getCallerWebRtcEP().gatherCandidates();

callMediaPipeline.record();

} else {
    JsonObject response = new JsonObject();
    response.addProperty("id", "callResponse");
    response.addProperty("response", "rejected");
    calleer.sendMessage(response);
}
}
}

```

Finally, the `play` method instantiates a `PlayMediaPipeline` object, which is used to create Media Pipeline in charge of the playback of the recorded streams in the Kurento Media Server.

```

private void play(final UserSession session, JsonObject jsonMessage)
    throws IOException {
    String user = jsonMessage.get("user").getAsString();
}

```

```

log.debug("Playing recorded call of user '{}', user);

JsonObject response = new JsonObject();
response.addProperty("id", "playResponse");

if (registry.getByName(user) != null
    && registry.getBySession(session.getSession()) != null) {
    final PlayMediaPipeline playMediaPipeline = new PlayMediaPipeline(
        kurento, user, session.getSession());
    String sdpOffer = jsonMessage.get("sdpOffer").getAsString();

    session.setPlayingWebRtcEndpoint(playMediaPipeline.getWebRtc());

    playMediaPipeline.getPlayer().addEndOfStreamListener(
        new EventListener<EndOfStreamEvent>() {
            @Override
            public void onEvent(EndOfStreamEvent event) {
                UserSession user = registry
                    .getBySession(session.getSession());
                releasePipeline(user);
                playMediaPipeline.sendPlayEnd(session.getSession());
            }
        });
}

playMediaPipeline.getWebRtc().addIceCandidateFoundListener(
    new EventListener<IceCandidateFoundEvent>() {

        @Override
        public void onEvent(IceCandidateFoundEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "iceCandidate");
            response.add("candidate", JsonUtils
                .toJsonObject(event.getCandidate()));
            try {
                synchronized (session) {
                    session.getSession()
                        .sendMessage(
                            new TextMessage(response
                                .toString()));
                }
            } catch (IOException e) {
                log.debug(e.getMessage());
            }
        }
    });
}

String sdpAnswer = playMediaPipeline.generateSdpAnswer(sdpOffer);

response.addProperty("response", "accepted");

response.addProperty("sdpAnswer", sdpAnswer);

playMediaPipeline.play();
pipelines.put(session.getSessionId(),
    playMediaPipeline.getPipeline());
synchronized (session.getSession()) {
    session.sendMessage(response);
}
}

```

```

        playMediaPipeline.getWebRtc().gatherCandidates();

    } else {
        response.addProperty("response", "rejected");
        response.addProperty("error", "No recording for user '" + user
            + "'. Please type a correct user in the 'Peer' field.");
        session.getSession().sendMessage(
            new TextMessage(response.toString()));
    }
}

```

The media logic in this demo is implemented in the classes `CallMediaPipeline` and `PlayMediaPipeline`. The first media pipeline consists on two `WebRtcEndpoint` elements interconnected with a `FaceOverlayFilter` in between, and also with and `RecorderEndpoint` to carry out the recording of the WebRTC communication. Please take note that the `WebRtc` endpoints needs to be connected twice, one for each media direction. In this class we can see the implementation of methods `generateSdpAnswerForCaller` and `generateSdpAnswerForCallee`. These methods delegate to `WebRtc` endpoints to create the appropriate answer.

```

public class CallMediaPipeline {

    private static final SimpleDateFormat df = new SimpleDateFormat(
        "yyyy-MM-dd_HH-mm-ss-S");
    public static final String RECORDING_PATH = "file:///tmp/"
        + df.format(new Date()) + "-";
    public static final String RECORDING_EXT = ".webm";

    private final MediaPipeline pipeline;
    private final WebRtcEndpoint webRtcCaller;
    private final WebRtcEndpoint webRtcCallee;
    private final RecorderEndpoint recorderCaller;
    private final RecorderEndpoint recorderCallee;

    public CallMediaPipeline(KurentoClient kurento, String from, String to) {

        // Media pipeline
        pipeline = kurento.createMediaPipeline();

        // Media Elements (WebRtcEndpoint, RecorderEndpoint, FaceOverlayFilter)
        webRtcCaller = new WebRtcEndpoint.Builder(pipeline).build();
        webRtcCallee = new WebRtcEndpoint.Builder(pipeline).build();

        recorderCaller = new RecorderEndpoint.Builder(pipeline, RECORDING_PATH
            + from + RECORDING_EXT).build();
        recorderCallee = new RecorderEndpoint.Builder(pipeline, RECORDING_PATH
            + to + RECORDING_EXT).build();

        String appServerUrl = System.getProperty("app.server.url",
            One2OneCallAdvApp.DEFAULT_APP_SERVER_URL);
        FaceOverlayFilter faceOverlayFilterCaller = new FaceOverlayFilter.Builder(
            pipeline).build();
        faceOverlayFilterCaller.setOverlaidImage(appServerUrl
            + "/img/mario-wings.png", -0.35F, -1.2F, 1.6F, 1.6F);

        FaceOverlayFilter faceOverlayFilterCallee = new FaceOverlayFilter.Builder(
            pipeline).build();
        faceOverlayFilterCallee.setOverlaidImage(

```

```

        appServerUrl + "/img/Hat.png", -0.2F, -1.35F, 1.5F, 1.5F);

    // Connections
    webRtcCaller.connect(faceOverlayFilterCaller);
    faceOverlayFilterCaller.connect(webRtcCallee);
    faceOverlayFilterCaller.connect(recorderCaller);

    webRtcCallee.connect(faceOverlayFilterCallee);
    faceOverlayFilterCallee.connect(webRtcCaller);
    faceOverlayFilterCallee.connect(recorderCallee);
}

public void record() {
    recorderCaller.record();
    recorderCallee.record();
}

public String generateSdpAnswerForCaller(String sdpOffer) {
    return webRtcCaller.processOffer(sdpOffer);
}

public String generateSdpAnswerForCallee(String sdpOffer) {
    return webRtcCallee.processOffer(sdpOffer);
}

public MediaPipeline getPipeline() {
    return pipeline;
}

public WebRtcEndpoint getCallerWebRtcEP() {
    return webRtcCaller;
}

public WebRtcEndpoint getCalleeWebRtcEP() {
    return webRtcCallee;
}
}

```

Note: Notice the hat URLs are provided by the application server and consumed by the KMS. This logic is assuming that the application server is hosted in local (*localhost*), and by the default the hat URLs are <https://localhost:8443/img/mario-wings.png> and <https://localhost:8443/img/Hat.png>. If your application server is hosted in a different host, it can be easily changed by means of the configuration parameter `app.server.url`, for example:

```
mvn compile exec:java -Dapp.server.url=https://app_server_host:app_server_port
```

The second media pipeline consists on a `PlayerEndpoint` connected to a `WebRtcEndpoint`. The `PlayerEndpoint` reads the previously recorded media in the file system of the Kurento Media Server. The `WebRtcEndpoint` is used in receive-only mode.

```

public class PlayMediaPipeline {

    private static final Logger log = LoggerFactory
        .getLogger(PlayMediaPipeline.class);

    private WebRtcEndpoint webRtc;

```

```
private PlayerEndpoint player;

public PlayMediaPipeline(KurentoClient kurento, String user,
    final WebSocketSession session) {
    // Media pipeline
    MediaPipeline pipeline = kurento.createMediaPipeline();

    // Media Elements (WebRtcEndpoint, PlayerEndpoint)
    webRtc = new WebRtcEndpoint.Builder(pipeline).build();
    player = new PlayerEndpoint.Builder(pipeline, RECORDING_PATH + user
        + RECORDING_EXT).build();

    // Connection
    player.connect(webRtc);

    // Player listeners
    player.addErrorListener(new EventListener<ErrorEvent>() {
        @Override
        public void onEvent(ErrorEvent event) {
            log.info("ErrorEvent: {}", event.getDescription());
            sendPlayEnd(session);
        }
    });
}

public void sendPlayEnd(WebSocketSession session) {
    try {
        JSONObject response = new JSONObject();
        response.addProperty("id", "playEnd");
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Error sending playEndOfStream message", e);
    }
}

public void play() {
    player.play();
}

public String generateSdpAnswer(String sdpOffer) {
    return webRtc.processOffer(sdpOffer);
}

public MediaPipeline getPipeline() {
    return pipeline;
}

public WebRtcEndpoint getWebRtc() {
    return webRtc;
}

public PlayerEndpoint getPlayer() {
    return player;
}

}
```

Client-Side

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`.

In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/call`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are six incoming messages to client: `resgisterResponse`, `callResponse`, `incomingCall`, `startCommunication`, `iceCandidate` and `play`. Convenient actions are taken to implement each step in the communication. On the one hand, in functions `call` and `incomingCall` (for caller and callee respectively), the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication. On the other hand in the function `play`, the function `WebRtcPeer.WebRtcPeerRecvonly` is called since the `WebRtcEndpoint` is used in receive-only.

```
var ws = new WebSocket('ws://' + location.host + '/call');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'resgisterResponse':
            resgisterResponse(parsedMessage);
            break;
        case 'callResponse':
            callResponse(parsedMessage);
            break;
        case 'incomingCall':
            incomingCall(parsedMessage);
            break;
        case 'startCommunication':
            startCommunication(parsedMessage);
            break;
        case 'stopCommunication':
            console.info("Communication ended by remote peer");
            stop(true);
            break;
        case 'playResponse':
            playResponse(parsedMessage);
            break;
        case 'playEnd':
            playEnd();
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (!error) return;
                console.error("Error adding candidate: " + error);
            });
            break;
        default:
            console.error('Unrecognized message', parsedMessage);
    }
}
```

```
function incomingCall(message) {
    // If bussy just reject without disturbing user
    if (callState != NO_CALL && callState != POST_CALL) {
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'reject',
            message : 'bussy'
        };
        return sendMessage(response);
    }

    setCallState(DISABLED);
    if (confirm('User ' + message.from
        + ' is calling you. Do you accept the call?')) {
        showSpinner(videoInput, videoOutput);

        from = message.from;
        var options = {
            localVideo: videoInput,
            remoteVideo: videoOutput,
            onicecandidate: onIceCandidate
        }
        webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
            function (error) {
                if(error) {
                    return console.error(error);
                }
                this.generateOffer (onOfferIncomingCall);
            });
    } else {
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'reject',
            message : 'user declined'
        };
        sendMessage(response);
        stop();
    }
}

function call() {
    if (document.getElementById('peer').value == '') {
        document.getElementById('peer').focus();
        window.alert("You must specify the peer name");
        return;
    }
    setCallState(DISABLED);
    showSpinner(videoInput, videoOutput);

    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput,
        onicecandidate: onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
```

```

        function (error) {
            if(error) {
                return console.error(error);
            }
            this.generateOffer (onOfferCall);
        });
    }

function play() {
    var peer = document.getElementById('peer').value;
    if (peer == '') {
        window.alert("You must insert the name of the user recording to be played");
        document.getElementById('Peer').focus();
        return;
    }

    document.getElementById('videoSmall').style.display = 'none';
    setCallState(DISABLED);
    showSpinner(videoOutput);

    var options = {
        remoteVideo: videoOutput,
        onicecandidate: onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
        function (error) {
            if(error) {
                return console.error(error);
            }
            this.generateOffer (onOfferPlay);
        });
}

function stop(message) {
    var stopMessageId = (callState == IN_CALL) ? 'stop' : 'stopPlay';
    setCallState(POST_CALL);
    if (webRtcPeer) {
        webRtcPeer.dispose();
        webRtcPeer = null;

        if (!message) {
            var message = {
                id : stopMessageId
            }
            sendMessage(message);
        }
    }
    hideSpinner(videoInput, videoOutput);
    document.getElementById('videoSmall').style.display = 'block';
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency

(*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side. Other client libraries are managed with webjars:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>draggabilly</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>jquery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>ekko-lightbox</artifactId>
    </dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

6.7 WebRTC Many-To-Many video call (Group Call)

This tutorial connects several participants to the same video conference. A group call will consist (in the media server side) in N*N WebRTC endpoints, where N is the number of clients connected to that conference.

6.7.1 Java - Group Call

This tutorial shows how to work with the concept of rooms, allowing to connect several clients between them using [WebRTC](#) technology, creating a multiconference.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-group-call
git checkout 6.7.1
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This tutorial shows how to work with the concept of rooms. Each room will create its own pipeline, being isolated from the other rooms. Clients connecting to a certain room, will only be able to exchange media with clients in the same room.

Each client will send its own media, and in turn will receive the media from all the other participants. This means that there will be a total of n^2 webrtc endpoints in each room, where n is the number of clients.

When a new client enters the room, a new webrtc will be created and negotiated receive the media on the server. On the other hand, all participant will be informed that a new user has connected. Then, all participants will request the server to receive the new participant's media.

The newcomer, in turn, gets a list of all connected participants, and requests the server to receive the media from all the present clients in the room.

When a client leaves the room, all clients are informed by the server. Then, the client-side code requests the server to cancel all media elements related to the client that left.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in [JavaScript](#). At the server-side, we use a Spring-Boot based server application consuming the [Kurento Java Client API](#), to control [Kurento Media Server](#) capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the

communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side with [Spring Boot](#) framework. This technology can be used to embed the Tomcat web server in the application and thus simplify the development process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is `GroupCallApp`. As you can see, the *KurentoClient* is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at *localhost* listening in port 8888. If you reproduce this example you'll need to insert the specific location of your Kurento Media Server instance there.

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@EnableWebSocket
@SpringBootApplication
public class GroupCallApp implements WebSocketConfigurer {

    @Bean
    public UserRegistry registry() {
        return new UserRegistry();
    }

    @Bean
    public RoomManager roomManager() {
        return new RoomManager();
    }

    @Bean
    public CallHandler groupCallHandler() {
        return new CallHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(GroupCallApp.class, args);
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(groupCallHandler(), "/groupcall");
    }
}
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/groupcall`.

`CallHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are five different kind of incoming messages to the application server: `joinRoom`, `receiveVideoFrom`, `leaveRoom` and `onIceCandidate`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```
public class CallHandler extends TextWebSocketHandler {

    private static final Logger log = LoggerFactory.getLogger(CallHandler.class);

    private static final Gson gson = new GsonBuilder().create();

    @Autowired
    private RoomManager roomManager;

    @Autowired
    private UserRegistry registry;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws
    Exception {
        final JSONObject jsonMessage = gson.fromJson(message.getPayload(), JSONObject.
        class);

        final UserSession user = registry.getBySession(session);

        if (user != null) {
            log.debug("Incoming message from user '{}': {}", user.getName(), jsonMessage);
        } else {
            log.debug("Incoming message from new user: {}", jsonMessage);
        }

        switch (jsonMessage.get("id").getAsString()) {
            case "joinRoom":
                joinRoom(jsonMessage, session);
                break;
            case "receiveVideoFrom":
                final String senderName = jsonMessage.get("sender").getAsString();
                final UserSession sender = registry.getByName(senderName);
                final String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
                user.receiveVideoFrom(sender, sdpOffer);
                break;
            case "leaveRoom":
                leaveRoom(user);
                break;
            case "onIceCandidate":
                JsonObject candidate = jsonMessage.get("candidate").getAsJsonObject();

                if (user != null) {
                    IceCandidate cand = new IceCandidate(candidate.get("candidate").
                    getAsString(),

```

```

        candidate.get("sdpMid").getAsString(), candidate.get("sdpMLineIndex") .
→getAsInt());
        user.addCandidate(cand, jsonMessage.get("name").getAsString());
    }
    break;
default:
    break;
}
}

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status) ↵
throws Exception {
    ...
}

private void joinRoom(JsonObject params, WebSocketSession session) throws ↵
IOException {
    ...
}

private void leaveRoom(UserSession user) throws IOException {
    ...
}
}

```

In the following snippet, we can see the `afterConnectionClosed` method. Basically, it removes the `UserSession` from registry and throws out the user from the room.

```

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status) ↵
throws Exception {
    UserSession user = registry.removeBySession(session);
    roomManager.getRoom(user.getRoomName()).leave(user);
}

```

In the `joinRoom` method, the server checks if there are a registered room with the name specified, add the user into this room and registries the user.

```

private void joinRoom(JsonObject params, WebSocketSession session) throws IOException
{
    final String roomName = params.get("room").getAsString();
    final String name = params.get("name").getAsString();
    log.info("PARTICIPANT {}: trying to join room {}", name, roomName);

    Room room = roomManager.getRoom(roomName);
    final UserSession user = room.join(name, session);
    registry.register(user);
}

```

The `leaveRoom` method finish the video call from one user.

```

private void leaveRoom(UserSession user) throws IOException {
    final Room room = roomManager.getRoom(user.getRoomName());
    room.leave(user);
    if (room.getParticipants().isEmpty()) {
        roomManager.removeRoom(room);
    }
}

```

```
}
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `conferenceroom.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/groupcall`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `existingParticipants`, `newParticipantArrived`, `participantLeft`, `receiveVideoAnswer` and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions start the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```
var ws = new WebSocket('wss://' + location.host + '/groupcall');
var participants = {};
var name;

window.onbeforeunload = function() {
    ws.close();
};

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'existingParticipants':
            onExistingParticipants(parsedMessage);
            break;
        case 'newParticipantArrived':
            onNewParticipant(parsedMessage);
            break;
        case 'participantLeft':
            onParticipantLeft(parsedMessage);
            break;
        case 'receiveVideoAnswer':
            receiveVideoResponse(parsedMessage);
            break;
        case 'iceCandidate':
            participants[parsedMessage.name].rtcPeer.addIceCandidate(parsedMessage.
            ↵candidate, function (error) {
                if (error) {
                    console.error("Error adding candidate: " + error);
                    return;
                }
            });
            break;
        default:
            console.error('Unrecognized message', parsedMessage);
    }
}
```

```

function register() {
    name = document.getElementById('name').value;
    var room = document.getElementById('roomName').value;

    document.getElementById('room-header').innerText = 'ROOM ' + room;
    document.getElementById('join').style.display = 'none';
    document.getElementById('room').style.display = 'block';

    var message = {
        id : 'joinRoom',
        name : name,
        room : room,
    }
    sendMessage(message);
}

function onNewParticipant(request) {
    receiveVideo(request.name);
}

function receiveVideoResponse(result) {
    participants[result.name].rtcPeer.processAnswer(result.sdpAnswer, function(error) {
        if (error) return console.error(error);
    });
}

function callResponse(message) {
    if (message.response != 'accepted') {
        console.info('Call not accepted by peer. Closing call');
        stop();
    } else {
        webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
            if (error) return console.error(error);
        });
    }
}

function onExistingParticipants(msg) {
    var constraints = {
        audio : true,
        video : {
            mandatory : {
                maxWidth : 320,
                maxFrameRate : 15,
                minFrameRate : 15
            }
        }
    };
    console.log(name + " registered in room " + room);
    var participant = new Participant(name);
    participants[name] = participant;
    var video = participant.getVideoElement();

    var options = {
        localVideo: video,
        mediaConstraints: constraints,
}

```

```

        onicecandidate: participant.onIceCandidate.bind(participant)
    }
    participant.rtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendonly(options,
        function (error) {
            if(error) {
                return console.error(error);
            }
            this.generateOffer (participant.offerToReceiveVideo.bind(participant));
        });

    msg.data.forEach(receiveVideo);
}

function leaveRoom() {
    sendMessage({
        id : 'leaveRoom'
    });

    for ( var key in participants) {
        participants[key].dispose();
    }

    document.getElementById('join').style.display = 'block';
    document.getElementById('room').style.display = 'none';

    ws.close();
}

function receiveVideo(sender) {
    var participant = new Participant(sender);
    participants[sender] = participant;
    var video = participant.getVideoElement();

    var options = {
        remoteVideo: video,
        onicecandidate: participant.onIceCandidate.bind(participant)
    }

    participant.rtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
        function (error) {
            if(error) {
                return console.error(error);
            }
            this.generateOffer (participant.offerToReceiveVideo.bind(participant));
        });
}

function onParticipantLeft(request) {
    console.log('Participant ' + request.name + ' left');
    var participant = participants[request.name];
    participant.dispose();
    delete participants[request.name];
}

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('Senging message: ' + jsonMessage);
    ws.send(jsonMessage);
}

```

```
}
```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with webjars:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>jquery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>ekko-lightbox</artifactId>
    </dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

6.8 Media Elements metadata

This tutorial detects and draws faces present in the webcam video. It connects filters: KmsDetectFaces and the KmsShowFaces.

6.8.1 Java - Metadata

This tutorial detects and draws faces into the webcam video. The demo connects two filters, the KmsDetectFaces and the KmsShowFaces.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-metadata-example
git checkout 6.7.1
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Note: This demo needs the `kms-datachannelexample` module installed in the media server. That module is available in the Kurento repositories, so it is possible to install it with:

```
sudo apt-get install kms-datachannelexample
```

Understanding this example

To implement this behavior we have to create a [Media Pipeline](#) composed by one **WebRtcEndpoint** and two filters **KmsDetectFaces** and **KmsShowFaces**. The first one detects faces into the image and it puts the info about the face (position and dimensions) into the buffer metadata. The second one reads the buffer metadata to find info about detected faces. If there is info about faces, the filter draws the faces into the image.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento**

Java Client API, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the *Spring Boot* framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is `MetadataApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at `localhost`, listening in port 8888. If you reproduce this example, you'll need to insert the specific location of your Kurento Media Server instance there.

Once the `Kurento Client` has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@EnableWebSocket
@SpringBootApplication
public class MetadataApp implements WebSocketConfigurer {

    static final String DEFAULT_APP_SERVER_URL = "https://localhost:8443";

    @Bean
    public MetadataHandler handler() {
        return new MetadataHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(handler(), "/metadata");
    }

    public static void main(String[] args) throws Exception {
        new SpringApplication(MetadataApp.class).run(args);
    }
}
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/metadata`.

`MetadataHandler` class implements `TextWebSocketHandler` to handle text WebSocket requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the WebSocket. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kinds of incoming messages to the *Server*: start, stop and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```
public class MetadataHandler extends TextWebSocketHandler {

    private final Logger log = LoggerFactory.getLogger(MetadataHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, UserSession> users = new ConcurrentHashMap<String, UserSession>();
    @Autowired
    private KurentoClient kurento;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {
        JSONObject jsonMessage = gson.fromJson(message.getPayload(), JSONObject.class);
        log.debug("Incoming message: {}", jsonMessage);

        switch (jsonMessage.get("id").getAsString()) {
            case "start":
                start(session, jsonMessage);
                break;
            case "stop":
                UserSession user = users.remove(session.getId());
                if (user != null) {
                    user.release();
                }
                break;
            case "onIceCandidate":
                JSONObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

                UserSession user = users.get(session.getId());
                if (user != null) {
                    IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate").
                    getAsString(),
                        jsonCandidate.get("sdpMid").getAsString(),
                        jsonCandidate.get("sdpMLineIndex").getAsInt());
                    user.addCandidate(candidate);
                }
                break;
            default:
                sendError(session, "Invalid message with id " + jsonMessage.get("id").
                getAsString());
                break;
        }
    }

    private void start(final WebSocketSession session, JSONObject jsonMessage) {
        ...
    }
}
```

```

    }

    private void sendError(WebSocketSession session, String message) {
        ...
    }
}

```

In the following snippet, we can see the `start` method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (WebRtcEndpoint, KmsShowFaces and KmsDetectFaces) and make the connections among them. A `startResponse` message is sent back to the client with the SDP answer.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // User session
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
        user.setWebRtcEndpoint(webRtcEndpoint);
        users.put(session.getId(), user);

        // ICE candidates
        webRtcEndpoint.addIceCandidateFoundListener(new EventListener
        ↵<IceCandidateFoundEvent>() {
            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {
                    synchronized (session) {
                        session.sendMessage(new TextMessage(response.toString()));
                    }
                } catch (IOException e) {
                    log.debug(e.getMessage());
                }
            }
        });
    }

    // Media logic
    KmsShowFaces showFaces = new KmsShowFaces.Builder(pipeline).build();
    KmsDetectFaces detectFaces = new KmsDetectFaces.Builder(pipeline).build();

    webRtcEndpoint.connect(detectFaces);
    detectFaces.connect(showFaces);
    showFaces.connect(webRtcEndpoint);

    // SDP negotiation (offer and answer)
    String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
    String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

    JsonObject response = new JsonObject();
    response.addProperty("id", "startResponse");
    response.addProperty("sdpAnswer", sdpAnswer);

    synchronized (session) {
        session.sendMessage(new TextMessage(response.toString()));
    }
}

```

```

        webRtcEndpoint.gatherCandidates();

    } catch (Throwable t) {
        sendError(session, t.getMessage());
    }
}

```

The `sendError` method is quite simple: it sends an `error` message to the client when an exception is caught in the server-side.

```

private void sendError(WebSocketSession session, String message) {
    try {
        JSONObject response = new JSONObject();
        response.addProperty("id", "error");
        response.addProperty("message", message);
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Exception sending message", e);
    }
}

```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called **kurento-utils.js** to simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/metadata`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```

var ws = new WebSocket('wss://' + location.host + '/metadata');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError("Error message from server: " + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (error) {

```

```
        console.error("Error adding candidate: " + error);
        return;
    }
});
break;
default:
    if (state == I_AM_STARTING) {
        setState(I_CAN_START);
    }
    onError('Unrecognized message', parsedMessage);
}
}

function start() {
    console.log("Starting video call ...")
    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(videoInput, videoOutput);

    console.log("Creating WebRtcPeer and generating local sdp offer ...");

    var options = {
        localVideo : videoInput,
        remoteVideo : videoOutput,
        onicecandidate : onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function(error) {
            if (error) {
                return console.error(error);
            }
            webRtcPeer.generateOffer(onOffer);
        });
}

function onOffer(error, offerSdp) {
    if (error)
        return console.error("Error generating the offer");
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}

function onError(error) {
    console.error(error);
}

function onIceCandidate(candidate) {
    console.log("Local candidate" + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
    };
    sendMessage(message);
}
```

```

}

function startResponse(message) {
    setState(I_CAN_STOP);
    console.log("SDP answer received from server. Processing ...");

    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function stop() {
    console.log("Stopping video call ...");
    setState(I_CAN_START);
    if (webRtcPeer) {
        webRtcPeer.dispose();
        webRtcPeer = null;

        var message = {
            id : 'stop'
        }
        sendMessage(message);
    }
    hideSpinner(videoInput, videoOutput);
}

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('Senging message: ' + jsonMessage);
    ws.send(jsonMessage);
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>

```

```
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>demo-console</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>adapter.js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

6.9 WebRTC Media Player

This tutorial reads a file from disk and plays the video to WebRTC.

6.9.1 Java - Player

This tutorial opens a URL and plays its content to WebRTC where it is possible to choose if it plays video and audio, only video or only audio.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-player
```

```
git checkout 6.7.1
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

To implement this behavior we have to create a [Media Pipeline](#) composed by one **PlayerEndpoint** and one **WebRtcEndpoint**. The **PlayerEndpoint** plays a video and **WebRtcEndpoint** shows it.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the [Spring Boot](#) framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is `PlayerApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it's located at `localhost` listening in port 8888. If you reproduce this example you'll need to insert the specific location of your Kurento Media Server instance there.

Once the `Kurento Client` has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@EnableWebSocket
@SpringBootApplication
public class PlayerApp implements WebSocketConfigurer {

    private static final String KMS_WS_URI_PROP = "kms.url";
    private static final String KMS_WS_URI_DEFAULT = "ws://localhost:8888/kurento";
```

```

@Bean
public PlayerHandler handler() {
    return new PlayerHandler();
}

@Bean
public KurentoClient kurentoClient() {
    return KurentoClient.create(System.getProperty(KMS_WS_URI_PROP, KMS_WS_URI_
 DEFAULT));
}

@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(handler(), "/player");
}

public static void main(String[] args) throws Exception {
    new SpringApplication(PlayerApp.class).run(args);
}
}

```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/player`.

`PlayerHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol, there are seven different kinds of incoming messages to the *Server*: `start`, `stop`, `pause`, `resume`, `doSeek`, `getPosition` and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```

public class PlayerHandler extends TextWebSocketHandler {

    @Autowired
    private KurentoClient kurento;

    private final Logger log = LoggerFactory.getLogger(PlayerHandler.class);
    private final Gson gson = new GsonBuilder().create();
    private final ConcurrentHashMap<String, PlayerMediaPipeline> pipelines =
        new ConcurrentHashMap<>();

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws_
    Exception {
        JSONObject jsonMessage = gson.fromJson(message.getPayload(), JSONObject.class);
        String sessionId = session.getId();
        log.debug("Incoming message {} from sessionId", jsonMessage, sessionId);

        try {
            switch (jsonMessage.get("id").getAsString()) {
                case "start":
                    start(session, jsonMessage);
                    break;

```

```

    case "stop":
        stop(sessionId);
        break;
    case "pause":
        pause(sessionId);
        break;
    case "resume":
        resume(session);
        break;
    case "doSeek":
        doSeek(session, jsonMessage);
        break;
    case "getPosition":
        getPosition(session);
        break;
    case "onIceCandidate":
        onIceCandidate(sessionId, jsonMessage);
        break;
    default:
        sendError(session, "Invalid message with id " + jsonMessage.get("id").
        ↪getAsString());
        break;
    }
} catch (Throwable t) {
    log.error("Exception handling message {} in sessionId {}", jsonMessage,
    ↪sessionId, t);
    sendError(session, t.getMessage());
}
}

private void start(final WebSocketSession session, JSONObject jsonMessage) {
    ...
}

private void pause(String sessionId) {
    ...
}

private void resume(final WebSocketSession session) {
    ...
}

private void doSeek(final WebSocketSession session, JSONObject jsonMessage) {
    ...
}

private void getPosition(final WebSocketSession session) {
    ...
}

private void stop(String sessionId) {
    ...
}

private void sendError(WebSocketSession session, String message) {
    ...
}

```

```
}
```

In the following snippet, we can see the `start` method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (`WebRtcEndpoint` and `PlayerEndpoint`) and makes the connections between them and plays the video. A `startResponse` message is sent back to the client with the SDP answer. When the `MediaConnected` event is received, info about the video is retrieved and sent back to the client in a `videoInfo` message.

```
private void start(final WebSocketSession session, JsonObject jsonMessage) {
    final UserSession user = new UserSession(); MediaPipeline pipeline =
    kurento.createMediaPipeline(); user.setMediaPipeline(pipeline);
    WebRtcEndpoint webRtcEndpoint = new
    WebRtcEndpoint.Builder(pipeline).build();
    user.setWebRtcEndpoint(webRtcEndpoint); String videourl =
    jsonMessage.get("videourl").getAsString(); final PlayerEndpoint
    playerEndpoint = new PlayerEndpoint.Builder(pipeline, videourl).build();
    user.setPlayerEndpoint(playerEndpoint); users.put(session.getId(), user);

    playerEndpoint.connect(webRtcEndpoint);

    // 2. WebRtcEndpoint // ICE candidates
    webRtcEndpoint.addIceCandidateFoundListener(new
    EventListener<IceCandidateFoundEvent>() {
        @Override public void onEvent(IceCandidateFoundEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "iceCandidate"); response.add("candidate",
            JsonUtils.toJsonObject(event.getCandidate())); try {
                synchronized (session) {
                    session.sendMessage(new
                    TextMessage(response.toString()));
                }
            } catch (IOException e) {
                log.debug(e.getMessage());
            }
        }
    });
}

String sdpOffer = jsonMessage.get("sdpOffer").getAsString(); String
sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

JsonObject response = new JsonObject(); response.addProperty("id",
"startResponse"); response.addProperty("sdpAnswer", sdpAnswer);
sendMessage(session, response.toString());

webRtcEndpoint.addMediaStateChangedListener(new
EventListener<MediaStateChangedEvent>() {
    @Override public void onEvent(MediaStateChangedEvent event) {

        if (event.getNewState() == MediaState.CONNECTED) {
            VideoInfo videoInfo = playerEndpoint.getVideoInfo();

            JsonObject response = new JsonObject();
            response.addProperty("id", "videoInfo");
            response.addProperty("isSeekable", videoInfo.getIsSeekable());
            response.addProperty("initSeekable", videoInfo.getSeekableInit());
            response.addProperty("endSeekable", videoInfo.getSeekableEnd());
            response.addProperty("videoDuration", videoInfo.getDuration());
            sendMessage(session, response.toString());
        }
    }
});
```

```

        }
    }
});

webRtcEndpoint.gatherCandidates();

// 3. PlayEndpoint playerEndpoint.addErrorListener(new
EventListener<ErrorEvent>() {
    @Override public void onEvent(ErrorEvent event) {
        log.info("ErrorEvent: {}", event.getDescription());
        sendPlayEnd(session);
    }
});

playerEndpoint.addEndOfStreamListener(new
EventListener<EndOfStreamEvent>() {
    @Override public void onEvent(EndOfStreamEvent event) {
        log.info("EndOfStreamEvent: {}", event.getTimestamp());
        sendPlayEnd(session);
    }
});

playerEndpoint.play();
}

```

The pause method retrieves the *user* associated to the current session, and invokes the *pause* method on the PlayerEndpoint.

```

private void pause(String sessionId) {
    UserSession user = users.get(sessionId);

    if (user != null) {
        user.getPlayerEndpoint().pause();
    }
}

```

The resume method starts the PlayerEndpoint of the current user, sending back the information about the video, so the client side can refresh the stats.

```

private void resume(String sessionId) {
    UserSession user = users.get(sessionId.getId());

    if (user != null) {
        user.getPlayerEndpoint().play(); VideoInfo videoInfo =
        user.getPlayerEndpoint().getVideoInfo();

        JSONObject response = new JSONObject(); response.addProperty("id",
"videoInfo"); response.addProperty("isSeekable",
videoInfo.getIsSeekable()); response.addProperty("initSeekable",
videoInfo.getSeekableInit()); response.addProperty("endSeekable",
videoInfo.getSeekableEnd()); response.addProperty("videoDuration",
videoInfo.getDuration()); sendMessage(session, response.toString());
    }
}

```

The doSeek method gets the *user* by *sessionId*, and calls the method setPosition of the PlayerEndpoint with the new playing position. A seek message is sent back to the client if the seek fails.

```
private void doSeek(final WebSocketSession session, JSONObject jsonMessage) {
    UserSession user = users.get(session.getId());

    if (user != null) {
        try {
            user.getPlayerEndpoint().setPosition(jsonMessage.get("position").getAsLong());
        } catch (KurentoException e) {
            log.debug("The seek cannot be performed"); JSONObject response =
                new JSONObject(); response.addProperty("id", "seek");
            response.addProperty("message", "Seek failed"); sendMessage(session,
                response.toString());
        }
    }
}
```

The `getPosition` calls the method `getPosition` of the `PlayerEndpoint` of the current `user`. A position message is sent back to the client with the actual position of the video.

```
private void getPosition(final WebSocketSession session) {
    UserSession user = users.get(session.getId());

    if (user != null) {
        long position = user.getPlayerEndpoint().getPosition();

        JSONObject response = new JSONObject(); response.addProperty("id",
            "position"); response.addProperty("position", position);
        sendMessage(session, response.toString());
    }
}
```

The `stop` method is quite simple: it searches the `user` by `sessionId` and stops the `PlayerEndpoint`. Finally, it releases the media elements and removes the user from the list of active users.

```
private void stop(String sessionId) {
    UserSession user = users.remove(sessionId);

    if (user != null) {
        user.release();
    }
}
```

The `sendError` method is quite simple: it sends an `error` message to the client when an exception is caught in the server-side.

```
private void sendError(WebSocketSession session, String message) {
    try {
        JSONObject response = new JSONObject(); response.addProperty("id",
            "error"); response.addProperty("message", message);
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Exception sending message", e);
    }
}
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/player`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are seven incoming messages to `client`: `startResponse`, `playEnd`, `error`, `videoInfo`, `seek`, `position` and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer.WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```
var ws = new WebSocket('wss://' + location.host + '/player');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Error message from server: ' + parsedMessage.message);
            break;
        case 'playEnd':
            playEnd();
            break;
            break;
        case 'videoInfo':
            showVideoData(parsedMessage);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (error)
                    return console.error('Error adding candidate: ' + error);
            });
            break;
        case 'seek':
            console.log (parsedMessage.message);
            break;
        case 'position':
            document.getElementById("videoPosition").value = parsedMessage.position;
            break;
        default:
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Unrecognized message', parsedMessage);
    }
}
```

```

function start() {
    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(video);

    var mode = $('input[name="mode"]:checked').val();
    console
        .log('Creating WebRtcPeer in ' + mode + " mode and generating local sdp\u2192offer ...');

    // Video and audio by default
    var userMediaConstraints = {
        audio : true,
        video : true
    }

    if (mode == 'video-only') {
        userMediaConstraints.audio = false;
    } else if (mode == 'audio-only') {
        userMediaConstraints.video = false;
    }

    var options = {
        remoteVideo : video,
        mediaConstraints : userMediaConstraints,
        onicecandidate : onIceCandidate
    }

    console.info('User media constraints' + userMediaConstraints);

    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
        function(error) {
            if (error)
                return console.error(error);
            webRtcPeer.generateOffer(onOffer);
        });
}

function onOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);

    var message = {
        id : 'start',
        sdpOffer : offerSdp,
        videourl : document.getElementById('videourl').value
    }
    sendMessage(message);
}

function onError(error) {
    console.error(error);
}

function onIceCandidate(candidate) {
    console.log('Local candidate' + JSON.stringify(candidate));
}

```

```

var message = {
    id : 'onIceCandidate',
    candidate : candidate
}
sendMessage(message);
}

function startResponse(message) {
    setState(I_CAN_STOP);
    console.log('SDP answer received from server. Processing ...');

    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function pause() {
    togglePause()
    console.log('Pausing video ...');
    var message = {
        id : 'pause'
    }
    sendMessage(message);
}

function resume() {
    togglePause()
    console.log('Resuming video ...');
    var message = {
        id : 'resume'
    }
    sendMessage(message);
}

function stop() {
    console.log('Stopping video ...');
    setState(I_CAN_START);
    if (webRtcPeer) {
        webRtcPeer.dispose();
        webRtcPeer = null;

        var message = {
            id : 'stop'
        }
        sendMessage(message);
    }
    hideSpinner(video);
}

function playEnd() {
    setState(I_CAN_START);
    hideSpinner(video);
}

function doSeek() {
    var message = {
        id : 'doSeek',

```

```
        position: document.getElementById("seekPosition").value
    }
    sendMessage(message);
}

function getPosition() {
    var message = {
        id : 'getPosition'
    }
    sendMessage(message);
}

function showVideoData(parsedMessage) {
//Show video info
isSeekable = parsedMessage.isSeekable;
if (isSeekable) {
    document.getElementById('isSeekable').value = "true";
    enableButton('#doSeek', 'doSeek()');
} else {
    document.getElementById('isSeekable').value = "false";
}

document.getElementById('initSeek').value = parsedMessage.initSeekable;
document.getElementById('endSeek').value = parsedMessage.endSeekable;
document.getElementById('duration').value = parsedMessage.videoDuration;

enableButton('#getPosition', 'getPosition()');
}

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('Senging message: ' + jsonMessage);
    ws.send(jsonMessage);
}
```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with `webjars`:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
```

```

<groupId>org.webjars.bower</groupId>
<artifactId>bootstrap</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>demo-console</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>adapter.js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```

<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>

```

6.10 WebRTC outgoing Data Channels

This tutorial injects video into a QR filter and then sends the stream to WebRTC. QR detection events are delivered by means of WebRTC Data Channels, to be displayed in browser.

6.10.1 Java - Send DataChannel

This tutorial connects a player with a QR code detection filter and sends output to WebRTC. Code detection events are sent to browser using WebRTC datachannels.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-send-data-channel
git checkout 6.7.1
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Note: This demo needs the `kms-datachannelexample` module installed in the media server. That module is available in the Kurento repositories, so it is possible to install it with:

```
sudo apt-get install kms-datachannelexample
```

Understanding this example

To implement this behavior we have to create a *Media Pipeline* composed by one **PlayerEndpoint**, one **KmsSendData** and one **WebRtcEndpoint**. The **PlayerEndpoint** plays a video and it detects QR codes into the images. The info about detected codes is sent through data channels (**KmsSendData**) from the Kurento media server to the browser (**WebRtcEndpoint**). The browser shows the info in a text form.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the **Spring Boot** framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is `SendDataChannelApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media

Server. In this example, we assume it is located at *localhost* listening in port 8888. If you reproduce this example you'll need to insert the specific location of your Kurento Media Server instance there.

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@EnableWebSocket
@SpringBootApplication
public class SendDataChannelApp implements WebSocketConfigurer {

    static final String DEFAULT_APP_SERVER_URL = "https://localhost:8443";

    @Bean
    public SendDataChannelHandler handler() {
        return new SendDataChannelHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(handler(), "/senddatachannel");
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SendDataChannelApp.class, args);
    }
}
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/senddatachannel`.

`SendDataChannelHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kinds of incoming messages to the *Server*: `start`, `stop` and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```
public class SendDataChannelHandler extends TextWebSocketHandler {

    private final Logger log = LoggerFactory.getLogger(SendDataChannelHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, UserSession> users = new ConcurrentHashMap<>();
    ↵();

    @Autowired
    private KurentoClient kurento;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws
    ↵Exception {
```

```

JsonObject jsonMessage = gson.fromJson(message.getPayload(), JsonObject.class);

log.debug("Incoming message: {}", jsonMessage);

switch (jsonMessage.get("id").getAsString()) {
    case "start":
        start(session, jsonMessage);
        break;
    case "stop": {
        UserSession user = users.remove(session.getId());
        if (user != null) {
            user.release();
        }
        break;
    }
    case "onIceCandidate": {
        JsonObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

        UserSession user = users.get(session.getId());
        if (user != null) {
            IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate") .
→getAsString(),
                jsonCandidate.get("sdpMid").getAsString(),
                jsonCandidate.get("sdpMLineIndex").getAsInt());
            user.addCandidate(candidate);
        }
        break;
    }
    default:
        sendError(session, "Invalid message with id " + jsonMessage.get("id") .
→getAsString());
        break;
    }
}

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    ...
}

private void sendError(WebSocketSession session, String message) {
    ...
}
}

```

In the following snippet, we can see the `start` method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (`WebRtcEndpoint`, `KmsSendData` and `PlayerEndpoint`) and make the connections among them. A `startResponse` message is sent back to the client with the SDP answer.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // User session
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline) .
→useDataChannels()
            .build();
        user.setWebRtcEndpoint(webRtcEndpoint);
    }
}

```

```

PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline,
    "http://files.kurento.org/video/filter/barcodes.webm").build();
user.setPlayer(player);
users.put(session.getId(), user);

// ICE candidates
webRtcEndpoint.addIceCandidateFoundListener(new EventListener
↳<IceCandidateFoundEvent>() {
    @Override
    public void onEvent(IceCandidateFoundEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
        try {
            synchronized (session) {
                session.sendMessage(new TextMessage(response.toString()));
            }
        } catch (IOException e) {
            log.debug(e.getMessage());
        }
    }
});

// Media logic
KmsSendData kmsSendData = new KmsSendData.Builder(pipeline).build();

player.connect(kmsSendData);
kmsSendData.connect(webRtcEndpoint);

// SDP negotiation (offer and answer)
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);

synchronized (session) {
    session.sendMessage(new TextMessage(response.toString()));
}

webRtcEndpoint.gatherCandidates();
player.play();

} catch (Throwable t) {
    sendError(session, t.getMessage());
}
}
}

```

The `sendError` method is quite simple: it sends an `error` message to the client when an exception is caught in the server-side.

```

private void sendError(WebSocketSession session, String message) {
    try {
        JsonObject response = new JsonObject();
        response.addProperty("id", "error");
        response.addProperty("message", message);
        session.sendMessage(new TextMessage(response.toString()));
    }
}

```

```

    } catch (IOException e) {
        log.error("Exception sending message", e);
    }
}

```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/senddatachannel`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer`.`WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```

var ws = new WebSocket('wss://' + location.host + '/senddatachannel');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError("Error message from server: " + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (error) {
                    console.error("Error adding candidate: " + error);
                    return;
                }
            });
            break;
        default:
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError('Unrecognized message', parsedMessage);
    }
}

function start() {
    console.log("Starting video call ...")
    // Disable start button
    setState(I_AM_STARTING);
}

```

```

showSpinner(videoOutput);

var servers = null;
var configuration = null;
var peerConnection = new RTCPeerConnection(servers, configuration);

console.log("Creating channel");
var dataConstraints = null;

channel = peerConnection.createDataChannel(getChannelName (), dataConstraints);

channel.onmessage = onMessage;

var dataChannelReceive = document.getElementById('dataChannelReceive');

function onMessage (event) {
  console.log("Received data " + event["data"]);
  dataChannelReceive.value = event["data"];
}

console.log("Creating WebRtcPeer and generating local sdp offer ...");

var options = {
  peerConnection: peerConnection,
  remoteVideo : videoOutput,
  onicecandidate : onIceCandidate
}
webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
  function(error) {
    if (error) {
      return console.error(error);
    }
    webRtcPeer.generateOffer(onOffer);
  });
}

function closeChannels(){

  if(channel){
    channel.close();
    $('#dataChannelSend').disabled = true;
    $('#send').attr('disabled', true);
    channel = null;
  }
}

function onOffer(error, offerSdp) {
  if (error)
    return console.error("Error generating the offer");
  console.info('Invoking SDP offer callback function ' + location.host);
  var message = {
    id : 'start',
    sdpOffer : offerSdp
  }
  sendMessage(message);
}

function onError(error) {

```

```

        console.error(error);
    }

    function onIceCandidate(candidate) {
        console.log("Local candidate" + JSON.stringify(candidate));

        var message = {
            id : 'onIceCandidate',
            candidate : candidate
        };
        sendMessage(message);
    }

    function startResponse(message) {
        setState(I_CAN_STOP);
        console.log("SDP answer received from server. Processing ...");

        webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
            if (error)
                return console.error(error);
        });
    }

    function stop() {
        console.log("Stopping video call ...");
        setState(I_CAN_START);
        if (webRtcPeer) {
            closeChannels();

            webRtcPeer.dispose();
            webRtcPeer = null;

            var message = {
                id : 'stop'
            }
            sendMessage(message);
        }
        hideSpinner(videoOutput);
    }

    function sendMessage(message) {
        var jsonMessage = JSON.stringify(message);
        console.log('Senging message: ' + jsonMessage);
        ws.send(jsonMessage);
    }
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>

```

```

<artifactId>kurento-client</artifactId>
</dependency>
<dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-utils-js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>bootstrap</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>demo-console</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>adapter.js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```

<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>

```

6.11 WebRTC incoming Data Channel

This tutorial shows how text messages sent from browser can be delivered by Data Channels, to be displayed together with loopback video.

6.11.1 Java - Show DataChannel

This demo allows sending text from browser to the media server through data channels. That text will be shown in the loopback video.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-show-data-channel
git checkout 6.7.1
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Note: This demo needs the `kms-datachannelexample` module installed in the media server. That module is available in the Kurento repositories, so it is possible to install it with:

```
sudo apt-get install kms-datachannelexample
```

Understanding this example

This tutorial creates a [Media Pipeline](#) consisting of media elements: **WebRtcEndpoint** and **KmsSendData**. Any text inserted in the textbox is sent from Kurento Media Server (**KmsSendData**) back to browser (**WebRtcEndpoint**) and shown with loopback video.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the *Spring Boot* framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is `ShowDataChannelApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it's located at *localhost* listening in port 8888. If you reproduce this example you'll need to insert the specific location of your Kurento Media Server instance there.

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@EnableWebSocket
@SpringBootApplication
public class ShowDataChannelApp implements WebSocketConfigurer {

    static final String DEFAULT_APP_SERVER_URL = "https://localhost:8443";

    @Bean
    public ShowDataChannelHandler handler() {
        return new ShowDataChannelHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(handler(), "/showdatachannel");
    }

    public static void main(String[] args) throws Exception {
        new SpringApplication(ShowDataChannelApp.class).run(args);
    }
}
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/showdatachannel`.

`ShowDataChannelHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kinds of incoming messages to the *Server* : `start`, `stop` and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```

public class ShowDataChannelHandler extends TextWebSocketHandler {

    private final Logger log = LoggerFactory.getLogger(ShowDataChannelHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    private final ConcurrentHashMap<String, UserSession> users = new ConcurrentHashMap<>
        ();

    @Autowired
    private KurentoClient kurento;

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws
    Exception {
        JSONObject jsonMessage = gson.fromJson(message.getPayload(), JSONObject.class);

        log.debug("Incoming message: {}", jsonMessage);

        switch (jsonMessage.get("id").getAsString()) {
            case "start":
                start(session, jsonMessage);
                break;
            case "stop": {
                UserSession user = users.remove(session.getId());
                if (user != null) {
                    user.release();
                }
                break;
            }
            case "onIceCandidate": {
                JSONObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

                UserSession user = users.get(session.getId());
                if (user != null) {
                    IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate") .
                    getAsString(),
                        jsonCandidate.get("sdpMid").getAsString(),
                        jsonCandidate.get("sdpMLineIndex").getAsString());
                    user.addCandidate(candidate);
                }
                break;
            }
            default:
                sendError(session, "Invalid message with id " + jsonMessage.get("id") .
                getAsString());
                break;
        }
    }

    private void start(final WebSocketSession session, JSONObject jsonMessage) {
        ...
    }

    private void sendError(WebSocketSession session, String message) {
        ...
    }
}

```

Following snippet shows method `start`, where ICE candidates are gathered and Media Pipeline and Media Elements (`WebRtcEndpoint` and `KmsSendData`) are created and connected. Message `startResponse` is sent back to client carrying the SDP answer.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // User session
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).
        ↵useDataChannels()
            .build();
        user.setWebRtcEndpoint(webRtcEndpoint);
        users.put(session.getId(), user);

        // ICE candidates
        webRtcEndpoint.addIceCandidateFoundListener(new EventListener
        ↵<IceCandidateFoundEvent>() {
            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {
                    synchronized (session) {
                        session.sendMessage(new TextMessage(response.toString()));
                    }
                } catch (IOException e) {
                    log.debug(e.getMessage());
                }
            }
        });
    }

    // Media logic
    KmsShowData kmsShowData = new KmsShowData.Builder(pipeline).build();

    webRtcEndpoint.connect(kmsShowData);
    kmsShowData.connect(webRtcEndpoint);

    // SDP negotiation (offer and answer)
    String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
    String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

    JsonObject response = new JsonObject();
    response.addProperty("id", "startResponse");
    response.addProperty("sdpAnswer", sdpAnswer);

    synchronized (session) {
        session.sendMessage(new TextMessage(response.toString()));
    }

    webRtcEndpoint.gatherCandidates();

} catch (Throwable t) {
    sendError(session, t.getMessage());
}
}
}

```

The `sendError` method is quite simple: it sends an `error` message to the client when an exception is caught in the server-side.

```
private void sendError(WebSocketSession session, String message) {
    try {
        JSONObject response = new JSONObject();
        response.addProperty("id", "error");
        response.addProperty("message", message);
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Exception sending message", e);
    }
}
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called `kurento-utils.js` to simplify the WebRTC interaction with the server. This library depends on `adapter.js`, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally `jquery.js` is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/showdatachannel`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `error`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer`.`WebRtcPeerSendrecv` of `kurento-utils.js` is used to start a WebRTC communication.

```
var ws = new WebSocket('wss://' + location.host + '/showdatachannel');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError("Error message from server: " + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (error) {
                    console.error("Error adding candidate: " + error);
                    return;
                }
            });
            break;
        default:
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
    }
}
```

```

        onError('Unrecognized message', parsedMessage);
    }
}

function start() {
    console.log("Starting video call ...")
    // Disable start button
    setState(I_AM_STARTING);
    showSpinner(videoInput, videoOutput);

    var servers = null;
    var configuration = null;
    var peerConnection = new RTCPeerConnection(servers, configuration);

    console.log("Creating channel");
    var dataConstraints = null;

    channel = peerConnection.createDataChannel(getChannelName (), dataConstraints);

    channel.onopen = onSendChannelStateChange;
    channel.onclose = onSendChannelStateChange;

    function onSendChannelStateChange () {
        if(!channel) return;
        var readyState = channel.readyState;
        console.log("sencChannel state changed to " + readyState);
        if(readyState == 'open'){
            dataChannelSend.disabled = false;
            dataChannelSend.focus();
            $('#send').attr('disabled', false);
        } else {
            dataChannelSend.disabled = true;
            $('#send').attr('disabled', true);
        }
    }

    var sendButton = document.getElementById('send');
    var dataChannelSend = document.getElementById('dataChannelSend');

    sendButton.addEventListener("click", function(){
        var data = dataChannelSend.value;
        console.log("Send button pressed. Sending data " + data);
        channel.send(data);
        dataChannelSend.value = "";
    });

    console.log("Creating WebRtcPeer and generating local sdp offer ...");

    var options = {
        peerConnection: peerConnection,
        localVideo : videoInput,
        remoteVideo : videoOutput,
        onicecandidate : onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function(error) {
            if (error) {
                return console.error(error);
            }
        }
    );
}

```

```
        }
        webRtcPeer.generateOffer(onOffer);
    });
}

function closeChannels(){

    if(channel){
        channel.close();
        $('#dataChannelSend').disabled = true;
        $('#send').attr('disabled', true);
        channel = null;
    }
}

function onOffer(error, offerSdp) {
    if (error)
        return console.error("Error generating the offer");
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}

function onError(error) {
    console.error(error);
}

function onIceCandidate(candidate) {
    console.log("Local candidate" + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
    };
    sendMessage(message);
}

function startResponse(message) {
    setState(I_CAN_STOP);
    console.log("SDP answer received from server. Processing ...");

    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function stop() {
    console.log("Stopping video call ...");
    setState(I_CAN_START);
    if (webRtcPeer) {
        closeChannels();

        webRtcPeer.dispose();
        webRtcPeer = null;
    }
}
```

```

    var message = {
      id : 'stop'
    }
    sendMessage(message);
  }
  hideSpinner(videoInput, videoOutput);
}

function sendMessage(message) {
  var jsonMessage = JSON.stringify(message);
  console.log('Senging message: ' + jsonMessage);
  ws.send(jsonMessage);
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
  <dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-client</artifactId>
  </dependency>
  <dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-utils-js</artifactId>
  </dependency>
  <dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>bootstrap</artifactId>
  </dependency>
  <dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>demo-console</artifactId>
  </dependency>
  <dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>adapter.js</artifactId>
  </dependency>
  <dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
  </dependency>
  <dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
  </dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

6.11.2 JavaScript - Hello World with Data Channels

This web application extends the [Hello World Tutorial](#), adding media processing to the basic [WebRTC](#) loopback and allowing send text from browser to the media server through data channels.

For the impatient: running this example

You'll need to install Kurento Media Server before running this example. Read [installation guide](#) for further information.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-hello-world-data-channel
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the parameter `ws_uri` to the URL, as follows:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Note: This demo needs the kms-datachannelexample module installed in the media server. That module is available in the Kurento repositories, so it is possible to install it with:

```
sudo apt-get install kms-datachannelexample
```

Understanding this example

The logic of the application is quite simple: the local stream is sent to Kurento Media Server, which returns it back to the client without modifications. To implement this behavior we need to create a *Media Pipeline* with a single *Media Element*, i.e. of type **WebRtcEndpoint**, which holds the capability of exchanging full-duplex (bidirectional) WebRTC media flows. It is important to set value of property *useDataChannels* to true during **WebRtcEndpoint** creation. This media element is connected to itself in order to deliver back received Media.

The application creates a channel between **PeerConnection** and **WebRtcEndpoint** used for message delivery.

Complete source code of this demo can be found in [GitHub](#).

JavaScript Logic

This demo follows a *Single Page Application* architecture ([SPA](#)). The interface is the following HTML page: [index.html](#). This web page links two Kurento JavaScript libraries:

- **kurento-client.js** : Implementation of the Kurento JavaScript Client.
- **kurento-utils.js** : Kurento utility library aimed to simplify the WebRTC management in the browser.

In addition, these two JavaScript libraries are also required:

- **Bootstrap** : Web framework for developing responsive web sites.
- **jquery.js** : Cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- **adapter.js** : WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **ekko-lightbox** : Module for Bootstrap to open modal images, videos, and galleries.
- **demo-console** : Custom JavaScript console.

The specific logic of this demo is coded in the following JavaScript page: [index.js](#). In this file, there is a function which is called when the green button labeled as *Start* in the GUI is clicked.

```
var startButton = document.getElementById("start");

startButton.addEventListener("click", function() {
    var options = {
        peerConnection: peerConnection,
        localVideo: videoInput,
        remoteVideo: videoOutput
    };

    webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
        if(error) return onError(error)
        this.generateOffer(onOffer)
    });

    [...]
})
```

The function `WebRtcPeer.WebRtcPeerSendrecv` abstracts the WebRTC internal details (i.e. `PeerConnection` and `getUserStream`) and makes possible to start a full-duplex WebRTC communication, using the HTML video tag with id `videoInput` to show the video camera (local stream) and the video tag `videoOutput` to show the remote stream provided by the Kurento Media Server.

Inside this function, a call to `generateOffer` is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the `KurentoClient` class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port 8433 at the same host than the HTTP serving the application.

```
[...]

var args = getopt(location.search,
{
  default:
  {
    ws_uri: 'wss://' + location.hostname + ':8433/kurento',
    ice_servers: undefined
  }
});

[...]

kurentoClient(args.ws_uri, function(error, client) {
  [...]
});
```

Once we have an instance of `kurentoClient`, the following step is to create a *Media Pipeline*, as follows:

```
client.create("MediaPipeline", function(error, _pipeline) {
  [...]
});
```

If everything works correctly, we have an instance of a media pipeline (variable `pipeline` in this example). With this instance, we are able to create *Media Elements*. In this example we just need a `WebRtcEndpoint` with `useDataChannels` property as `true`. Then, this media elements is connected itself:

```
pipeline.create("WebRtcEndpoint", {useDataChannels: true}, function(error, webRtc) {
  if(error) return onError(error);

  setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

  webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {
    if(error) return onError(error);

    webRtc.gatherCandidates(onError);

    webRtcPeer.processAnswer(sdpAnswer, onError);
  });

  webRtc.connect(webRtc, function(error) {
    if(error) return onError(error);

    console.log("Loopback established");
  });
});
```

In the following snippet, we can see how to create the channel and the `send` method of one channel.

```
var dataConstraints = null;
var channel = peerConnection.createDataChannel(getChannelName (), dataConstraints);
...
sendButton.addEventListener("click", function() {
    ...
    channel.send(data);
    ...
});
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},{
    "urls":"stun:stun2.example.net"}]
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",
    "username":"user","credential":"myPassword"}]
```

Dependencies

Demo dependencies are defined in file `bower.json`. They are managed using *Bower*.

```
"dependencies": {
    "kurento-client": "6.7.1",
    "kurento-utils": "6.7.1"
}
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [Bower](#).

6.12 WebRTC recording

This tutorial has two parts:

1. A *WebRTC loopback* records the stream to disk.
2. The stream is played back.

Users can choose which type of media to send and record: audio, video or both.

6.12.1 Java - Recorder

This web application extends *Hello World* adding recording capabilities.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-hello-world-recording
git checkout 6.7.1
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

In the first part of this tutorial, the local stream is sent to the media server, which in turn sends it back to the client, while recording it at the same time. In order to implement this behavior, we need to create a [Media Pipeline](#) consisting on a **WebRtcEndpoint** and a **RecorderEndpoint**.

The second part of this demo shows how to play recorded media. To achieve this, we need to create a [Media Pipeline](#) composed by a **WebRtcEndpoint** and a **PlayerEndpoint**. The `uri` property of the player is the uri of the recorded file.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in depth the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the [Spring Boot](#) framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is [HelloWorldRecApp](#). As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the

application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at *localhost* listening in port 8888. If you reproduce this example you'll need to insert the specific location of your Kurento Media Server instance there.

Once the *Kurento Client* has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@SpringBootApplication
@EnableWebSocket
public class HelloWorldRecApp implements WebSocketConfigurer {

    @Bean
    public HelloWorldRecHandler handler() {
        return new HelloWorldRecHandler();
    }

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create();
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(handler(), "/recording");
    }

    @Bean
    public UserRegistry registry() {
        return new UserRegistry();
    }

    public static void main(String[] args) throws Exception {
        new SpringApplication(HelloWorldRecApp.class).run(args);
    }
}
```

This web application follows a *Single Page Application* architecture (*SPA*), and uses a *WebSocket* to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/recording`.

`HelloWorldRecHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kinds of incoming messages to the *Server*: `start`, `stop`, `play` and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```
public class HelloWorldRecHandler extends TextWebSocketHandler {

    private static final String RECORDER_FILE_PATH = "file:///tmp/HelloWorldRecorded.
    ↪webm";

    private final Logger log = LoggerFactory.getLogger(HelloWorldRecHandler.class);
    private static final Gson gson = new GsonBuilder().create();

    @Autowired
    private UserRegistry registry;
```

```

@Autowired
private KurentoClient kurento;

@Override
public void handleTextMessage(WebSocketSession session, TextMessage message) throws_
Exception {
    JSONObject jsonMessage = gson.fromJson(message.getPayload(), JsonObject.class);

    log.debug("Incoming message: {}", jsonMessage);

    UserSession user = registry.getBySession(session);
    if (user != null) {
        log.debug("Incoming message from user '{}': {}", user.getId(), jsonMessage);
    } else {
        log.debug("Incoming message from new user: {}", jsonMessage);
    }

    switch (jsonMessage.get("id").getAsString()) {
        case "start":
            start(session, jsonMessage);
            break;
        case "stop":
        case "stopPlay":
            if (user != null) {
                user.release();
            }
            break;
        case "play":
            play(user, session, jsonMessage);
            break;
        case "onIceCandidate":
            JsonObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

            if (user != null) {
                IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate") .
getAsString(),
                    jsonCandidate.get("sdpMid").getAsString(),
                    jsonCandidate.get("sdpMLineIndex").getAsInt());
                user.addCandidate(candidate);
            }
            break;
        }
        default:
            sendError(session, "Invalid message with id " + jsonMessage.get("id") .
getAsString());
            break;
    }
}

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    ...
}

private void play(UserSession user, final WebSocketSession session, JsonObject_
jsonMessage) {
    ...
}

```

```

    }

    private void sendError(WebSocketSession session, String message) {
        ...
    }
}

```

In the following snippet, we can see the `start` method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (`WebRtcEndpoint` and `RecorderEndpoint`) and make the connections among them. A `startResponse` message is sent back to the client with the SDP answer.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {

        // 1. Media logic (webRtcEndpoint in loopback)
        MediaPipeline pipeline = kurento.createMediaPipeline();
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
        webRtcEndpoint.connect(webRtcEndpoint);

        MediaProfileSpecType profile = getMediaProfileFromMessage(jsonMessage);

        RecorderEndpoint recorder = new RecorderEndpoint.Builder(pipeline, RECORDER_
        ↪FILE_PATH)
            .withMediaProfile(profile).build();

        connectAccordingToProfile(webRtcEndpoint, recorder, profile);

        // 2. Store user session
        UserSession user = new UserSession(session);
        user.setMediaPipeline(pipeline);
        user.setWebRtcEndpoint(webRtcEndpoint);
        registry.register(user);

        // 3. SDP negotiation
        String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
        String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

        // 4. Gather ICE candidates
        webRtcEndpoint.addIceCandidateFoundListener(new EventListener
        ↪<IceCandidateFoundEvent>() {
            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {
                    synchronized (session) {
                        session.sendMessage(new TextMessage(response.toString()));
                    }
                } catch (IOException e) {
                    log.error(e.getMessage());
                }
            }
        });

        JsonObject response = new JsonObject();
        response.addProperty("id", "startResponse");
        response.addProperty("sdpAnswer", sdpAnswer);
    }
}

```

```

    synchronized (user) {
        session.sendMessage(new TextMessage(response.toString()));
    }

    webRtcEndpoint.gatherCandidates();

    recorder.record();
} catch (Throwable t) {
    log.error("Start error", t);
    sendError(session, t.getMessage());
}
}
}

```

The play method, creates a Media Pipeline with the Media Elements (WebRtcEndpoint and PlayerEndpoint) and make the connections among them. It will then send the recorded media to the client.

```

private void play(UserSession user, final WebSocketSession session, JSONObject
↳jsonMessage) {
    try {

        // 1. Media logic
        final MediaPipeline pipeline = kurento.createMediaPipeline();
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
        PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline, RECORDER_FILE_
↳PATH).build();
        player.connect(webRtcEndpoint);

        // Player listeners
        player.addErrorListener(new EventListener<ErrorEvent>() {
            @Override
            public void onEvent(ErrorEvent event) {
                log.info("ErrorEvent for session '{}': {}", session.getId(), event.
↳getDescription());
                sendPlayEnd(session, pipeline);
            }
        });
        player.addEndOfStreamListener(new EventListener<EndOfStreamEvent>() {
            @Override
            public void onEvent(EndOfStreamEvent event) {
                log.info("EndOfStreamEvent for session '{}'", session.getId());
                sendPlayEnd(session, pipeline);
            }
        });

        // 2. Store user session
        user.setMediaPipeline(pipeline);
        user.setWebRtcEndpoint(webRtcEndpoint);

        // 3. SDP negotiation
        String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
        String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

        JSONObject response = new JSONObject();
        response.addProperty("id", "playResponse");
        response.addProperty("sdpAnswer", sdpAnswer);

        // 4. Gather ICE candidates
    }
}
}

```

```
    webRtcEndpoint.addIceCandidateFoundListener(new EventListener
↳<IceCandidateFoundEvent>() {
    @Override
    public void onEvent(IceCandidateFoundEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
        try {
            synchronized (session) {
                session.sendMessage(new TextMessage(response.toString()));
            }
        } catch (IOException e) {
            log.error(e.getMessage());
        }
    }
});

// 5. Play recorded stream
player.play();

synchronized (session) {
    session.sendMessage(new TextMessage(response.toString()));
}

webRtcEndpoint.gatherCandidates();
} catch (Throwable t) {
    log.error("Play error", t);
    sendError(session, t.getMessage());
}
}
```

```
The sendError method is quite simple: it sends an error message to the client when  
server-side.
```

```
private void sendError(WebSocketSession session, String message) {  
    try {  
        JSONObject response = new JSONObject();  
        response.addProperty("id", "error");  
        response.addProperty("message", message);  
        session.sendMessage(new TextMessage(response.toString()));  
    } catch (IOException e) {  
        log.error("Exception sending message", e);  
    }  
}
```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called **kurento-utils.js** to simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/recording`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming

messages to client: startResponse, playResponse, playEnd, “error”, and iceCandidate. Convenient actions are taken to implement each step in the communication. For example, in functions start the function WebRtcPeer.WebRtcPeerSendrecv of *kurento-utils.js* is used to start a WebRTC communication.

```
var ws = new WebSocket('wss://' + location.host + '/recording');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'playResponse':
            playResponse(parsedMessage);
            break;
        case 'playEnd':
            playEnd();
            break;
        case 'error':
            setState(NO_CALL);
            onError('Error message from server: ' + parsedMessage.message);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
                if (error)
                    return console.error('Error adding candidate: ' + error);
            });
            break;
        default:
            setState(NO_CALL);
            onError('Unrecognized message', parsedMessage);
    }
}

function start() {
    console.log('Starting video call ...');

    // Disable start button
    setState(DISABLED);
    showSpinner(videoInput, videoOutput);
    console.log('Creating WebRtcPeer and generating local sdp offer ...');

    var options = {
        localVideo : videoInput,
        remoteVideo : videoOutput,
        mediaConstraints : getConstraints(),
        onicecandidate : onIceCandidate
    }

    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function(error) {
        if (error)
            return console.error(error);
        webRtcPeer.generateOffer(onOffer);
    });
}
```

```

function onOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp,
        mode : $('input[name="mode"]:checked').val()
    }
    sendMessage(message);
}

function onError(error) {
    console.error(error);
}

function onIceCandidate(candidate) {
    console.log('Local candidate' + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
    };
    sendMessage(message);
}

function startResponse(message) {
    setState(IN_CALL);
    console.log('SDP answer received from server. Processing ...');

    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function stop() {
    var stopMessageId = (state == IN_CALL) ? 'stop' : 'stopPlay';
    console.log('Stopping video while in ' + state + '...');
    setState(POST_CALL);
    if (webRtcPeer) {
        webRtcPeer.dispose();
        webRtcPeer = null;

        var message = {
            id : stopMessageId
        }
        sendMessage(message);
    }
    hideSpinner(videoInput, videoOutput);
}

function play() {
    console.log("Starting to play recorded video...");

    // Disable start button
    setState(DISABLED);
}

```

```

showSpinner(videoOutput);

console.log('Creating WebRtcPeer and generating local sdp offer ...');

var options = {
    remoteVideo : videoOutput,
    mediaConstraints : getConstraints(),
    onicecandidate : onIceCandidate
}

webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
    function(error) {
    if (error)
        return console.error(error);
    webRtcPeer.generateOffer(onPlayOffer);
});

function onPlayOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'play',
        sdpOffer : offerSdp
    }
    sendMessage(message);
}

function getConstraints() {
    var mode = $('input[name="mode"]:checked').val();
    var constraints = {
        audio : true,
        video : true
    }

    if (mode == 'video-only') {
        constraints.audio = false;
    } else if (mode == 'audio-only') {
        constraints.video = false;
    }

    return constraints;
}

function playResponse(message) {
    setState(IN_PLAY);
    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function playEnd() {
    setState(POST_CALL);
    hideSpinner(videoInput, videoOutput);
}

```

```

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('Senging message: ' + jsonMessage);
    ws.send(jsonMessage);
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>jquery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>ekko-lightbox</artifactId>
    </dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

6.12.2 JavaScript - Recorder

This web application extends the [Hello World Tutorial](#), adding recording capabilities.

For the impatient: running this example

You'll need to install Kurento Media Server before running this example. Read [installation guide](#) for further information.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You will need to download the source code from GitHub. There are two implementations of this tutorial, but they are functionally the same. It's just the internal implementation that changes. After checking out the code, you can start the web server.

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-recorder
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-hello-world-recorder-generator
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the parameter `ws_uri` to the URL, as follows:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must be connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

In the first part of this demo, the local stream is sent to Kurento Media Server, which returns it back to the client and records to the same time. In order to implement this behavior we need to create a ‘Media Pipeline’ term: consisting of a **WebRtcEndpoint** and a **RecorderEndpoint**.

The second part of this demo shows how to play recorded media. To achieve this, we need to create a *Media Pipeline* composed by a **WebRtcEndpoint** and a **PlayerEndpoint**. The *uri* property of the player is the uri of the recorded file.

There are two implementations for this demo to be found in github:

- Using callbacks.
- Using `yield`.

Note: The snippets are based in demo with callbacks.

JavaScript Logic

This demo follows a *Single Page Application* architecture (*SPA*). The interface is the following HTML page: `index.html`. This web page links two Kurento JavaScript libraries:

- **kurento-client.js** : Implementation of the Kurento JavaScript Client.
- **kurento-utils.js** : Kurento utility library aimed to simplify the WebRTC management in the browser.

In addition, these two JavaScript libraries are also required:

- **Bootstrap** : Web framework for developing responsive web sites.
- **jquery.js** : Cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- **adapter.js** : WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **ekko-lightbox** : Module for Bootstrap to open modal images, videos, and galleries.
- **demo-console** : Custom JavaScript console.

The specific logic of this demo is coded in the following JavaScript page: `index.js`. In this file, there is a function which is called when the green button, labeled as *Start* in the GUI, is clicked.

```
var startRecordButton = document.getElementById("start");

startRecordButton.addEventListener("click", startRecording);

function startRecording() {
    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput
    };

    webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
        if(error) return onError(error)
        this.generateOffer(onOffer)
    });
}

[...]
```

The function `WebRtcPeer.WebRtcPeerSendrecv` abstracts the WebRTC internal details (i.e. `PeerConnection` and `getUserStream`) and makes possible to start a full-duplex WebRTC communication, using the HTML video tag with id `videoInput` to show the video camera (local stream) and the video tag `videoOutput` to show the remote stream provided by the Kurento Media Server.

Inside this function, a call to `generateOffer` is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the `KurentoClient` class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port 8433 at the same host than the HTTP serving the application.

```
[...]

var args = getopt(location.search,
{
  default:
  {
    ws_uri: 'wss://' + location.hostname + ':8433/kurento',
    file_uri: 'file:///tmp/recorder_demo.webm', // file to be stored in media server
    ice_servers: undefined
  }
});

[...]

kurentoClient(args.ws_uri, function(error, client) {
  [...]
});
```

Once we have an instance of `kurentoClient`, the following step is to create a *Media Pipeline*, as follows:

```
client.create("MediaPipeline", function(error, _pipeline) {
  [...]
});
```

If everything works correctly, we have an instance of a media pipeline (variable `pipeline` in this example). With this instance, we are able to create *Media Elements*. In this example we just need a *WebRtcEndpoint* and a *RecorderEndpoint*. Then, these media elements are interconnected:

```
var elements =
[
  {type: 'RecorderEndpoint', params: {uri : args.file_uri}},
  {type: 'WebRtcEndpoint', params: {}}
]

pipeline.create(elements, function(error, elements) {
  if (error) return onError(error);

  var recorder = elements[0]
  var webRtc = elements[1]

  setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

  webRtc.processOffer(offer, function(error, answer) {
    if (error) return onError(error);

    console.log("offer");

    webRtc.gatherCandidates(onError);
  });
});
```

```

    webRtcPeer.processAnswer(answer);
});

client.connect(webRtc, webRtc, recorder, function(error) {
  if (error) return onError(error);

  console.log("Connected");

  recorder.record(function(error) {
    if (error) return onError(error);

    console.log("record");
  });
});
});
}
);

```

When stop button is clicked, the recoder element stops to record, and all elements are released.

```

stopRecordButton.addEventListener("click", function(event) {
  recorder.stop();
  pipeline.release();
  webRtcPeer.dispose();
  videoInput.src = "";
  videoOutput.src = "";

  hideSpinner(videoInput, videoOutput);

  var playButton = document.getElementById('play');
  playButton.addEventListener('click', startPlaying);
})
}

```

In the second part, after play button is clicked, we have an instance of a media pipeline (variable `pipeline` in this example). With this instance, we are able to create *Media Elements*. In this example we just need a *WebRtcEndpoint* and a *PlayerEndpoint* with *uri* option like path where the media was recorded. Then, these media elements are interconnected:

```

var options = {uri : args.file_uri}

pipeline.create("PlayerEndpoint", options, function(error, player) {
  if (error) return onError(error);

  player.on('EndOfStream', function(event) {
    pipeline.release();
    videoPlayer.src = "";

    hideSpinner(videoPlayer);
  });

  player.connect(webRtc, function(error) {
    if (error) return onError(error);

    player.play(function(error) {
      if (error) return onError(error);
      console.log("Playing ...");
    });
  });
})
}

```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},{  
  "urls":"stun:stun2.example.net"}]  
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",  
  "username":"user","credential":"myPassword"}]
```

Dependencies

Demo dependencies are located in file `bower.json`. *Bower* is used to collect them.

```
"dependencies": {  
  "kurento-client": "6.7.1",  
  "kurento-utils": "6.7.1"  
}
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [Bower](#).

6.13 WebRTC repository

This is similar to the recording tutorial, but using the repository to store metadata.

6.13.1 Java - Repository

This web application extends *Hello World* adding recording capabilities by means of the Kurento Repository.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

You need to have installed the Kurento Media Server before running this example. Read the [installation guide](#) for further information.

In addition, you also need the **kurento-repository-server**. This component is in charge of the storage and retrieval of the media. Please visit the [Kurento Repository Server installation guide](#) for further details.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git  
cd kurento-tutorial-java/kurento-hello-world-repository/  
git checkout 6.7.1  
mvn compile exec:java
```

Access the application connecting to the URL <https://localhost:8443/> in a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. In addition, by default this demo is also supposing that the Kurento Repository is up and running in the localhost. It can be changed by means of the property `repository.uri`. All in all, and due to the fact that we can use Maven to run the tutorial, you should execute the following command:

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento \
-Drepository.uri=http://repository_host:repository_url
```

Understanding this example

On top of the recording capabilities from the base tutorial, this application creates a repository element to store media in that repository. Additionally, metadata about the recorded file can be also stored in the repository.

This is a web application, and therefore it follows a client-server architecture. At the client-side, the logic is implemented in **JavaScript**. At the server-side, we use a Spring-Boot based server application consuming the **Kurento Java Client API**, to control **Kurento Media Server** capabilities. All in all, the high level architecture of this demo is three-tier. To communicate these entities, two WebSockets are used. First, a WebSocket is created between client and application server to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication takes place using the **Kurento Protocol**. For further information on it, please see this [page](#) of the documentation.

The following sections analyze in deep the server (Java) and client-side (JavaScript) code of this application. The complete source code can be found in [GitHub](#).

Application Server Logic

This demo has been developed using **Java** in the server-side, based on the **Spring Boot** framework, which embeds a Tomcat web server within the generated maven artifact, and thus simplifies the development and deployment process.

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. Here we chose Spring Boot for convenience.

The main class of this demo is `HelloWorldRecApp`. As you can see, the `KurentoClient` is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at `localhost` listening in port 8888. If you reproduce this example you'll need to insert the specific location of your Kurento Media Server instance there.

Once the `Kurento Client` has been instantiated, you are ready for communicating with Kurento Media Server and controlling its multimedia capabilities.

```
@SpringBootApplication
@EnableWebSocket
public class HelloWorldRecApp implements WebSocketConfigurer {

    protected static final String DEFAULT_REPOSITORY_SERVER_URI = "http://localhost:7676";
}
```

```

protected static final String REPOSITORY_SERVER_URI =
    System.getProperty("repository.uri", DEFAULT_REPOSITORY_SERVER_URI);

@Bean
public HelloWorldRecHandler handler() {
    return new HelloWorldRecHandler();
}

@Bean
public KurentoClient kurentoClient() {
    return KurentoClient.create();
}

@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(handler(), "/repository");
}

@Bean
public RepositoryClient repositoryServiceProvider() {
    return REPOSITORY_SERVER_URI.startsWith("file://") ? null
        : RepositoryClientProvider.create(REPOSITORY_SERVER_URI);
}

@Bean
public UserRegistry registry() {
    return new UserRegistry();
}

public static void main(String[] args) throws Exception {
    new SpringApplication(HelloWorldRecApp.class).run(args);
}
}
}

```

This web application follows a *Single Page Application* architecture ([SPA](#)), and uses a [WebSocket](#) to communicate client with application server by means of requests and responses. Specifically, the main app class implements the interface `WebSocketConfigurer` to register a `WebSocketHandler` to process `WebSocket` requests in the path `/repository`.

`HelloWorldRecHandler` class implements `TextWebSocketHandler` to handle text `WebSocket` requests. The central piece of this class is the method `handleTextMessage`. This method implements the actions for requests, returning responses through the `WebSocket`. In other words, it implements the server part of the signaling protocol depicted in the previous sequence diagram.

In the designed protocol there are three different kinds of incoming messages to the *Server* : `start`, `stop`, `stopPlay`, `play` and `onIceCandidates`. These messages are treated in the `switch` clause, taking the proper steps in each case.

```

public class HelloWorldRecHandler extends TextWebSocketHandler {

    // slightly larger timeout
    private static final int REPOSITORY_DISCONNECT_TIMEOUT = 5500;

    private static final String RECORDING_EXT = ".webm";

    private final Logger log = LoggerFactory.getLogger(HelloWorldRecHandler.class);
    private static final SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd_HH-mm-
→ss-S");
}

```

```

private static final Gson gson = new GsonBuilder().create();

@Autowired
private UserRegistry registry;

@Autowired
private KurentoClient kurento;

@Autowired
private RepositoryClient repositoryClient;

@Override
public void handleTextMessage(WebSocketSession session, TextMessage message) throws
Exception {
    JsonObject jsonMessage = gson.fromJson(message.getPayload(), JsonObject.class);

    log.debug("Incoming message: {}", jsonMessage);

    UserSession user = registry.getBySession(session);
    if (user != null) {
        log.debug("Incoming message from user '{}': {}", user.getId(), jsonMessage);
    } else {
        log.debug("Incoming message from new user: {}", jsonMessage);
    }

    switch (jsonMessage.get("id").getAsString()) {
        case "start":
            start(session, jsonMessage);
            break;
        case "stop":
        case "stopPlay":
            if (user != null) {
                user.release();
            }
            break;
        case "play":
            play(user, session, jsonMessage);
            break;
        case "onIceCandidate":
            JsonObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

            if (user != null) {
                IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate") .
                    getAsString(),
                    jsonCandidate.get("sdpMid").getAsString(),
                    jsonCandidate.get("sdpMLineIndex").getAsInt());
                user.addCandidate(candidate);
            }
            break;
    }
    default:
        sendError(session, "Invalid message with id " + jsonMessage.get("id") .
            getAsString());
        break;
    }
}

private void start(final WebSocketSession session, JsonObject jsonMessage) {

```

```

    ...
}

private void play(UserSession user, final WebSocketSession session, JsonObject jsonMessage) {
    ...
}

private void sendError(WebSocketSession session, String message) {
    ...
}
}

```

In the following snippet, we can see the `start` method. If a repository REST client or interface has been created, it will obtain a `RepositoryItem` from the remote service. This item contains an ID and a recording URI that will be used by the Kurento Media Server. The ID will be used after the recording ends in order to manage the stored media. If the client doesn't exist, the recording will be performed to a local URI, on the same machine as the KMS. This method also deals with the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (`WebRtcEndpoint` and `RecorderEndpoint`) and makes the connections between them. A `startResponse` message is sent back to the client with the SDP answer.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // 0. Repository logic
        RepositoryItemRecorder repoItem = null;
        if (repositoryClient != null) {
            try {
                Map<String, String> metadata = Collections.emptyMap();
                repoItem = repositoryClient.createRepositoryItem(metadata);
            } catch (Exception e) {
                log.warn("Unable to create kurento repository items", e);
            }
        } else {
            String now = df.format(new Date());
            String filePath = HelloWorldRecApp.REPOSITORY_SERVER_URI + now + RECORDING_
EXT;
            repoItem = new RepositoryItemRecorder();
            repoItem.setId(now);
            repoItem.setUrl(filePath);
        }
        log.info("Media will be recorded {}by KMS: id={} , url={}",
            (repositoryClient == null ? "locally" : ""), repoItem.getId(), repoItem.
getUrl());
    }

    // 1. Media logic (webRtcEndpoint in loopback)
    MediaPipeline pipeline = kurento.createMediaPipeline();
    WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
    webRtcEndpoint.connect(webRtcEndpoint);
    RecorderEndpoint recorder = new RecorderEndpoint.Builder(pipeline, repoItem.
getUrl())
        .withMediaProfile(MediaProfileSpecType.WEBM).build();
    webRtcEndpoint.connect(recorder);

    // 2. Store user session
    UserSession user = new UserSession(session);
    user.setMediaPipeline(pipeline);
    user.setWebRtcEndpoint(webRtcEndpoint);
}

```

```
user.setRepoItem(repoItem);
registry.register(user);

// 3. SDP negotiation
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

// 4. Gather ICE candidates
webRtcEndpoint.addIceCandidateFoundListener(new EventListener
→<IceCandidateFoundEvent>() {
    @Override
    public void onEvent(IceCandidateFoundEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
        try {
            synchronized (session) {
                session.sendMessage(new TextMessage(response.toString()));
            }
        } catch (IOException e) {
            log.error(e.getMessage());
        }
    }
}
```

The `play` method, creates a Media Pipeline with the Media Elements (`WebRtcEndpoint` and `PlayerEndpoint`) and make the connections between them. It will then send the recorded media to the client. The media can be served from the repository or directly from the disk. If the repository interface exists, it will try to connect to the remote service in order to obtain an URI from which the KMS will read the media streams. The inner workings of the repository restrict reading an item before it has been closed (after the upload finished). This will happen only when a certain number of seconds elapse after the last byte of media is uploaded by the KMS (safe-guard for gaps in the network communications).

```
private void play(UserSession user, final WebSocketSession session, JSONObject jsonMessage) {
    try {
        // 0. Repository logic
        RepositoryItemPlayer itemPlayer = null;
        if (repositoryClient != null) {
            try {
                Date stopTimestamp = user.getStopTimestamp();
                if (stopTimestamp != null) {
                    Date now = new Date();
                    long diff = now.getTime() - stopTimestamp.getTime();
                    if (diff >= 0 && diff < REPOSITORY_DISCONNECT_TIMEOUT) {
                        log.info(
                            "Waiting for {}ms before requesting the repository read endpoint "
                            + "(requires {}ms before upload is considered terminated "
                            + "and only {}ms have passed)",
                            REPOSITORY_DISCONNECT_TIMEOUT - diff, REPOSITORY_DISCONNECT_TIMEOUT,
                            diff);
                        Thread.sleep(REPOSITORY_DISCONNECT_TIMEOUT - diff);
                    }
                } else {
                    log.warn("No stop timeout was found, repository endpoint might not be ready");
                }
            }
        }
    }
}
```

```

        itemPlayer = repositoryClient.getReadEndpoint(user.getRepoItem().getId());
    } catch (Exception e) {
        log.warn("Unable to obtain kurento repository endpoint", e);
    }
} else {
    itemPlayer = new RepositoryItemPlayer();
    itemPlayer.setId(user.getRepoItem().getId());
    itemPlayer.setUrl(user.getRepoItem().getUrl());
}
log.debug("Playing from {}: id={}, url={}",
    (repositoryClient == null ? "disk" : "repository"), itemPlayer.getId(),
    itemPlayer.getUrl());

// 1. Media logic
final MediaPipeline pipeline = kurento.createMediaPipeline();
WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline, itemPlayer.
→getUrl()).build();
player.connect(webRtcEndpoint);

// Player listeners
player.addErrorListener(new EventListener<ErrorEvent>() {
    @Override
    public void onEvent(ErrorEvent event) {
        log.info("ErrorEvent for session '{}': {}", session.getId(), event.
→getDescription());
        sendPlayEnd(session, pipeline);
    }
});
player.addEndOfStreamListener(new EventListener<EndOfStreamEvent>() {
    @Override
    public void onEvent(EndOfStreamEvent event) {
        log.info("EndOfStreamEvent for session '{}'", session.getId());
        sendPlayEnd(session, pipeline);
    }
});

// 2. Store user session
user.setMediaPipeline(pipeline);
user.setWebRtcEndpoint(webRtcEndpoint);

// 3. SDP negotiation
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

JsonObject response = new JsonObject();
response.addProperty("id", "playResponse");
response.addProperty("sdpAnswer", sdpAnswer);

// 4. Gather ICE candidates
webRtcEndpoint.addIceCandidateFoundListener(new EventListener
→<IceCandidateFoundEvent>() {
    @Override
    public void onEvent(IceCandidateFoundEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
        try {

```

```

        synchronized (session) {
            session.sendMessage(new TextMessage(response.toString()));
        }
    } catch (IOException e) {
        log.error(e.getMessage());
    }
}
);

```

The `sendError` method is quite simple: it sends an `error` message to the client when an exception is caught in the server-side.

```

private void sendError(WebSocketSession session, String message) {
    try {
        JSONObject response = new JSONObject();
        response.addProperty("id", "error");
        response.addProperty("message", message);
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Exception sending message", e);
    }
}

```

Client-Side Logic

Let's move now to the client-side of the application. To call the previously created WebSocket service in the server-side, we use the JavaScript class `WebSocket`. We use a specific Kurento JavaScript library called **kurento-utils.js** to simplify the WebRTC interaction with the server. This library depends on **adapter.js**, which is a JavaScript WebRTC utility maintained by Google that abstracts away browser differences. Finally **jquery.js** is also needed in this application.

These libraries are linked in the `index.html` web page, and are used in the `index.js`. In the following snippet we can see the creation of the WebSocket (variable `ws`) in the path `/repository`. Then, the `onmessage` listener of the WebSocket is used to implement the JSON signaling protocol in the client-side. Notice that there are three incoming messages to client: `startResponse`, `playResponse`, `playEnd`, `"error"`, and `iceCandidate`. Convenient actions are taken to implement each step in the communication. For example, in functions `start` the function `WebRtcPeer.WebRtcPeerSendrecv` of *kurento-utils.js* is used to start a WebRTC communication.

```

var ws = new WebSocket('wss://' + location.host + '/repository');

ws.onmessage = function(message) {
    var parsedMessage = JSON.parse(message.data);
    console.info('Received message: ' + message.data);

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'playResponse':
            playResponse(parsedMessage);
            break;
        case 'playEnd':
            playEnd();
            break;
        case 'error':
            setState(NO_CALL);
    }
}

```

```

        onError('Error message from server: ' + parsedMessage.message);
        break;
    case 'iceCandidate':
        webRtcPeer.addIceCandidate(parsedMessage.candidate, function(error) {
            if (error)
                return console.error('Error adding candidate: ' + error);
        });
        break;
    default:
        setState(NO_CALL);
        onError('Unrecognized message', parsedMessage);
    }
}

function start() {
console.log('Starting video call ...');

// Disable start button
setState(DISABLED);
showSpinner(videoInput, videoOutput);

console.log('Creating WebRtcPeer and generating local sdp offer ...');

var options = {
    localVideo : videoInput,
    remoteVideo : videoOutput,
    onicecandidate : onIceCandidate
}
webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
    function(error) {
        if (error)
            return console.error(error);
        webRtcPeer.generateOffer(onOffer);
    });
}

function onOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp,
        mode : $('input[name="mode"]:checked').val()
    }
    sendMessage(message);
}

function onError(error) {
    console.error(error);
}

function onIceCandidate(candidate) {
    console.log('Local candidate' + JSON.stringify(candidate));

    var message = {
        id : 'onIceCandidate',
        candidate : candidate
}

```

```

    };
    sendMessage(message);
}

function startResponse(message) {
    setState(IN_CALL);
    console.log('SDP answer received from server. Processing ...');

    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function stop() {
    var stopMessageId = (state == IN_CALL) ? 'stop' : 'stopPlay';
    console.log('Stopping video while in ' + state + '...');
    setState(POST_CALL);
    if (webRtcPeer) {
        webRtcPeer.dispose();
        webRtcPeer = null;

        var message = {
            id : stopMessageId
        }
        sendMessage(message);
    }
    hideSpinner(videoInput, videoOutput);
}

function play() {
    console.log("Starting to play recorded video...");

    // Disable start button
    setState(DISABLED);
    showSpinner(videoOutput);

    console.log('Creating WebRtcPeer and generating local sdp offer ...');

    var options = {
        remoteVideo : videoOutput,
        onicecandidate : onIceCandidate
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
        function(error) {
            if (error)
                return console.error(error);
            webRtcPeer.generateOffer(onPlayOffer);
        });
}

function onPlayOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'play',
        sdpOffer : offerSdp
}

```

```
        }
        sendMessage(message);
    }

function playResponse(message) {
    setState(IN_PLAY);
    webRtcPeer.processAnswer(message.sdpAnswer, function(error) {
        if (error)
            return console.error(error);
    });
}

function playEnd() {
    setState(POST_CALL);
    hideSpinner(videoInput, videoOutput);
}

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('Senging message: ' + jsonMessage);
    ws.send(jsonMessage);
}
```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side. Other client libraries are managed with `webjars`:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
```

```

<groupId>org.webjars.bower</groupId>
<artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```

<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>

```

6.14 WebRTC statistics

This tutorial implements a *WebRTC loopback* and shows how to collect WebRTC statistics.

6.14.1 JavaScript - Loopback stats

This web application extends *the Hello World tutorial* showing how statistics are collected.

For the impatient: running this example

You'll need to install Kurento Media Server before running this example. Read *installation guide* for further information.

Be sure to have installed *Node.js* and *Bower* in your system. In an Ubuntu machine, you can install both as follows:

```

curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower

```

Due to *Same-origin policy*, this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using *npm*:

```
sudo npm install http-server -g
```

Clone source code from GitHub and then start the HTTP server:

```

git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-loopback-stats
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key

```

Connect to URL <https://localhost:8443/> using a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the parameter `ws_uri` to the URL, as follows:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

The logic of the application is quite simple: the local stream is sent to the Kurento Media Server, which returns it back to the client without modifications. To implement this behavior we need to create a [*Media Pipeline*](#) composed by the [*Media Element*](#) **WebRtcEndpoint**, which holds the capability of exchanging full-duplex (bidirectional) WebRTC media flows. This media element is connected to itself so any received media (from browser) is send back (to browser). Using method `getStats` the application shows all stats of element **WebRtcEndpoint**.

The complete source code of this demo can be found in [GitHub](#).

JavaScript Logic

This demo follows a *Single Page Application* architecture ([*SPA*](#)). The interface is the following HTML page: `index.html`. This web page links two Kurento JavaScript libraries:

- **kurento-client.js** : Implementation of the Kurento JavaScript Client.
- **kurento-utils.js** : Kurento utility library aimed to simplify the WebRTC management in the browser.

In addition, these two JavaScript libraries are also required:

- **Bootstrap** : Web framework for developing responsive web sites.
- **jquery.js** : Cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- **adapter.js** : WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **ekko-lightbox** : Module for Bootstrap to open modal images, videos, and galleries.
- **demo-console** : Custom JavaScript console.

The specific logic of this demo is coded in the following JavaScript page: `index.js`. In this file, there is a function which is called when the green button labeled as *Start* in the GUI is clicked.

```
var startButton = document.getElementById("start");

startButton.addEventListener("click", function() {
    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput
    };

    webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
        if(error) return onError(error)
        this.generateOffer(onOffer)
    });
});
```

```
[...]
}
```

The function `WebRtcPeer.WebRtcPeerSendrecv` hides internal details (i.e. `PeerConnection` and `getUserStream`) and makes possible to start a full-duplex WebRTC communication, using the HTML video tag with id `videoInput` to show the video camera (local stream) and the video tag `videoOutput` to show the remote stream provided by the Kurento Media Server.

Inside this function, a call to `generateOffer` is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the `KurentoClient` class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port 8433 at the same host than the HTTP serving the application.

```
[...]

var args = getopt(location.search,
{
  default:
  {
    ws_uri: 'wss://' + location.hostname + ':8433/kurento',
    ice_servers: undefined
  }
});

[...]

kurentoClient(args.ws_uri, function(error, client) {
  [...]
});
```

Once we have an instance of `kurentoClient`, the following step is to create a *Media Pipeline*, as follows:

```
client.create("MediaPipeline", function(error, _pipeline) {
  [...]
});
```

If everything works correctly, we have an instance of a media pipeline (variable `pipeline` in this example). With this instance, we are able to create *Media Elements*. In this example we just need a `WebRtcEndpoint`. Then, this media elements is connected itself:

```
pipeline.create("WebRtcEndpoint", function(error, webRtc) {
  if (error) return onError(error);

  webRtcEndpoint = webRtc;

  setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

  webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {
    if (error) return onError(error);

    webRtc.gatherCandidates(onError);

    webRtcPeer.processAnswer(sdpAnswer, onError);
  });

  webRtc.connect(webRtc, function(error) {
    if (error) return onError(error);
```

```

        console.log("Loopback established");

        webRtcEndpoint.on('MediaStateChanged', function(event) {
            if (event.newState == "CONNECTED") {
                console.log("MediaState is CONNECTED ... printing stats...")
                activateStatsTimeout();
            }
        });
    });
});
```

In the following snippet, we can see `getStats` method. This method returns several statistic values of **WebRtcEndpoint**.

```

function getBrowserOutgoingVideoStats(webRtcPeer, callback) {
    var peerConnection = webRtcPeer.peerConnection;

    peerConnection.getStats(function(stats) {
        var results = stats.result();

        for (var i = 0; i < results.length; i++) {
            var res = results[i];
            if (res.type != 'ssrc') continue;

            //Publish it to be compliant with W3C stats draft
            var retVal = {
                timeStamp: res.timestamp,
                //StreamStats below
                associateStatsId: res.id,
                codecId: "--",
                firCount: res.stat('googFirsReceived'),
                isRemote: false,
                mediaTrackId: res.stat('googTrackId'),
                nackCount: res.stat('googNacksReceived'),
                pliCount: res.stat('googPlisReceived'),
                sliCount: 0,
                ssrc: res.stat('ssrc'),
                transportId: res.stat('transportId'),
                //Specific outbound below
                bytesSent: res.stat('bytesSent'),
                packetsSent: res.stat('packetsSent'),
                roundTripTime: res.stat('googRtt'),
                packetsLost: res.stat('packetsLost'),
                targetBitrate: "?",
                remb: "?"
            }
            return callback(null, retVal);
        }
        return callback("Error: could not find ssrc type on track stats", null);
    }, localVideoTrack);
}
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"}, {"urls":"stun:stun2.example.net"}]
```

```
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",
  ↵"username":"user","credential":"myPassword"}]
```

Dependencies

Demo dependencies are located in file `bower.json`. *Bower* is used to collect them.

```
"dependencies": {
  "kurento-client": "6.7.1",
  "kurento-utils": "6.7.1"
}
```

Note: We are in active development. You can find the latest version of Kurento JavaScript Client at [Bower](#).

Features

This page summarizes the features that Kurento provides, with links to their own documentation page for the most important ones.

7.1 Kurento API, Clients, and Protocol

Kurento Media Server exposes all its functionality through an RPC API called [Kurento API](#). This API can be queried directly by any kind of JSON-compatible client, but the recommended way to work with it is by means of a [Kurento Client](#) library; these are currently provided for *Java*, *Browser Javascript*, and *Node.js*.

If you prefer a different programming language, it's possible to write a custom client library by following the specification of the [Kurento Protocol](#), based on *WebSocket* and *JSON-RPC*.

The picture below shows how to use Kurento Clients in three scenarios:

- Using the Kurento JavaScript Client directly in a compliant WebRTC browser.
- Using the Kurento Java Client in a Java EE Application Server.
- Using the Kurento JavaScript Client in a Node.js server.

Complete examples for these three technologies is described in the [Tutorials section](#).

The Kurento Client API is based on the concept of **Media Elements**. A Media Element holds a specific media capability. For example, the media element called *WebRtcEndpoint* holds the capability of sending and receiving WebRTC media streams; the media element called *RecorderEndpoint* has the capability of recording into the file system any media streams it receives; the *FaceOverlayFilter* detects faces on the exchanged video streams and adds a specific overlaid image on top of them, etc. Kurento exposes a rich toolbox of media elements as part of its APIs.

To better understand these concepts it is recommended to take a look to the sections [Kurento API](#) and [Kurento Protocol](#). You can also take a look at the Reference Documentation of the API implementations that are currently provided: [Kurento Client](#).

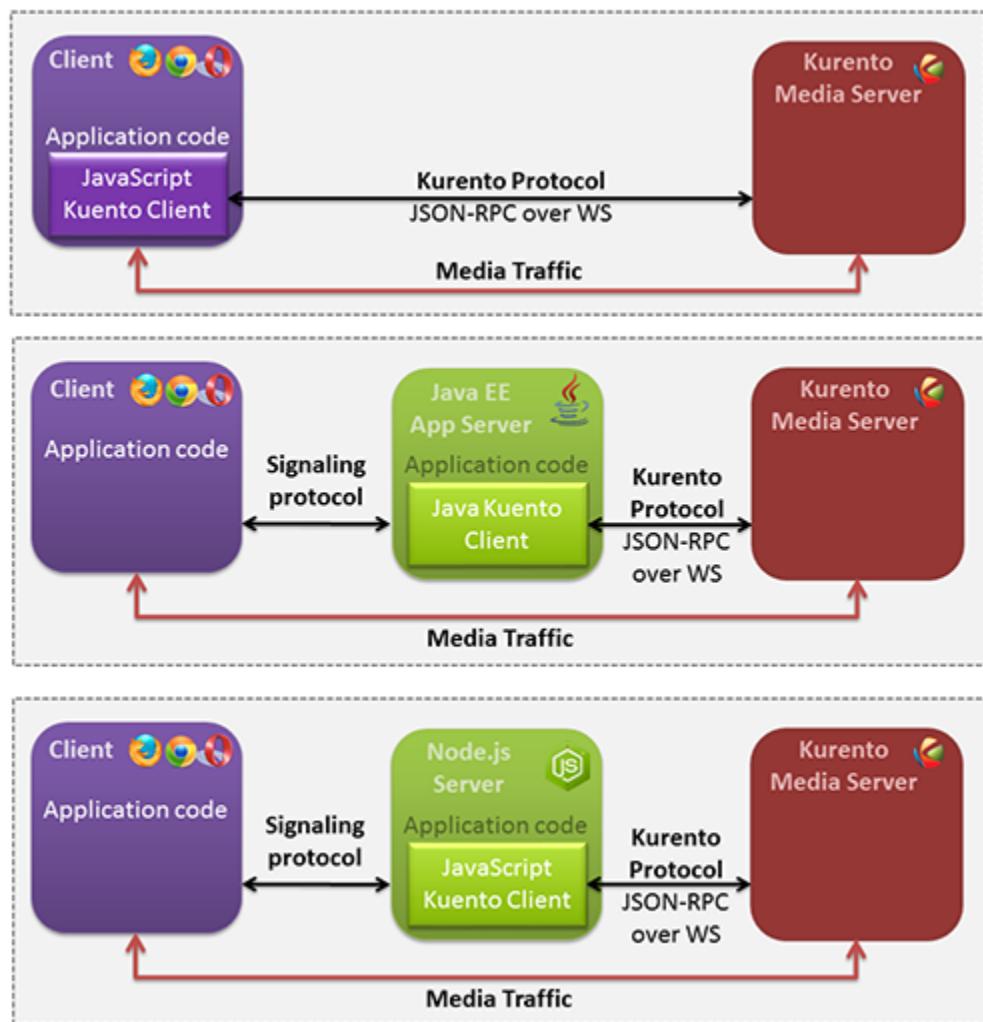


Fig. 7.1: Connection of Kurento Clients (Java and JavaScript) to Kurento Media Server

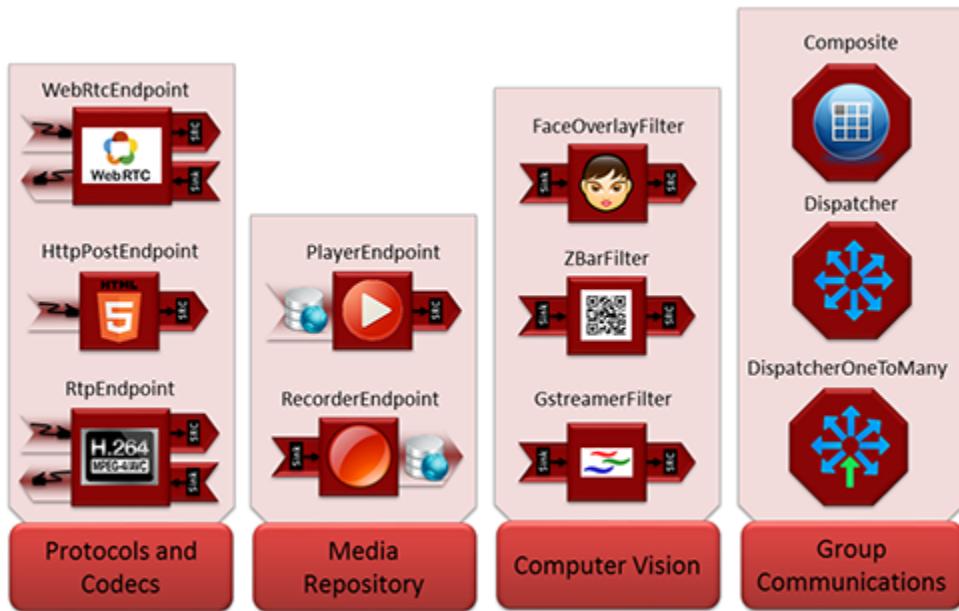


Fig. 7.2: Some Media Elements provided out of the box by Kurento

7.2 Kurento Modules

Kurento has been designed as a pluggable framework. Kurento Media Server uses several modules by default, named *kms-core*, *kms-elements* and *kms-filters*.

In addition, there are others built-in modules to enhance the capabilities provided by the Kurento Media Server. These modules are named *kms-crowddetector*, *kms-pointerdetector*, *kms-chroma*, and *kms-platedetector*.

Finally, Kurento Media Server can be expanded with new custom modules.

For more information, read the section [Kurento Modules](#).

7.3 RTP Streaming

Besides WebRTC connections, Kurento Media Server is able to manage standard RTP streams, allowing to connect an instance of KMS to a wide variety of devices.

There are two topics to note when dealing with RTP connections: the automatic congestion control algorithms that KMS implements (see [Congestion Control \(REMB\)](#)), and the NAT traversal capabilities (see [NAT Traversal](#)).

7.4 Congestion Control (REMB)

Kurento implements the *Google Congestion Control* algorithm, so it is able to generate and parse both `abs-send-time` RTP headers and *REMB* RTCP messages.

It is enabled by passing the media-level attribute `goog-remb` in the SDP Offer. For example:

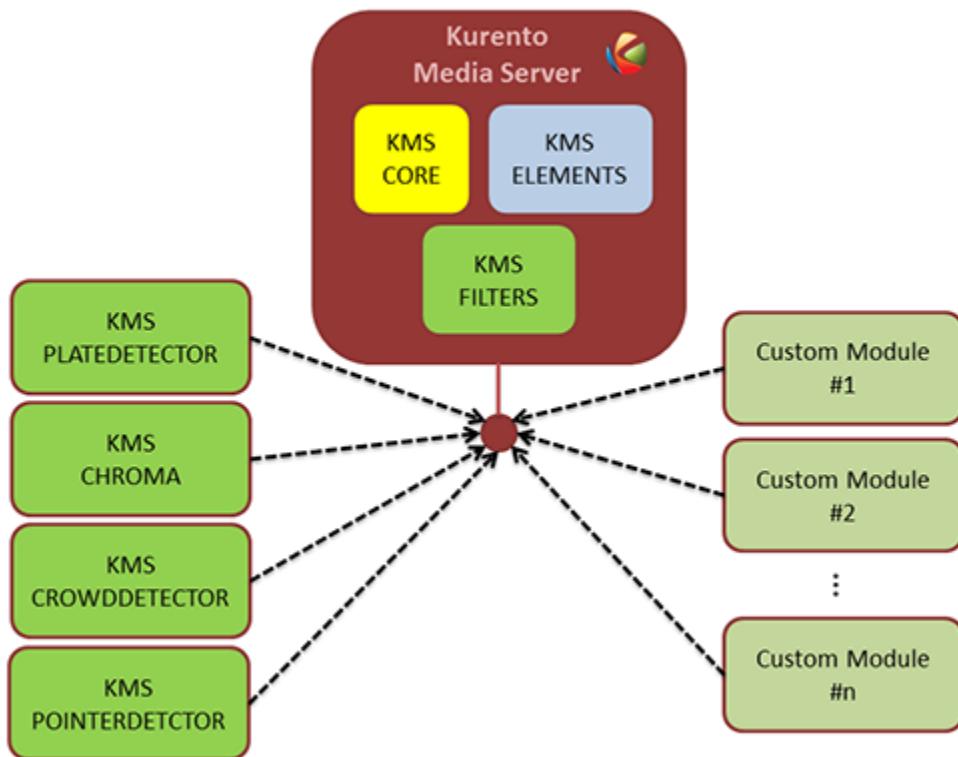


Fig. 7.3: *Kurento modules architecture. Kurento Media Server can be extended with built-it modules (crowddetector, pointerdetector, chroma, platedetector) and also with other custom modules.*

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
m=video 5004 RTP/AVPF 103
a=rtpmap:103 H264/90000
a=rtcp-fb:103 goog-remb
a=sendonly
a=ssrc:112233 cname:user@example.com
```

a=rtcp-fb is an *RTCP Feedback Capability Attribute*, as defined in [RFC 4585](#).

Also it is important to note that KMS implements REMB propagation between the sender and receiver legs of a connection. This means that when KMS is used as a proxy between a video sender and one or more video receivers, the smallest REMB value from the receivers will be relayed to the sender. This allows the sender to choose a lower bitrate that will accommodate all of the receivers connected to KMS at the other side.

For more context about what is REMB and how it fits in the greater project of RMCAT, please read our Knowledge Base document: [Congestion Control \(RMCAT\)](#).

CHAPTER 8

Configuration Guide

Kurento works by orchestrating a broad set of technologies that must be made to work together. Some of these technologies can accept different configuration parameters that Kurento makes available through several configuration files:

- `/etc/kurento/kurento.conf.json`: The main configuration file. Provides settings for the behavior of Kurento Media Server itself.
- `/etc/kurento/modules/kurento/MediaElement.conf.ini`: Generic parameters for all kinds of *MediaElement*.
- `/etc/kurento/modules/kurento/SdpEndpoint.conf.ini`: Audio/video parameters for *SdpEndpoint*'s (*i.e.* `*WebRtcEndpoint` and `RtpEndpoint`).
- `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`: Specific parameters for `WebRtcEndpoint`.
- `/etc/kurento/modules/kurento/HttpEndpoint.conf.ini`: Specific parameters for `HttpEndpoint`.
- `/etc/default/kurento-media-server`: This file is loaded by the system's service init files. Defines some environment variables, which have an effect on features such as the *Debug Logging*, or the *Kernel Dump* files that are generated when a crash happens.

8.1 Media Server

File: `/etc/kurento/kurento.conf.json`.

[TODO] Explain parameters.

8.2 MediaElement

File: `/etc/kurento/modules/kurento/MediaElement.conf.ini`.

[TODO] Explain parameters.

8.3 SdpEndpoint

File: /etc/kurento/modules/kurento/SdpEndpoint.conf.ini.

[TODO] Explain parameters.

8.4 WebRtcEndpoint

File: /etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini.

[TODO] Explain parameters.

8.5 HttpEndpoint

File: /etc/kurento/modules/kurento/HttpEndpoint.conf.ini.

[TODO] Explain parameters.

8.6 Debug Logging

File: /etc/default/kurento-media-server.

CHAPTER 9

Writing Kurento Applications

Table of Contents

- *Writing Kurento Applications*
 - *Global Architecture*
 - *Application Architecture*
 - * *Communicating client, server and Kurento*
 - *1. Media negotiation phase (signaling)*
 - *2. Media exchange phase*
 - * *Real time WebRTC applications with Kurento*
 - *Media Plane*

9.1 Global Architecture

Kurento can be used following the architectural principles of the web. That is, creating a multimedia application based on Kurento can be a similar experience to creating a web application using any of the popular web development frameworks.

At the highest abstraction level, web applications have an architecture comprised of three different layers:

- **Presentation layer (client side):** Here we can find all the application code which is in charge of interacting with end users so that information is represented in a comprehensive way. This usually consists on HTML pages.
- **Application logic (server side):** This layer is in charge of implementing the specific functions executed by the application.
- **Service layer (server or Internet side):** This layer provides capabilities used by the application logic such as databases, communications, security, etc. These services can be hosted in the same server as the application

logic, or can be provided by external parties.

Following this parallelism, multimedia applications created using Kurento can also be implemented with the same architecture:

- **Presentation layer (client side):** Is in charge of multimedia representation and multimedia capture. It is usually based on specific built-in capabilities of the client. For example, when creating a browser-based application, the presentation layer will use capabilities such as the <video> HTML tag or the [WebRTC](#) JavaScript APIs.
- **Application logic:** This layer provides the specific multimedia logic. In other words, this layer is in charge of building the appropriate pipeline (by chaining the desired Media Elements) that the multimedia flows involved in the application will need to traverse.
- **Service layer:** This layer provides the multimedia services that support the application logic such as media recording, media ciphering, etc. The Kurento Media Server (i.e. the specific [Media Pipeline](#) of [Media Elements](#)) is in charge of this layer.

The interesting aspect of this discussion is that, as happens with web development, Kurento applications can place the Presentation layer at the client side and the Service layer at the server side. However the Application logic, in both cases, can be located at either of the sides or even distributed between them. This idea is represented in the following picture:

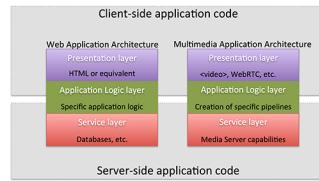


Fig. 9.1: *Layered architecture of web and multimedia applications. Applications created using Kurento (right) can be similar to standard Web applications (left). Both types of applications may choose to place the application logic at the client or at the server code.*

This means that Kurento developers can choose to include the code creating the specific media pipeline required by their applications at the client side (using a suitable [Kurento Client](#) or directly with [Kurento Protocol](#)) or can place it at the server side.

Both options are valid but each of them implies different development styles. Having said this, it is important to note that in the web developers usually tend to maintain client side code as simple as possible, bringing most of their application logic to the server. Reproducing this kind of development experience is the most usual way of using Kurento.

Note: In the following sections it is considered that all Kurento handling is done at the server side. Although this is the most common way of using Kurento, is important to note that all multimedia logic can be implemented at the client with the [Kurento JavaScript Client](#).

9.2 Application Architecture

Kurento, as most multimedia communication technologies out there, is built using two layers (called *planes*) to abstract key functions in all interactive communication systems:

- **Signaling Plane.** The parts of the system in charge of the management of communications, that is, the modules that provides functions for media negotiation, QoS parametrization, call establishment, user registration, user presence, etc. are conceived as forming part of the [Signaling Plane](#).

- **Media Plane.** Functionalities such as media transport, media encoding/decoding and media processing make the *Media Plane*, which takes care of handling the media. The distinction comes from the telephony differentiation between the handling of voice and the handling of meta-information such as tone, billing, etc.

The following figure shows a conceptual representation of the high level architecture of Kurento:

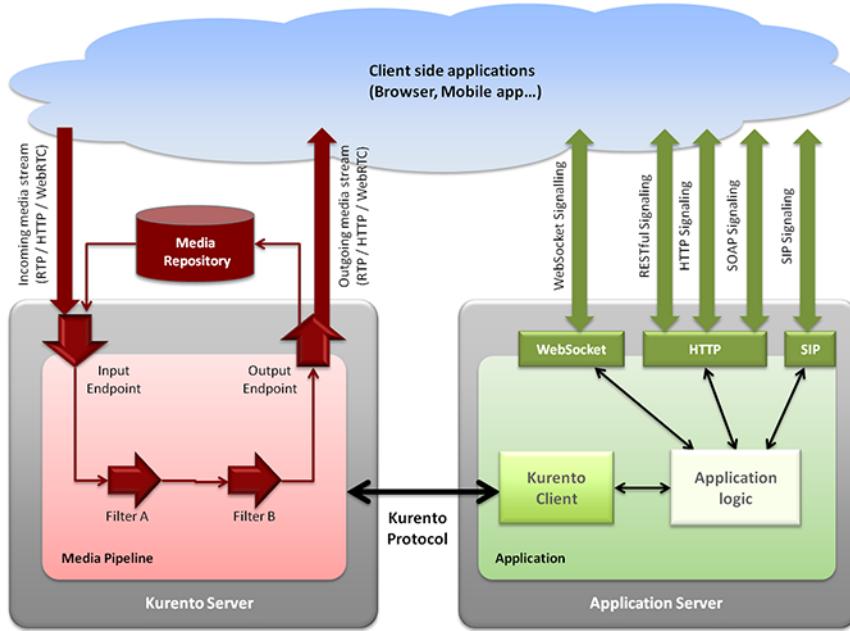


Fig. 9.2: *Kurento Architecture*. Kurento architecture follows the traditional separation between signaling and media planes.

The **right side** of the picture shows the application, which is in charge of the signaling plane and contains the business logic and connectors of the particular multimedia application being deployed. It can be built with any programming technology like Java, Node.js, PHP, Ruby, .NET, etc. The application can use mature technologies such as [HTTP](#) and [SIP](#) Servlets, Web Services, database connectors, messaging services, etc. Thanks to this, this plane provides access to the multimedia signaling protocols commonly used by end-clients such as [SIP](#), RESTful and raw HTTP based formats, SOAP, RMI, CORBA or JMS. These signaling protocols are used by client side of applications to command the creation of media sessions and to negotiate their desired characteristics on their behalf. Hence, this is the part of the architecture, which is in contact with application developers and, for this reason, it needs to be designed pursuing simplicity and flexibility.

On the **left side**, we have the Kurento Media Server, which implements the media plane capabilities providing access to the low-level media features: media transport, media encoding/decoding, media transcoding, media mixing, media processing, etc. The Kurento Media Server must be capable of managing the multimedia streams with minimal latency and maximum throughput. Hence the Kurento Media Server must be optimized for efficiency.

9.2.1 Communicating client, server and Kurento

As can be observed in the figure above, a Kurento application involves interactions among three main modules:

- **Client Application:** Involves the native multimedia capabilities of the client platform plus the specific client-side application logic. It can use Kurento Clients designed for client platforms (for example, Kurento JavaScript Client).
- **Application Server:** Involves an application server and the server-side application logic. It can use Kurento Clients designed for server platforms (for example, Kurento Java Client for Java EE and Kurento JavaScript Client for Node.js).

- **Kurento Media Server:** Receives commands to create specific multimedia capabilities (i.e. specific pipelines adapted to the needs of the application).

The interactions maintained among these modules depend on the specifics of each application. However, in general, for most applications can be reduced to the following conceptual scheme:

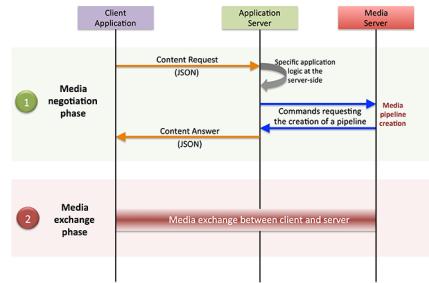


Fig. 9.3: Main interactions between architectural modules. These occur in two phases: negotiation and media exchange. Remark that the color of the different arrows and boxes is aligned with the architectural figures presented above. For example, orange arrows show exchanges belonging to the signaling plane, blue arrows show exchanges belonging to the Kurento Protocol, red boxes are associated to the Kurento Media Server; and green boxes with the application.

1. Media negotiation phase (signaling)

At a first stage, a client (a browser in a computer, a mobile application, etc.) issues a message to the application requesting some kind of multimedia capability. This message can be implemented with any protocol (HTTP, WebSocket, SIP, etc.). For instance, that request could ask for the visualization of a given video clip.

When the application receives the request, if appropriate, it will carry out the specific server side application logic, which can include Authentication, Authorization and Accounting (AAA), CDR generation, consuming some type of web service, etc.

After that, the application processes the request and, according to the specific instructions programmed by the developer, commands Kurento Media Server to instantiate the suitable Media Elements and to chain them in an appropriate Media Pipeline. Once the pipeline has been created successfully, Kurento Media Server responds accordingly and the application forwards the successful response to the client, showing it how and where the media service can be reached.

During the above mentioned steps no media data is really exchanged. All the interactions have the objective of negotiating the *whats*, *hows*, *wheres* and *whens* of the media exchange. For this reason, we call it the negotiation phase. Clearly, during this phase only signaling protocols are involved.

2. Media exchange phase

After the signaling part, a new phase starts with the aim to produce the actual media exchange. The client addresses a request for the media to the Kurento Media Server using the information gathered during the negotiation phase.

Following with the video-clip visualization example mentioned above, the browser will send a GET request to the IP address and port of the Kurento Media Server where the clip can be obtained and, as a result, an HTTP reponse containing the media will be received.

Following the discussion with that simple example, one may wonder why such a complex scheme for just playing a video, when in most usual scenarios clients just send the request to the appropriate URL of the video without requiring any negotiation. The answer is straightforward. Kurento is designed for media applications involving complex media processing. For this reason, we need to establish a two-phase mechanism enabling a negotiation before the media exchange. The price to pay is that simple applications, such as one just downloading a video, also need to get through

these phases. However, the advantage is that when creating more advanced services the same simple philosophy will hold. For example, if we want to add Augmented Reality or Computer Vision features to that video-clip, we just need to create the appropriate pipeline holding the desired Media Elements during the negotiation phase. After that, from the client perspective, the processed clip will be received as any other video.

9.2.2 Real time WebRTC applications with Kurento

The client communicates its desired media capabilities through an *SDP Offer/Answer* negotiation. Hence, Kurento is able to instantiate the appropriate WebRTC endpoint, and to require it to generate an SDP Answer based on its own capabilities and on the SDP Offer. When the SDP Answer is obtained, it is given back to the client and the media exchange can be started. The interactions among the different modules are summarized in the following picture:

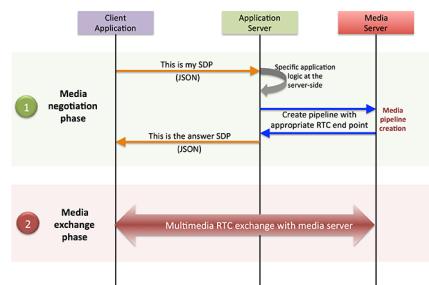


Fig. 9.4: *Interactions in a WebRTC session. During the negotiation phase, an SDP Offer is sent to KMS, requesting the capabilities of the client. As a result, Kurento Media Server generates an SDP Answer that can be used by the client for establishing the media exchange.*

The application developer is able to create the desired pipeline during the negotiation phase, so that the real-time multimedia stream is processed accordingly to the application needs.

As an example, imagine that you want to create a WebRTC application recording the media received from the client and augmenting it so that if a human face is found, a hat will be rendered on top of it. This pipeline is schematically shown in the figure below, where we assume that the Filter element is capable of detecting the face and adding the hat to it.

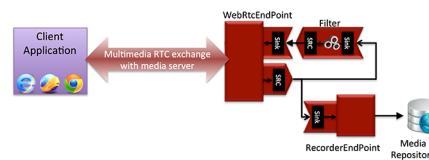


Fig. 9.5: *Example pipeline for a WebRTC session. A WebRtcEndPoint is connected to a RecorderEndpoint storing the received media stream and to an Augmented Reality filter, which feeds its output media stream back to the client. As a result, the end user will receive its own image filtered (e.g. with a hat added onto her head) and the stream will be recorded and made available for further recovery into a repository (e.g. a file).*

9.3 Media Plane

From the application developer perspective, Media Elements are like *Lego* pieces: you just need to take the elements needed for an application and connect them, following the desired topology. In Kurento jargon, a graph of connected media elements is called a **Media Pipeline**. Hence, when creating a pipeline, developers need to determine the capabilities they want to use (the Media Elements) and the topology determining which Media Element provides media to which other Media Elements (the connectivity).

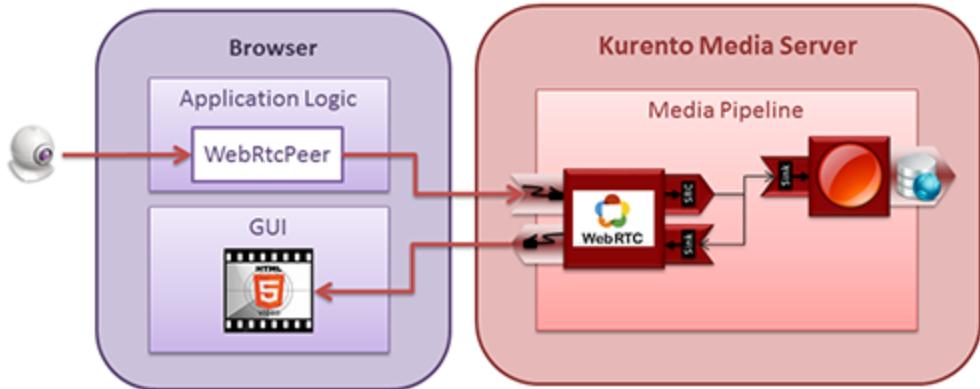


Fig. 9.6: Simple Example of a Media Pipeline

The connectivity is controlled through the *connect* primitive, exposed on all Kurento Client APIs.

This primitive is always invoked in the element acting as source and takes as argument the sink element following this scheme:

```
sourceMediaElement.connect(sinkMediaElement)
```

For example, if you want to create an application recording WebRTC streams into the file system, you'll need two media elements: *WebRtcEndpoint* and *RecorderEndpoint*. When a client connects to the application, you will need to instantiate these media elements making the stream received by the *WebRtcEndpoint* (which is capable of receiving WebRTC streams) to be fed to the *RecorderEndpoint* (which is capable of recording media streams into the file system). Finally you will need to connect them so that the stream received by the former is transferred into the later:

```
WebRtcEndpoint.connect(RecorderEndpoint)
```

To simplify the handling of WebRTC streams in the client-side, Kurento provides an utility called *WebRtcPeer*. Nevertheless, the standard WebRTC API (*getUserMedia*, *RTCPeerConnection*, and so on) can also be used to connect to *WebRtcEndpoints*. For further information please visit the [Tutorials section](#).

CHAPTER 10

Writing Kurento Modules

Table of Contents

- *Writing Kurento Modules*
 - *OpenCV module*
 - *GStreamer module*
 - *For both kind of modules*
 - *Examples*

[TODO REVIEW]

You can expand the Kurento Media Server developing your own modules. There are two flavors of Kurento modules:

- Modules based on *OpenCV*. This kind of modules are recommended if you would like to develop a filter providing Computer Vision or Augmented Reality features.
- Modules based on *GStreamer*. This kind of modules provide a generic entry point for media processing with the GStreamer framework. Such modules are more powerful but also they are more difficult to develop. Skills in GStreamer development are necessary.

The starting point to develop a filter is to create the filter structure. For this task, you can use the `kurento-module-scaffold` tool. This tool is distributed with the `kurento-media-server-dev` package. To install this tool run this command:

```
sudo apt-get install kurento-media-server-dev
```

The tool usage is different depending on the chosen flavor:

1. OpenCV module:

```
kurento-module-scaffold.sh <module_name> <output_directory> opencv_filter
```

2. Gstreamer module:

```
kurento-module-scaffold.sh <module_name> <output_directory>
```

The tool generates the folder tree, all the needed `CmakeLists.txt` files, and example files of Kurento module descriptor files (`.kmd`). These files contain the description of the modules, the constructor, the methods, the properties, the events and the complex types defined by the developer.

Once `kmd` files are completed it is time to generate the corresponding code. The tool `kurento-module-creator` generates glue code to server-side. Run this from the root directory:

```
cd build  
cmake ..
```

The following sections detail how to create your module depending on the filter type you chose (OpenCV or GStreamer).

10.1 OpenCV module

We have four files in `src/server/implementation/`:

```
ModuleNameImpl.cpp  
ModuleNameImpl.hpp  
ModuleNameOpenCVImpl.cpp  
ModuleNameOpenCVImpl.hpp
```

The first two files should not be modified. The last two files will contain the logic of your module.

The file `ModuleNameOpenCVImpl.cpp` contains functions to deal with the methods and the parameters (you must implement the logic). Also, this file contains a function called `process`. This function will be called with each new frame, thus you must implement the logic of your filter inside it.

10.2 GStreamer module

In this case, we have two directories inside the `src/` folder:

- The `gst-plugins/` folder contains the implementation of your GStreamer Element (the `kurento-module-scaffold` generates a dummy filter).
- Inside the `server/objects/` folder you have two files:

```
ModuleNameImpl.cpp  
ModuleNameImpl.hpp
```

In the file `ModuleNameImpl.cpp` you have to invoke the methods of your GStreamer element. The module logic will be implemented in the GStreamer Element.

10.3 For both kind of modules

If you need extra compilation dependencies you can add compilation rules to the `kurento-module-creator` using the function `generate_code` in the `CmakeLists.txt` file, located in `src/server/`.

The following parameters are available:

- SERVER_STUB_DESTINATION (required) The generated code that you may need to modify will be generated on the folder indicated by this parameter.
- MODELS (required) This parameter receives the folders where the models (.kmd files) are located.
- INTERFACE_LIB_EXTRA_SOURCES INTERFACE_LIB_EXTRA_INCLUDE_DIRS INTERFACE_LIB_EXTRA_HEADERS INTERFACE_LIB_EXTRA_LIBRARIES These parameters allow to add additional source code to the static library. Files included in INTERFACE_LIB_EXTRA_HEADERS will be installed in the system as headers for this library. All the parameters accept a list as input.
- SERVER_IMPL_LIB_EXTRA_SOURCES SERVER_IMPL_LIB_EXTRA_INCLUDE_DIRS SERVER_IMPL_LIB_EXTRA_HEADERS SERVER_IMPL_LIB_EXTRA_LIBRARIES These parameters allow to add additional source code to the interface library. Files included in SERVER_IMPL_LIB_EXTRA_HEADERS will be installed in the system as headers for this library. All the parameters accept a list as input.
- MODULE_EXTRA_INCLUDE_DIRS MODULE_EXTRA_LIBRARIES These parameters allow to add extra include directories and libraries to the module.
- SERVER_IMPL_LIB_FIND_CMAKE_EXTRA_LIBRARIES This parameter receives a list of strings. Each string has this format: libname[libversion range] (possible ranges can use symbols AND OR < <= > >= ^ and ~).
 - ^ indicates a version compatible using *Semantic Versioning*.
 - ~ Indicates a version similar, that can change just last indicated version character.

Once the module logic is implemented and the compilation process is finished, you need to install your module in your system. You can follow two different ways:

1. You can generate the Debian package (`debuild -us -uc`) and install it (`dpkg -i`).
2. You can define the following environment variables in the file `/etc/default/kurento`:

```
KURENTO_MODULES_PATH=<module_path>/build/src
GST_PLUGIN_PATH=<module_path>/build/src
```

Now, you need to generate code for Java or JavaScript to use your module from the client-side.

- For Java, from the build directory you have to execute `cmake .. -DGENERATE_JAVA_CLIENT_PROJECT=TRUE` command, that generates a Java folder with client code. You can run `make java_install` and your module will be installed in your Maven local repository. To use the module in your Maven project, you have to add the dependency to the `pom.xml` file:

```
<dependency>
  <groupId>org.kurento.module</groupId>
  <artifactId>modulename</artifactId>
  <version>moduleversion</version>
</dependency>
```

- For JavaScript, you should run `cmake .. -DGENERATE_JS_CLIENT_PROJECT=TRUE`. This command generates a `js/` folder with client code. Now you can manually add the JavaScript library to use your module in your application. Alternatively, you can use *Bower* (for *Browser JavaScript*) or *NPM* (for *Node.js*). To do that, you should add your JavaScript module as a dependency in your `bower.json` or `package.json` file respectively, as follows:

```
"dependencies": {
  "modulename": "moduleversion"
}
```

10.4 Examples

Simple examples for both kind of modules are available in GitHub:

- [OpenCV module.](#)
- [GStreamer module.](#)

There are a lot of examples showing how to define methods, parameters or events in all our public built-in modules:

- [kms-pointerdetector.](#)
- [kms-crowddetector.](#)
- [kms-chroma.](#)
- [kms-platedetector.](#)

Moreover, all our modules are developed using this methodology. For that reason you can take a look to our main modules:

- [kms-core.](#)
- [kms-elements.](#)
- [kms-filters.](#)

CHAPTER 11

Support

11.1 Usage Questions

If you have questions about how to use Kurento, or have an issue that isn't related to a bug, [Stack Overflow](#) is the best place to ask. Tag questions with `kurento` so other folks can find them easily.

Another option you have is to contact the community through the [Kurento Public Mailing List](#), which is a medium that allows more discussion to happen around topics.

Good questions to ask would be:

- What is the best Media Pipeline to use for <place your use case here>?
- How do I check that the ICE connectivity checks are working properly for WebRTC?
- Which audio/video codec combination should I use to ensure no transcoding needs to take place?

11.2 Community Support

Kurento is a community supported project, nobody is paid explicitly to offer user support. **All people answering your questions are doing it with their own time, so please be kind and provide as much information as possible.**

11.2.1 Bugs & Support Issues

You can file bug reports on our [Issue Tracker](#), and they will be addressed as soon as possible.

Support is a volunteer effort, and there is no guaranteed response time. If you need answers quickly, you can buy commercial support as explained below.

11.2.2 Reporting Issues

When reporting a bug, please include as much information as possible, this will help us solve the problem. Also, try to follow these guidelines as closely as possible, because making it easier for someone to work on the issue means that there are more probabilities that the issue gets fixed:

- **Be curious.** Has it been asked before? Is it really a bug? Everybody hates duplicated reports. Google is your friend!
- **Be proactive.** Check with other versions, specially with the *latest development version*. We can't emphasize this enough: *it's the first thing that we are going to ask*.
- **Be precise.** Don't wander around your situation and go straight to the point, unless the context around it is technically required to understand what is going on. Describe as precisely as possible what you are doing and what is happening but shouldn't happen.
- **Be specific.** Explain how to reproduce the problem, being very systematic about it: step by step, so others can reproduce the bug. Also, report only *one problem per opened issue*.

If you definitely think you have hit a bug, try to include these in your bug report:

- A description of the problem (e.g. what type of abnormal effect you are seeing).
- A detailed specification of what you were executing (e.g. a specific code snippet firing the bug).
- A detailed description of the execution environment (e.g. browser, operating system, KMS version, etc).
- The **relevant** log generated by KMS, browser and, eventually, application server. Make sure to honor the *relevant* part: providing a 50MB log where only 10 lines are of interest *is not* providing a relevant log.
- **A proof-of-concept is of great help.** This may need a bit of upfront work on your side, to isolate the actual component that presents issues from the rest of your application logic. Doing this will hugely increase the chances that developers of Kurento start working right away on the issue, if they are able to reproduce the problem without needing to have your whole system in place.

11.3 Commercial Support

We offer commercial support for Kurento, custom features, as well as consulting around all media server issues. You can contact us at openvidu@gmail.com to learn more.

CHAPTER 12

Frequently Asked Questions

12.1 How To ...

12.1.1 Install Coturn (TURN/STUN server)

If you are installing Kurento in a NAT environment (eg. in any cloud provider), you need to use a TURN/STUN server, and configure KMS appropriately in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Apart from that, you need to open all UDP ports in your security group, as STUN/TURN will use any port available from the whole 0-65535 range.

On Ubuntu 16.04 (Xenial), Coturn can be installed directly from the package repositories:

```
sudo apt-get install coturn
```

However, Ubuntu 14.04 (Trusty) lacks this package, but it can be downloaded and installed manually from the Debian repositories:

1. Download the file `coturn_<...>_amd64.deb` from any of the mirrors listed here: <https://packages.debian.org/jessie-backports/amd64/coturn/download>
2. Install it, together with all dependencies.

```
sudo apt-get update  
sudo apt-get install gdebi-core  
sudo gdebi coturn*.deb
```

3. Edit the file `/etc/turnserver.conf` and configure the TURN server.
 - For Amazon EC2 or similar, the Local and External IPs should be configured via the `relay-ip` and `external-ip` parameters, respectively.
 - Enable the options needed for WebRTC:
 - `fingerprint`
 - `lt-cred-mech`

- realm=kurento.org
- Create a user and a password for the TURN server. As an example, the user “kurento” and password “kurentopw” are used. Add them in the configuration file: user=kurento:kurentopw.
 - Optionally, debug logging messages can be suppressed so they don’t clutter the standard output, enabling the option no-stdout-log.
 - Other parameters can be tuned as needed. For more information, check the Coturn help pages:
 - <https://github.com/coturn/coturn/wiki/turnserver>
 - <https://github.com/coturn/coturn/wiki/CoturnConfig>
4. Edit the file /etc/default/coturn and uncomment TURNSERVER_ENABLED=1, so the TURN server initiates automatically as a system service daemon.
 5. Configure KMS and point it to where the TURN server is listening for connections. Edit the file /etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini and set the turnURL parameter:

```
turnURL=<user>:<password>@<serverIp>:<serverPort>
turnURL=kurento:kurentopw@<serverIp>:3478
```

The parameter serverIp should be the public IP address of the TURN server. It must be an IP address, **not a domain name**.

The following ports should be open in the firewall:

- 3478 TCP & UDP.
- 49152 - 65535 UDP: As per [RFC 5766](#), these are the ports that the TURN server will use to exchange media. These ports can be changed using the min-port and max-port parameters on the TURN server.

Note: While the RFC specifies the ports used by TURN, if you are using STUN you will need to open **all UDP ports**, as STUN doesn’t constrain the range of ports that might be used.

6. Lastly, start the Coturn server and the media server:

```
sudo service coturn start
sudo service kurento-media-server restart
```

Note: Make sure to check your installation using this test application:

<https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/>

12.1.2 Know how many Media Pipelines do I need for my Application?

Media Elements can only communicate with each other when they are part of the same pipeline. Different Mediapielines in the server are independent do not share audio, video, data or events.

A good heuristic is that you will need one pipeline per each set of communicating partners in a channel, and one Endpoint in this pipeline per audio/video streams reaching a partner.

12.1.3 Know how many Endpoints do I need?

Your application will need to create an Endpoint for each media stream flowing to (or from) the pipeline. As we said in the previous answer, each set of communicating partners in a channel will be in the same Media Pipeline, and each of them will use one or more Endpoints. They could use more than one if they are recording or reproducing several streams.

12.1.4 Know to what client a given WebRtcEndPoint belongs or where is it coming from?

Kurento API currently offers no way to get application attributes stored in a Media Element. However, the application developer can maintain a hashmap or equivalent data structure mapping the `WebRtcEndpoint` internal Id (which is a string) to whatever application information is desired.

12.2 Why do I get the error ...

12.2.1 “Cannot create gstreamer element”?

This is a typical error which happens when you update Kurento Media Server from version 4 to 5. The problem is related to the GStreamer dependency version. The solution is the following:

```
sudo apt-get remove kurento*
sudo apt-get autoremove
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get install kurento-media-server
```


CHAPTER 13

Kurento API

Table of Contents

- *Kurento API*
 - *Media Elements and Media Pipelines*
 - *Endpoints*
 - *Filters*
 - *Hubs*

Kurento Media Server can be controlled through the API it exposes, so application developers can use high level languages to interact with it. The Kurento project already provides *Kurento Client* implementations of this API for several platforms.

If you prefer a programming language different from the supported ones, you can implement your own Kurento Client by using the *Kurento Protocol*, which is based on *WebSocket* and *JSON-RPC*.

In the following sections we will describe the Kurento API from a high-level point of view, showing the media capabilities exposed by Kurento Media Server to clients. If you want to see working demos using Kurento, please refer to the *Tutorials section*.

13.1 Media Elements and Media Pipelines

Kurento is based on two concepts that act as building blocks for application developers:

- **Media Elements.** A Media Element is a functional unit performing a specific action on a media stream. Media Elements are a way of every capability is represented as a self-contained “black box” (the Media Element) to the application developer, who does not need to understand the low-level details of the element for using it. Media Elements are capable of *receiving* media from other elements (through media sources) and of *sending* media to other elements (through media sinks). Depending on their function, Media Elements can be split into different groups:

- **Input Endpoints:** Media Elements capable of receiving media and injecting it into a pipeline. There are several types of input endpoints. File input endpoints take the media from a file, Network input endpoints take the media from the network, and Capture input endpoints are capable of capturing the media stream directly from a camera or other kind of hardware resource.
- **Filters:** Media Elements in charge of transforming or analyzing media. Hence there are filters for performing operations such as mixing, muxing, analyzing, augmenting, etc.
- **Hubs:** Media Objects in charge of managing multiple media flows in a pipeline. A Hub has several hub ports where other Media Elements are connected. Depending on the Hub type, there are different ways to control the media. For example, there are a Hub called Composite that merge all input video streams in a unique output video stream with all inputs in a grid.
- **Output Endpoints:** Media Elements capable of taking a media stream out of the pipeline. Again, there are several types of output endpoints specialized in files, network, screen, etc.
- **Media Pipeline:** A Media Pipeline is a chain of Media Elements, where the output stream generated by one element (source) is fed into one or more other elements input streams (sinks). Hence, the pipeline represents a “machine” capable of performing a sequence of operations over a stream.

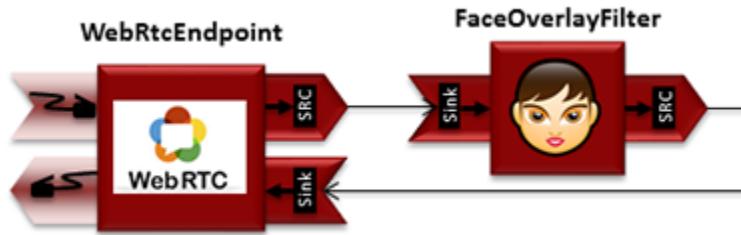


Fig. 13.1: Example of a Media Pipeline implementing an interactive multimedia application receiving media from a WebRtcEndpoint, overlaying an image on the detected faces and sending back the resulting stream

Kurento API is an object-oriented API. That is, there are classes that can be instantiated. These classes define operations that can be invoked over objects of this classes. The classes can have an inheritance relationship with other classes, inheriting operations from parent classes to children ones.

The following class diagram shows some of the relationships of the main classes in the Kurento API:

13.2 Endpoints

A **WebRtcEndpoint** is an input/output endpoint that provides media streaming for Real Time Communications (RTC) through the web. It implements [WebRTC](#) technology to communicate with browsers.



An **RtpEndpoint** is an input/output endpoint that provides bidirectional content delivery capabilities with remote networked peers, through the [RTP](#) protocol. It uses [SDP](#) for media negotiation.

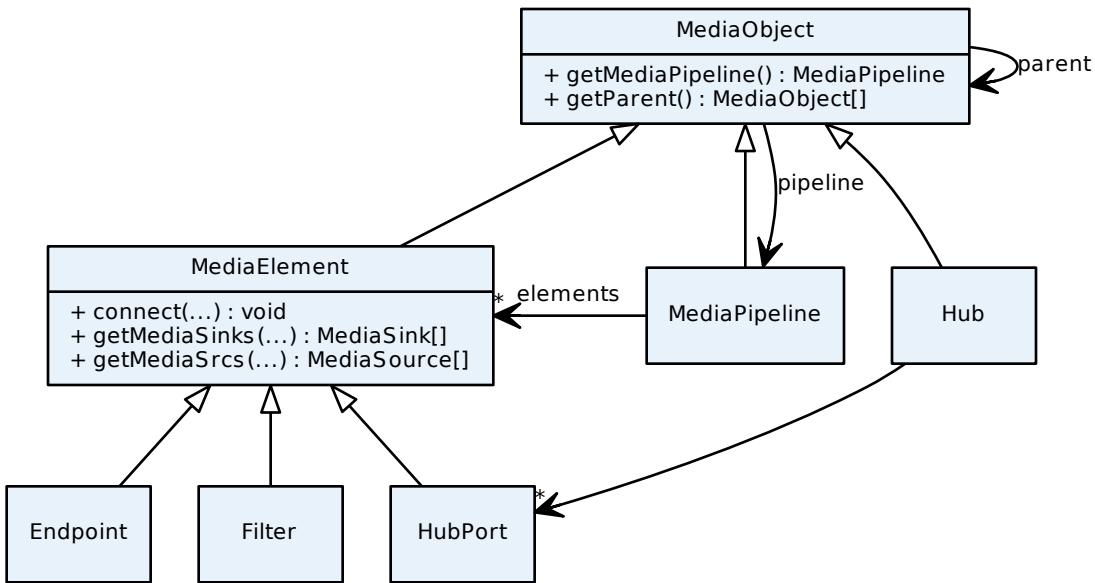


Fig. 13.2: Class diagram of main classes in Kurento API



An **HttpPostEndpoint** is an input endpoint that accepts media using HTTP POST requests like HTTP file upload function.



A **PlayerEndpoint** is an input endpoint that retrieves content from file system, HTTP URL or RTSP URL and injects it into the Media Pipeline.



A **RecorderEndpoint** is an output endpoint that provides function to store contents in reliable mode (doesn't discard data). It contains **Media Sink** pads for audio and video.



The following class diagram shows the relationships of the main endpoint classes:

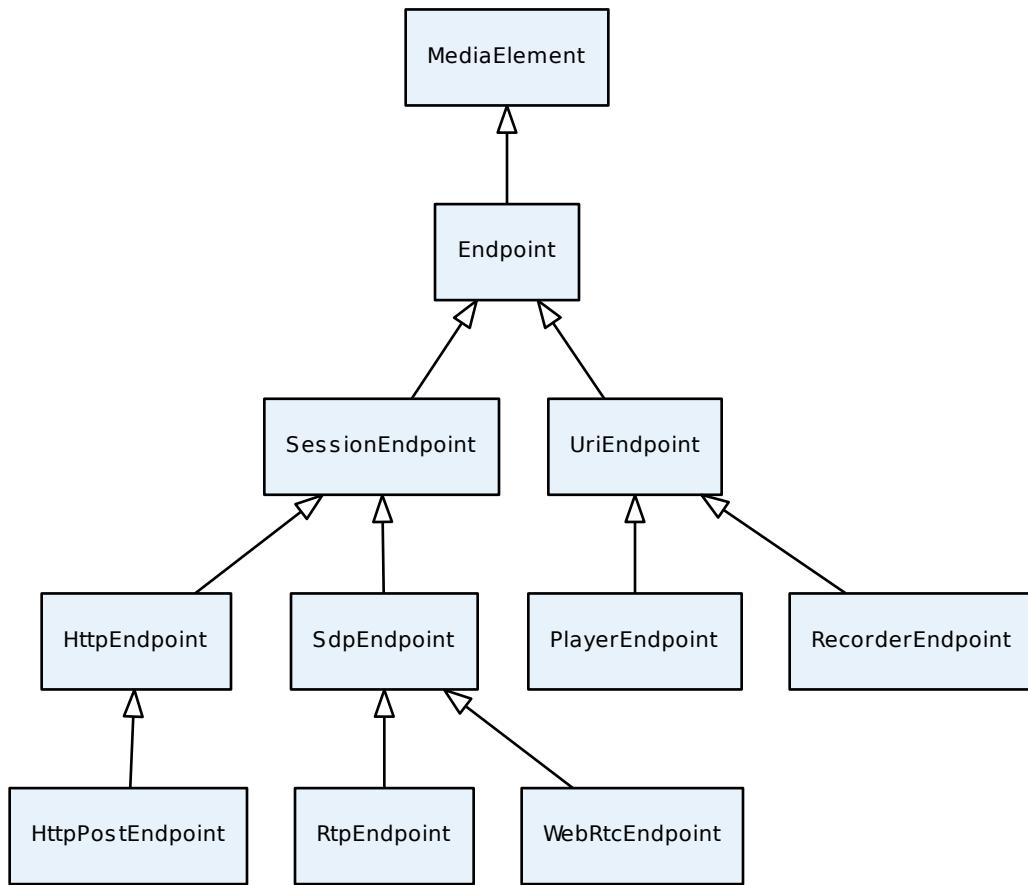


Fig. 13.3: Class diagram of main Endpoints in Kurento API

13.3 Filters

Filters are MediaElements that perform media processing, Computer Vision, Augmented Reality, and so on.

The **ZBarFilter** filter detects QR and bar codes in a video stream. When a code is found, the filter raises a `CodeFoundEvent`. Clients can add a listener to this event to execute some action.



The **FaceOverlayFilter** filter detects faces in a video stream and overlaid it with a configurable image.



GStreamerFilter is a generic filter interface that allow use GStreamer filter in Kurento Media Pipelines.



The following class diagram shows the relationships of the main filter classes:

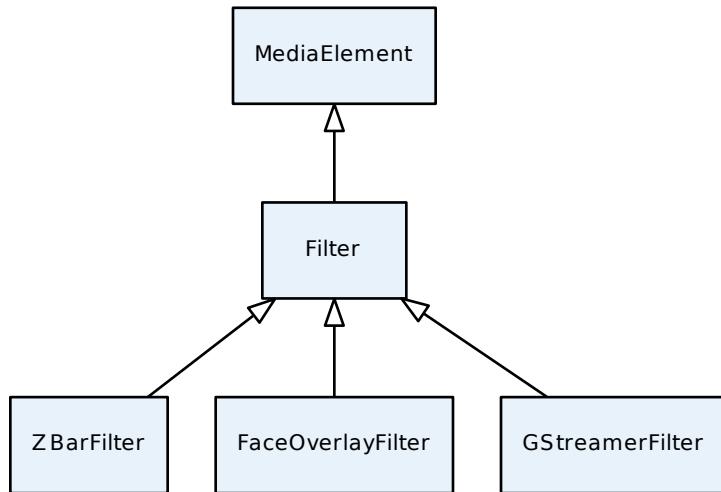


Fig. 13.4: Class diagram of main Filters in Kurento API

13.4 Hubs

Hubs are media objects in charge of managing multiple media flows in a pipeline. A Hub has several hub ports where other Media Elements are connected.

Composite is a hub that mixes the audio stream of its connected inputs and constructs a grid with the video streams of them.



DispatcherOneToMany is a Hub that sends a given input to all the connected output HubPorts.



Dispatcher is a hub that allows routing between arbitrary input-output HubPort pairs.



The following class diagram shows the relationships of the hubs:

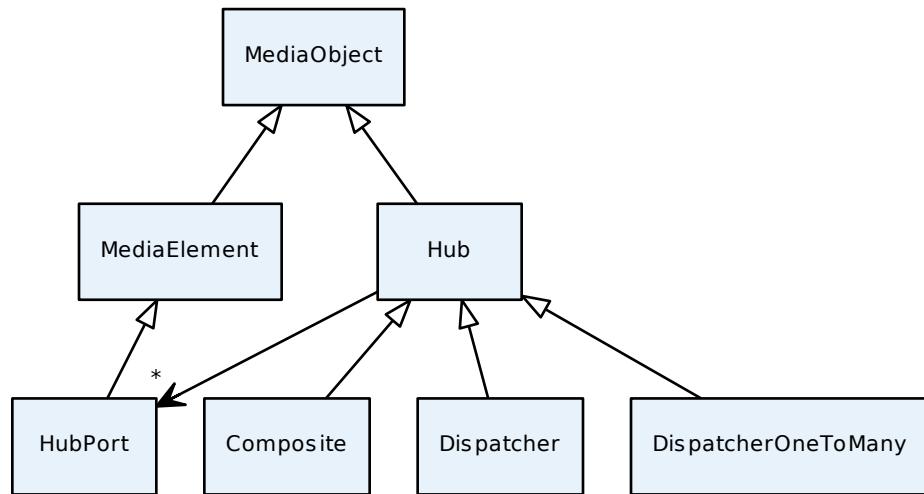


Fig. 13.5: Class diagram of main Hubs in Kurento API

CHAPTER 14

Kurento Client

Currently, the Kurento project provides implementations of the *Kurento API* for two programming languages: *Java* and *JavaScript*.

In the future, additional Kurento Clients can be created, exposing the same kind of modularity in other languages such as Python, C/C++, PHP, etc.

14.1 Kurento Java Client

Kurento Java Client is a Java SE layer which consumes the Kurento API and exposes its capabilities through a simple-to-use interface based on Java POJOs representing Media Elements and Media Pipelines.

This API is abstract in the sense that all the non-intuitive inherent complexities of the internal Kurento Protocol workings are abstracted and developers do not need to deal with them when creating applications. Using the Kurento Java Client only requires adding the appropriate dependency to a *Maven* project or to download the corresponding *jar* into the application's *Java Classpath*.

It is important to remark that the Kurento Java Client is a media-plane control API. In other words, its objective is to expose the capability of managing media objects, but it does not provide any signaling plane capabilities.

14.2 Kurento JavaScript Client

Kurento JavaScript Client is a JavaScript layer which consumes the Kurento API and exposes its capabilities to JavaScript developers. It allow to build *Node.js* and browser based applications.

14.3 Reference Documentation

- [Kurento Client JavaDoc](#)
- [Kurento Client JsDoc](#)

- [Kurento Js Utils](#): a JavaScript utility library aimed to simplify the development of WebRTC applications.

CHAPTER 15

Kurento Protocol

Table of Contents

- *Kurento Protocol*
 - *JSON-RPC message format*
 - * *Request*
 - * *Successful Response*
 - * *Error Response*
 - *Kurento API over JSON-RPC*
 - * *Ping*
 - * *Create*
 - * *Invoke*
 - * *Release*
 - * *Subscribe*
 - * *Unsubscribe*
 - * *OnEvent*
 - *Network issues*
 - *Example: WebRTC in loopback*
 - *Creating a custom Kurento Client*
 - * *Kurento Module Creator*

Kurento Media Server is controlled by means of an *RPC API*, implemented in terms of the **Kurento Protocol** specification as described in this document, based on *WebSocket* and *JSON-RPC*.

15.1 JSON-RPC message format

Kurento Protocol uses the [JSON-RPC](#) 2.0 Specification to encode its API messages. The following subsections describe the contents of the [JSON](#) messages that follow this spec.

15.1.1 Request

An *RPC call* is represented by sending a *request* message to a server. The *request* message has the following members:

- **jsonrpc**: A string specifying the version of the JSON-RPC protocol. It must be 2.0.
- **id**: A unique identifier established by the client that contains a string or number. The server must reply with the same value in the *response* message. This member is used to correlate the context between both messages.
- **method**: A string containing the name of the method to be invoked.
- **params**: A structured value that holds the parameter values to be used during the invocation of the method.

The following JSON shows a sample request for the creation of a *PlayerEndpoint* Media Element:

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "create",  
  "params": {  
    "type": "PlayerEndpoint",  
    "constructorParams": {  
      "pipeline": "6829986",  
      "uri": "http://host/app/video.mp4"  
    },  
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"  
  }  
}
```

15.1.2 Successful Response

When an *RPC call* is made, the server replies with a *response* message. In case of a successful response, the *response* message will contain the following members:

- **jsonrpc**: A string specifying the version of the JSON-RPC protocol. It must be 2.0.
- **id**: Must match the value of the *id* member in the *request* message.
- **result**: Its value is determined by the method invoked on the server.
- In case the connection is rejected, the response includes a message with a *rejected* attribute containing a message with a *code* and *message* attributes with the reason why the session was not accepted, and no *sessionId* is defined.

The following example shows a typical successful response:

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "result": {  
    "value": "442352747",  
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"  
}
```

```

    }
}
```

15.1.3 Error Response

When an *RPC call* is made, the server replies with a *response* message. In case of an error response, the *response* message will contain the following members:

- **jsonrpc**: A string specifying the version of the JSON-RPC protocol. It must be `2.0`.
- **id**: Must match the value of the `id` member in the *request* message. If there was an error in detecting the `id` in the *request* message (e.g. *Parse Error/Invalid Request*), `id` is `null`.
- **error**: A message describing the error through the following members:
 - **code**: An integer number that indicates the error type that occurred.
 - **message**: A string providing a short description of the error.
 - **data**: A primitive or structured value that contains additional information about the error. It may be omitted. The value of this member is defined by the server.

The following example shows a typical error response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": "33",
    "message": "Invalid parameter format"
  }
}
```

15.2 Kurento API over JSON-RPC

As explained in the [Kurento API section](#), Kurento Media Server exposes a full fledged API to let applications process media in several ways.

To allow this rich API, Kurento Clients require full-duplex communications between client and server. For this reason, the Kurento Protocol is based on the [WebSocket](#) transport.

Before issuing commands, the Kurento Client requires establishing a WebSocket connection with Kurento Media Server to this URL: `ws://hostname:port/kurento`.

Once the WebSocket has been established, the Kurento Protocol offers different types of request/response messages:

- **ping**: Keep-alive method between client and Kurento Media Server.
- **create**: Instantiates a new media object, that is, a pipeline or media element.
- **invoke**: Calls a method of an existing media object.
- **subscribe**: Subscribes to some specific event, to receive notifications when it gets emitted by a media object.
- **unsubscribe**: Removes an existing subscription to an event.
- **release**: Deletes the object and release resources used by it.

The Kurento Protocol allows that Kurento Media Server sends requests to clients:

- **onEvent**: This request is sent from Kurento Media server to subscribed clients when an event occurs.

15.2.1 Ping

In order to warrant the WebSocket connectivity between the client and the Kurento Media Server, a *keep-alive* method is implemented. This method is based on a ping method sent by the client, which must be replied with a pong message from the server. If no response is obtained in a time interval, the client will assume that the connectivity with the media server has been lost. The parameter `interval` is the time available to receive the pong message from the server, in milliseconds. By default this value is **240000** (40 seconds).

This is an example of a ping request:

```
{  
  "id": 1,  
  "method": "ping",  
  "params": {  
    "interval": 240000  
  },  
  "jsonrpc": "2.0"  
}
```

The response to a ping request must contain a `result` object with a `value` parameter with a fixed name: pong. The following snippet shows the pong response to the previous ping request:

```
{  
  "id": 1,  
  "result": {  
    "value": "pong"  
  },  
  "jsonrpc": "2.0"  
}
```

15.2.2 Create

This message requests the creation of an object from the Kurento API (Media Pipelines and Media Elements). The parameter `type` specifies the type of the object to be created. The parameter `constructorParams` contains all the information needed to create the object. Each message needs different `constructorParams` to create the object. These parameters are defined in the [Kurento API section](#).

Media Elements have to be contained in a previously created Media Pipeline. Therefore, before creating Media Elements, a Media Pipeline must exist. The response of the creation of a Media Pipeline contains a parameter called `sessionId`, which must be included in the next create requests for Media Elements.

The following example shows a request message for the creation of an object of the type `MediaPipeline`:

```
{  
  "id": 2,  
  "method": "create",  
  "params": {  
    "type": "MediaPipeline",  
    "constructorParams": {},  
    "properties": {}  
  },  
  "jsonrpc": "2.0"  
}
```

The response to this request message is as follows. Notice that the parameter `value` identifies the created Media Pipelines, and `sessionId` is the identifier of the current session:

```
{
  "id": 2,
  "result": {
    "value": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
  "jsonrpc": "2.0"
}
```

The response message contains the identifier of the new object in the field `value`. As usual, the field `id` must match the value of the `id` member in the *request* message. The `sessionId` is also returned in each response.

The following example shows a request message for the creation of an object of the type `WebRtcEndpoint` within an existing Media Pipeline (identified by the parameter `mediaPipeline`). Notice that in this request, the `sessionId` is already present, while in the previous example it was not (since at that point it was unknown for the client):

```
{
  "id": 3,
  "method": "create",
  "params": {
    "type": "WebRtcEndpoint",
    "constructorParams": {
      "mediaPipeline": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline"
    },
    "properties": {},
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
  "jsonrpc": "2.0"
}
```

The response to this request message is as follows:

```
{
  "id": 3,
  "result": {
    "value": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/087b7777-
    ↲aab5-4787-816f-f0de19e5b1d9_kurento.WebRtcEndpoint",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
  "jsonrpc": "2.0"
}
```

15.2.3 Invoke

This message requests the invocation of an operation in the specified object. The parameter `object` indicates the `id` of the object in which the operation will be invoked. The parameter `operation` carries the name of the operation to be executed. Finally, the parameter `operationParams` has the parameters needed to execute the operation.

The following example shows a request message for the invocation of the operation `connect` on a `PlayerEndpoint` connected to a `WebRtcEndpoint`:

```
{
  "id": 5,
  "method": "invoke",
```

```

"params": {
    "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/76dcb8d7-
    ↪5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
    "operation": "connect",
    "operationParams": {
        "sink": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/087b7777-
    ↪aab5-4787-816f-f0de19e5b1d9_kurento.WebRtcEndpoint"
    },
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
},
"jsonrpc": "2.0"
}

```

The response message contains the value returned while executing the operation invoked in the object, or nothing if the operation doesn't return any value.

This is the typical response while invoking the operation connect (that doesn't return anything):

```

{
    "id": 5,
    "result": {
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}

```

15.2.4 Release

This message requests releasing the resources of the specified object. The parameter `object` indicates the `id` of the object to be released:

```

{
    "id": 36,
    "method": "release",
    "params": {
        "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline",
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}

```

The response message only contains the `sessionId`:

```

{
    "id": 36,
    "result": {
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}

```

15.2.5 Subscribe

This message requests the subscription to a certain kind of events in the specified object. The parameter `object` indicates the `id` of the object to subscribe for events. The parameter `type` specifies the type of the events. If a client

is subscribed for a certain type of events in an object, each time an event is fired in this object a request with method `onEvent` is sent from Kurento Media Server to the client. This kind of request is described few sections later.

The following example shows a request message requesting the subscription of the event type `EndOfStream` on a `PlayerEndpoint` object:

```
{
  "id": 11,
  "method": "subscribe",
  "params": {
    "type": "EndOfStream",
    "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/76dcb8d7-
    ↪5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
  "jsonrpc": "2.0"
}
```

The response message contains the subscription identifier. This value can be used later to remove this subscription.

This is the response of the subscription request. The `value` attribute contains the subscription id:

```
{
  "id": 11,
  "result": {
    "value": "052061c1-0d87-4fbd-9cc9-66b57c3e1280",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
  "jsonrpc": "2.0"
}
```

15.2.6 Unsubscribe

This message requests the cancellation of a previous event subscription. The parameter `subscription` contains the subscription `id` received from the server when the subscription was created.

The following example shows a request message requesting the cancellation of the subscription `353be312-b7f1-4768-9117-5c2f5a087429` for a given object:

```
{
  "id": 38,
  "method": "unsubscribe",
  "params": {
    "subscription": "052061c1-0d87-4fbd-9cc9-66b57c3e1280",
    "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/76dcb8d7-
    ↪5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
  "jsonrpc": "2.0"
}
```

The response message only contains the `sessionId`:

```
{
  "id": 38,
  "result": {
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  },
}
```

```
"jsonrpc": "2.0"  
}
```

15.2.7 OnEvent

When a client is subscribed to some events from an object, the server sends an `onEvent` request each time an event of that type is fired in the object. This is possible because the Kurento Protocol is implemented with WebSockets and there is a full duplex channel between client and server. The request that server sends to client has all the information about the event:

- **source**: The object source of the event.
- **type**: The type of the event.
- **timestamp**: Date and time of the media server.
- **tags**: Media elements can be labeled using the methods `setSendTagsInEvents` and `addTag`, present in each element. These tags are key-value metadata that can be used by developers for custom purposes. Tags are returned with each event by the media server in this field.

The following example shows a notification sent from server to client, notifying of an event `EndOfStream` for a `PlayerEndpoint` object:

```
{  
  "jsonrpc": "2.0",  
  "method": "onEvent",  
  "params": {  
    "value": {  
      "data": {  
        "source": "681f1bc8-2d13-4189-a82a-2e2b92248a21_kurento.MediaPipeline/e983997e-  
        ↪ac19-4f4b-9575-3709af8c01be_kurento.PlayerEndpoint",  
        "tags": [],  
        "timestamp": "1441277150",  
        "type": "EndOfStream"  
      },  
      "object": "681f1bc8-2d13-4189-a82a-2e2b92248a21_kurento.MediaPipeline/e983997e-  
        ↪ac19-4f4b-9575-3709af8c01be_kurento.PlayerEndpoint",  
      "type": "EndOfStream"  
    }  
  }  
}
```

Notice that this message has no `id` field due to the fact that no response is required.

15.3 Network issues

Resources handled by KMS are high-consuming. For this reason, KMS implements a garbage collector.

A Media Element is collected when the client is disconnected longer than 4 minutes. After that time, these media elements are disposed automatically. Therefore, the WebSocket connection between client and KMS should be active at all times. In case of temporary network disconnection, KMS implements a mechanism that allows the client to reconnect.

For this, there is a special kind of message with the format shown below. This message allows a client to reconnect to the same KMS instance to which it was previously connected:

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "connect",
  "params": {
    "sessionId": "4f5255d5-5695-4e1c-aa2b-722e82db5260"
  }
}
```

If KMS replies as follows:

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "result": {
    "sessionId": "4f5255d5-5695-4e1c-aa2b-722e82db5260"
  }
}
```

... this means that the client was able to reconnect to the same KMS instance. In case of reconnection to a different KMS instance, the message is the following:

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "error": {
    "code": 40007,
    "message": "Invalid session",
    "data": {
      "type": "INVALID_SESSION"
    }
  }
}
```

In this case, the client is supposed to invoke the `connect` primitive once again in order to get a new sessionId:

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "connect"
}
```

15.4 Example: WebRTC in loopback

This section describes an example of the messages exchanged between a Kurento Client and the Kurento Media Server, in order to create a WebRTC in loopback. This example is fully depicted in the [Tutorials section](#). The steps are the following:

1. Client sends a request message in order to create a Media Pipeline:

```
{
  "id": 1,
  "method": "create",
  "params": {
    "type": "MediaPipeline",
    "label": "loopback"
  }
}
```

```

    "constructorParams":{},
    "properties":{}
},
"jsonrpc":"2.0"
}

```

2. KMS sends a response message with the identifier for the Media Pipeline and the Media Session:

```
{
  "id":1,
  "result":{
    "value":"c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline",
    "sessionId":"ba4be2a1-2b09-444e-a368-f81825a6168c"
  },
  "jsonrpc":"2.0"
}
```

3. Client sends a request to create a WebRtcEndpoint:

```

{
  "id":2,
  "method":"create",
  "params": {
    "type": "WebRtcEndpoint",
    "constructorParams": {
      "mediaPipeline": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline"
    },
    "properties": {},
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  },
  "jsonrpc": "2.0"
}

```

4. KMS creates the WebRtcEndpoint and sends back to the client the Media Element identifier:

```
{
  "id":2,
  "result": {
    "value": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-
    ↵48ff-99ef-02f7fadabaf7_WebRtcEndpoint",
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  },
  "jsonrpc": "2.0"
}
```

5. Client invokes the connect primitive in the WebRtcEndpoint in order to create a loopback:

```
{
  "id":3,
  "method": "invoke",
  "params": {
    "object": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-
    ↵48ff-99ef-02f7fadabaf7_WebRtcEndpoint",
    "operation": "connect",
    "operationParams": {
      "sink": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-
    ↵48ff-99ef-02f7fadabaf7_WebRtcEndpoint"
    }
  }
}
```

```

    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
},
"jsonrpc": "2.0"
}

```

6. KMS carries out the connection and acknowledges the operation:

```

{
  "id": 3,
  "result": {
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  },
  "jsonrpc": "2.0"
}

```

7. Client invokes the `processOffer` primitive in the `WebRtcEndpoint` in order to start the *SDP Offer/Answer* negotiation for WebRTC:

```

{
  "id": 4,
  "method": "invoke",
  "params": {
    "object": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-
    ↵48ff-99ef-02f7fadabaf7_WebRtcEndpoint",
    "operation": "processOffer",
    "operationParams": {
      "offer": "SDP"
    },
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  },
  "jsonrpc": "2.0"
}

```

8. KMS carries out the SDP negotiation and returns the SDP Answer:

```

{
  "id": 4,
  "result": {
    "value": "SDP"
  },
  "jsonrpc": "2.0"
}

```

15.5 Creating a custom Kurento Client

In order to implement a Kurento Client you need to follow the reference documentation. The best way to know all details is to take a look to the IDL file that defines the interface of the Kurento elements.

We have defined a custom IDL format based on JSON. From it, we automatically generate the client code for the Kurento Client libraries:

- KMS core
- KMS elements
- KMS filters

15.5.1 Kurento Module Creator

Kurento Clients contain code that is automatically generated from the IDL interface files, using a tool named **Kurento Module Creator**. This tool can be also used to create custom clients in other languages.

Kurento Module Creator can be installed in an Ubuntu machine using the following command:

```
sudo apt-get install kurento-module-creator
```

The aim of this tool is to generate the client code and also the glue code needed in the server-side. For code generation it uses [Freemarker](#) as the template engine. The typical way to use Kurento Module Creator is by running a command like this:

```
kurento-module-creator -c <CODEGEN_DIR> -r <ROM_FILE> -r <TEMPLATES_DIR>
```

Where:

- CODEGEN_DIR: Destination directory for generated files.
- ROM_FILE: A space-separated list of *Kurento Media Element Description* (kmd files), or folders containing these files. For example, you can take a look to the kmd files within the [Kurento Media Server](#) source code.
- TEMPLATES_DIR: Directory that contains template files. As an example, you can take a look to the internal [Java templates](#) and [JavaScript templates](#) directories.

CHAPTER 16

Kurento Modules

Kurento is a pluggable framework. Each plugin in Kurento is called a *module*.

If you are interested in writing our own modules, please read the section about [Writing Kurento Modules](#).

We classify Kurento modules into three groups, namely:

- **Main modules.** Incorporated out of the box with Kurento Media Server:
 - **kms-core:** Main components of Kurento Media Server.
 - **kms-elements:** Implementation of Kurento Media Elements (*WebRtcEndpoint*, *PlayerEndpoint*, etc.)
 - **kms-filters:** Implementation of Kurento Filters (**FaceOverlayFilter**, **ZBarFilter**, etc.)
- **Built-in modules.** Extra modules developed by the Kurento team to enhance the basic capabilities of Kurento Media Server. So far, there are four built-in modules, namely:
 - **kms-pointerdetector:** Filter that detects pointers in video streams, based on color tracking. Install command:

```
sudo apt-get install kms-pointerdetector
```
 - **kms-chroma:** Filter that takes a color range in the top layer and makes it transparent, revealing another image behind. Install command:

```
sudo apt-get install kms-chroma
```
 - **kms-crowddetector:** Filter that detects people agglomeration in video streams. Install command:

```
sudo apt-get install kms-crowddetector
```
 - **kms-platedetector:** Filter that detects vehicle plates in video streams. Install command:

```
sudo apt-get install kms-platedetector
```

Warning: The plate detector module is a prototype and its results are not always accurate. Consider this if you are planning to use this module in a production environment.

- **Custom modules.** Extensions to Kurento Media Server which provides new media capabilities.

The following picture shows an schematic view of the Kurento Media Server with its different modules:

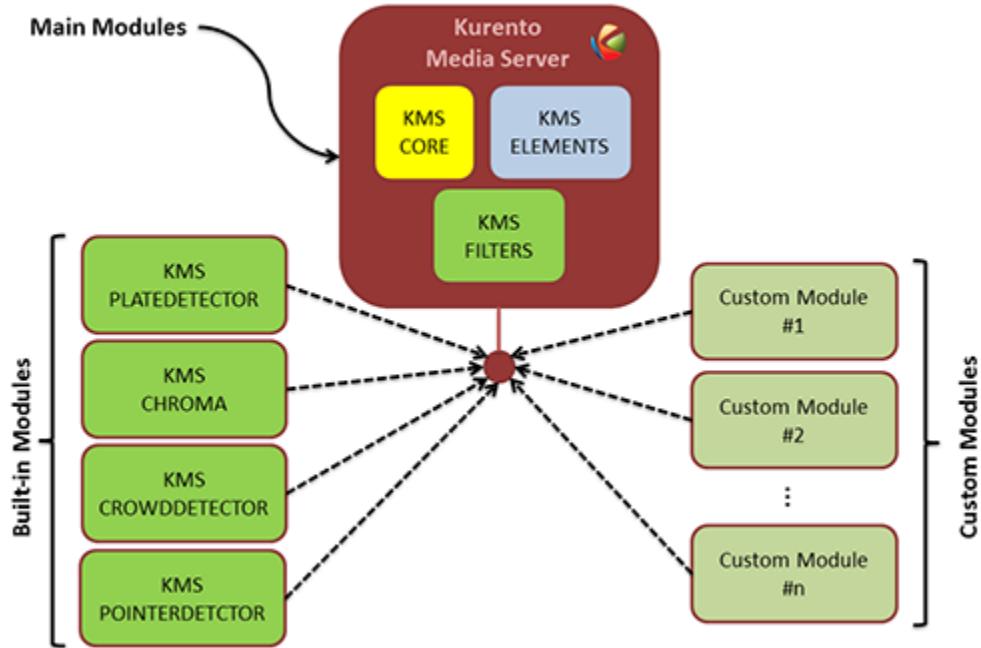


Fig. 16.1: **Kurento modules architecture** Kurento Media Server can be extended with built-it modules (*crowddetector*, *pointerdetector*, *chroma*, *platedetector*) and also with other custom modules.

Taking into account the built-in modules, the Kurento toolbox is extended as follows:

The remainder of this page is structured in four sections in which the built-in modules (*kms-pointerdetector*, *kms-chroma*, *kms-crowddetector*, *kms-platedetector*) are used to develop simple applications (tutorials) aimed to show how to use them.

16.1 Module Tutorial - Pointer Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a pointer-tracking filter element.

16.1.1 Java Module - Pointer Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a pointer tracking filter element.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

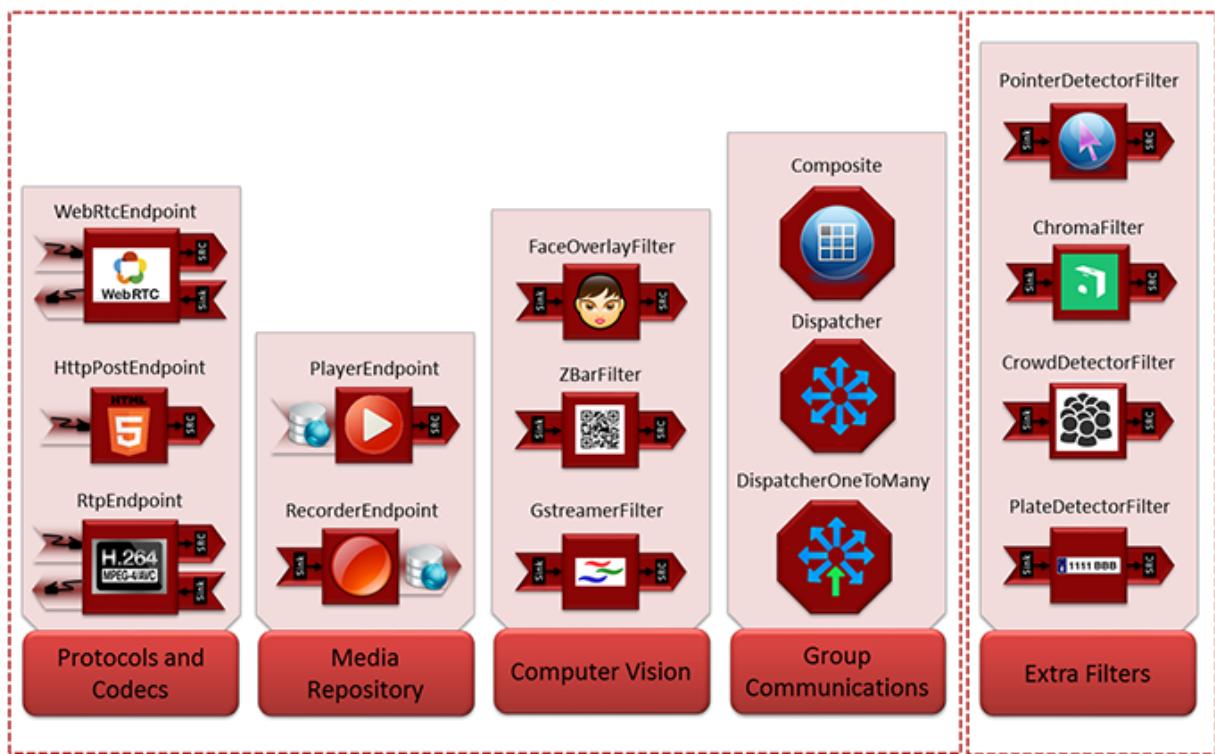


Fig. 16.2: **Extended Kurento Toolbox** The basic Kurento toolbox (left side of the picture) is extended with more Computer Vision and Augmented Reality filters (right side of the picture) provided by the built-in modules.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-pointerdetector` should be also installed:

```
sudo apt-get install kms-pointerdetector
```

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-pointerdetector
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a pointer in a WebRTC stream based on color tracking.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a [Media Pipeline](#) composed by the following [Media Element](#)s:

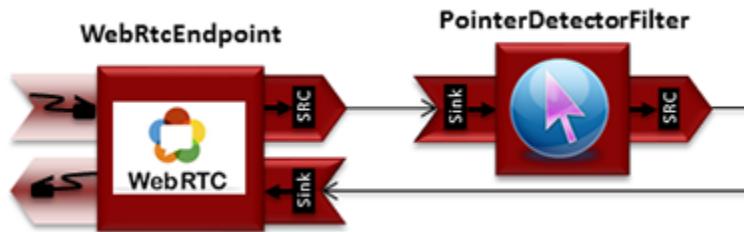
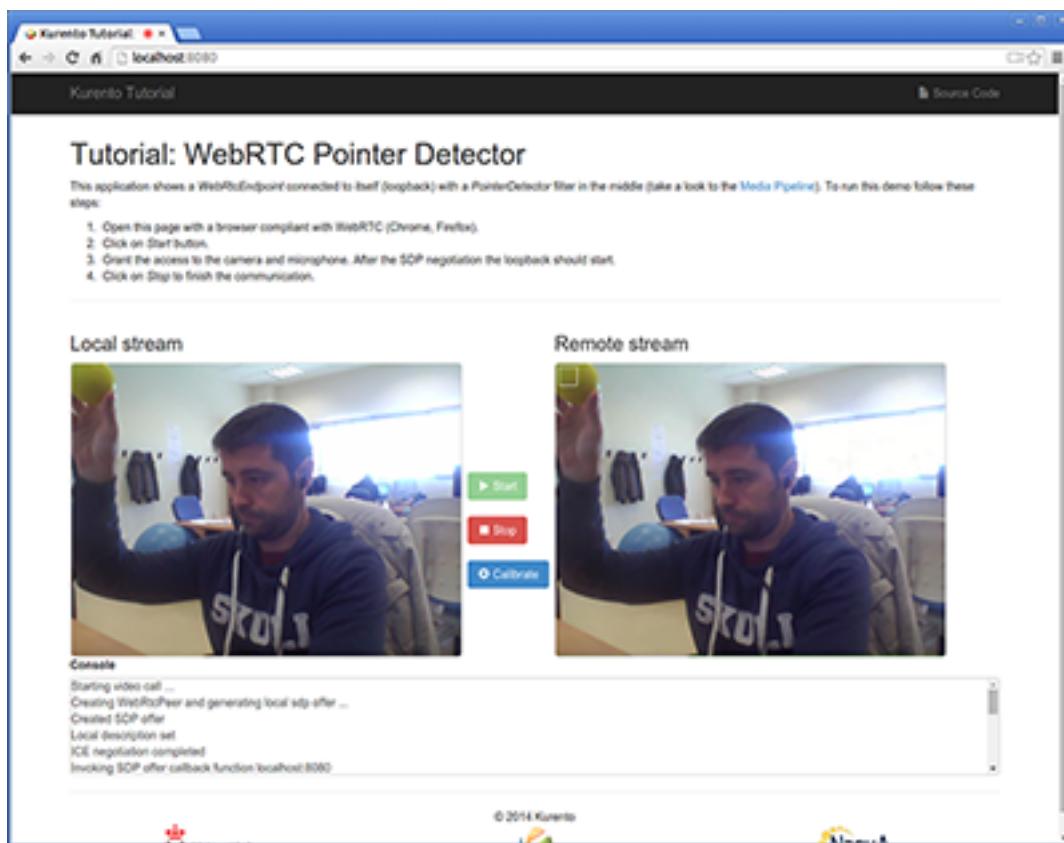


Fig. 16.3: WebRTC with PointerDetector filter in loopback Media Pipeline

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **PointerDetector** instead of **FaceOverlay** filter.

In order to perform pointer detection, there must be a calibration stage, in which the color of the pointer is registered by the filter. To accomplish this step, the pointer should be placed in a square visible in the upper left corner of the video after going through the filter, as follows:

Fig. 16.4: *Pointer calibration stage*

When the desired color to track is filling that box, a calibration message is sent from the client to the server. This is done by clicking on the *Calibrate* blue button of the GUI.

After that, the color of the pointer is tracked in real time by Kurento Media Server. `PointerDetectorFilter` can also define regions in the screen called *windows* in which some actions are performed when the pointer is detected when the pointer enters (`WindowInEvent` event) and exits (`WindowOutEvent` event) the windows. This is implemented in the server-side logic as follows:

```
// Media Logic (Media Pipeline and Elements)
UserSession user = new UserSession();
MediaPipeline pipeline = kurento.createMediaPipeline();
user.setMediaPipeline(pipeline);
WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline)
    .build();
user.setWebRtcEndpoint(webRtcEndpoint);
users.put(session.getId(), user);

webRtcEndpoint
    .addIceCandidateFoundListener(new EventListener<IceCandidateFoundEvent>() {

        @Override
        public void onEvent(IceCandidateFoundEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "iceCandidate");
            response.add("candidate", JsonUtils
                .toJsonObject(event.getCandidate()));
            try {
                synchronized (session) {
                    session.sendMessage(new TextMessage(
                        response.toString()));
                }
            } catch (IOException e) {
                log.debug(e.getMessage());
            }
        }
    });

pointerDetectorFilter = new PointerDetectorFilter.Builder(pipeline,
    new WindowParam(5, 5, 30, 30)).build();

pointerDetectorFilter
    .addWindow(new PointerDetectorWindowMediaParam("window0",
        50, 50, 500, 150));

pointerDetectorFilter
    .addWindow(new PointerDetectorWindowMediaParam("window1",
        50, 50, 500, 250));

webRtcEndpoint.connect(pointerDetectorFilter);
pointerDetectorFilter.connect(webRtcEndpoint);

pointerDetectorFilter
    .addWindowInListener(new EventListener<WindowInEvent>() {
        @Override
        public void onEvent(WindowInEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "windowIn");
            response.addProperty("roiId", event.getWindowId());
            try {

```

```

        session.sendMessage(new TextMessage(response
            .toString()));
    } catch (Throwable t) {
        sendError(session, t.getMessage());
    }
}
});

pointerDetectorFilter
    .addWindowOutListener(new EventListener<WindowOutEvent>() {

    @Override
    public void onEvent(WindowOutEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "windowOut");
        response.addProperty("roiId", event.getWindowId());
        try {
            session.sendMessage(new TextMessage(response
                .toString()));
        } catch (Throwable t) {
            sendError(session, t.getMessage());
        }
    }
});
}

// SDP negotiation (offer and answer)
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);
synchronized (session) {
    session.sendMessage(new TextMessage(response.toString()));
}

webRtcEndpoint.gatherCandidates();

```

The following picture illustrates the pointer tracking in one of the defined windows:

In order to send the calibration message from the client side, this function is used in the JavaScript side of this demo:

```

function calibrate() {
    console.log("Calibrate color");

    var message = {
        id : 'calibrate'
    }
    sendMessage(message);
}

```

When this message is received in the application server side, this code is execute to carry out the calibration:

```

private void calibrate(WebSocketSession session, JsonObject jsonMessage) {
    if (pointerDetectorFilter != null) {
        pointerDetectorFilter.trackColorFromCalibrationRegion();
    }
}

```

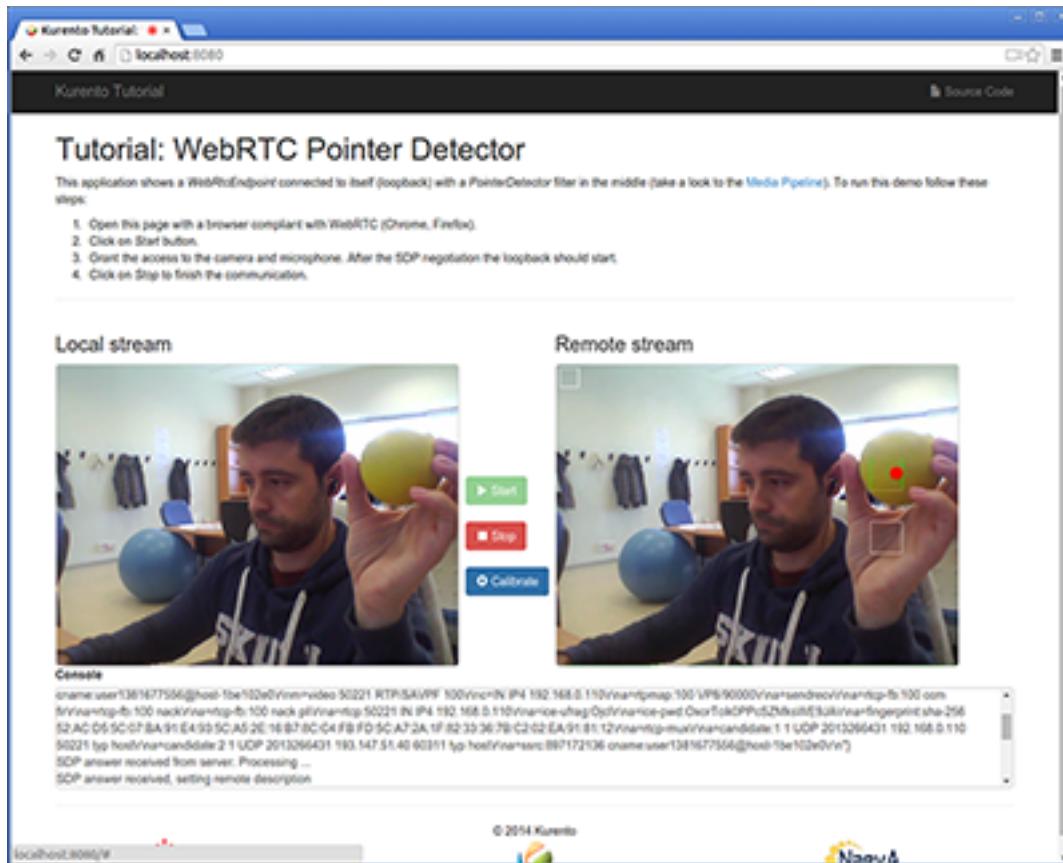


Fig. 16.5: Pointer tracking over a window

```
}
```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with webjars:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>jquery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>ekko-lightbox</artifactId>
    </dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

16.1.2 JavaScript Module - Pointer Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a pointer tracking filter element.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-pointerdetector` should be also installed:

```
sudo apt-get install kms-pointerdetector
```

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-pointerdetector
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. Kurento Media Server must use WebSockets over SSL/TLS (WSS), so make sure you check [this](#) too. It is possible to locate the KMS in other machine simple adding the parameter `ws_uri` to the URL:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

This application uses computer vision and augmented reality techniques to detect a pointer in a WebRTC stream based on color tracking.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

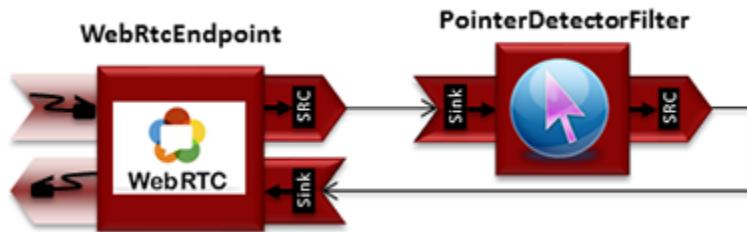


Fig. 16.6: *WebRTC with PointerDetector filter in loopback Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **PointerDetector** instead of **FaceOverlay** filter.

In order to perform pointer detection, there must be a calibration stage, in which the color of the pointer is registered by the filter. To accomplish this step, the pointer should be placed in a square in the upper left corner of the video, as follows:

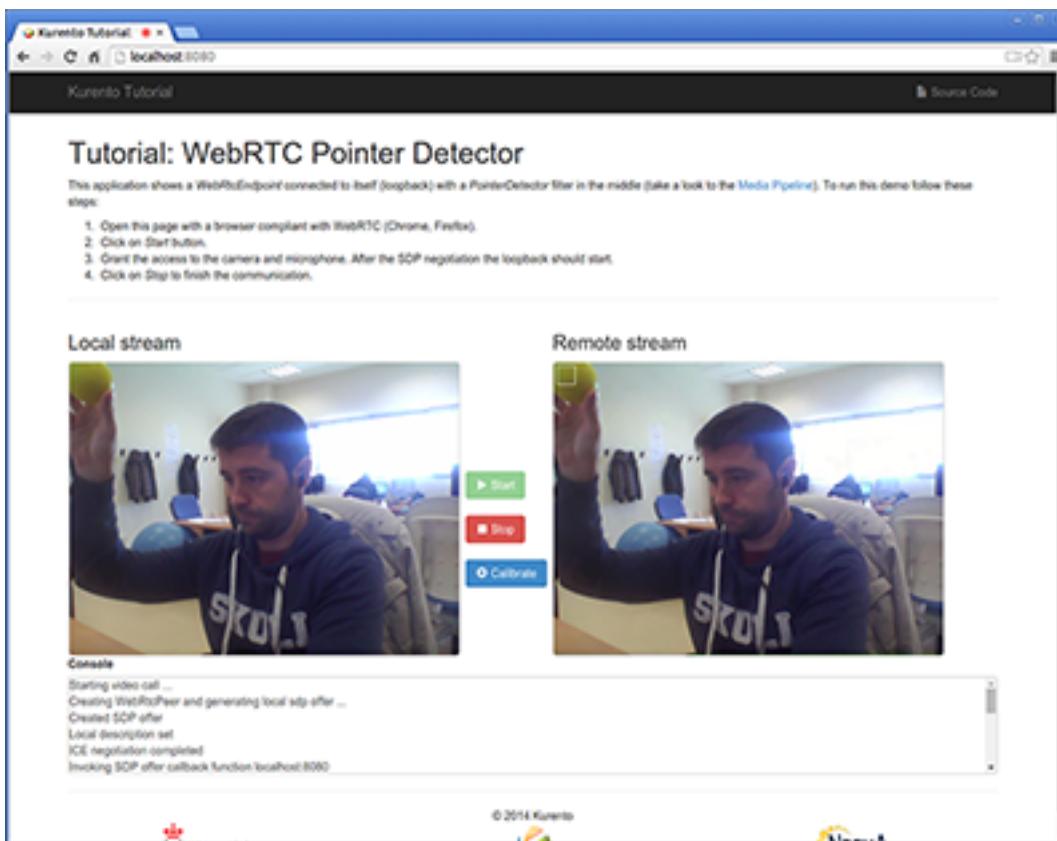


Fig. 16.7: *Pointer calibration stage*

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

In that precise moment, a calibration operation should be carried out. This is done by clicking on the *Calibrate* blue button of the GUI.

After that, the color of the pointer is tracked in real time by Kurento Media Server. `PointerDetectorFilter` can also define regions in the screen called *windows* in which some actions are performed when the pointer is detected when the pointer enters (`WindowIn` event) and exits (`WindowOut` event) the windows. This is implemented in the JavaScript logic as follows:

```
...
kurentoClient.register('kurento-module-pointerdetector')
const PointerDetectorWindowMediaParam = kurentoClient.getComplexType('pointerdetector.
˓→PointerDetectorWindowMediaParam')
const WindowParam = kurentoClient.getComplexType('pointerdetector.
˓→WindowParam')
...

kurentoClient(args.ws_uri, function(error, client) {
    if (error) return onError(error);

    client.create('MediaPipeline', function(error, _pipeline) {
        if (error) return onError(error);

        pipeline = _pipeline;

        console.log("Got MediaPipeline");

        pipeline.create('WebRtcEndpoint', function(error, webRtc) {
            if (error) return onError(error);

            console.log("Got WebRtcEndpoint");

            setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

            webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {
                if (error) return onError(error);

                console.log("SDP answer obtained. Processing ...");

                webRtc.gatherCandidates(onError);
                webRtcPeer.processAnswer(sdpAnswer);
            });

            var options =
            {
                calibrationRegion: WindowParam({
                    topRightCornerX: 5,
                    topRightCornerY: 5,
                    width: 30,
                    height: 30
                })
            };

            pipeline.create('pointerdetector.PointerDetectorFilter', options,_
˓→function(error, _filter) {
```

```

if (error) return onError(error);

filter = _filter;

var options = PointerDetectorWindowMediaParam({
  id: 'window0',
  height: 50,
  width: 50,
  upperRightX: 500,
  upperRightY: 150
});

filter.addWindow(options, onError);

var options = PointerDetectorWindowMediaParam({
  id: 'window1',
  height: 50,
  width: 50,
  upperRightX: 500,
  upperRightY: 250
});

filter.addWindow(options, onError);

filter.on ('WindowIn', function (data){
  console.log ("Event window in detected in window " + data.windowId);
});

filter.on ('WindowOut', function (data){
  console.log ("Event window out detected in window " + data.windowId);
});

console.log("Connecting ...");
client.connect(webRtc, filter, webRtc, function(error) {
  if (error) return onError(error);

  console.log("WebRtcEndpoint --> Filter --> WebRtcEndpoint");
});
});
});
});
});
});
```

The following picture illustrates the pointer tracking in one of the defined windows:

In order to carry out the calibration process, this JavaScript function is used:

```

function calibrate() {
  if(filter) filter.trackColorFromCalibrationRegion(onError);
}

function onError(error) {
  if(error) console.error(error);
}
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

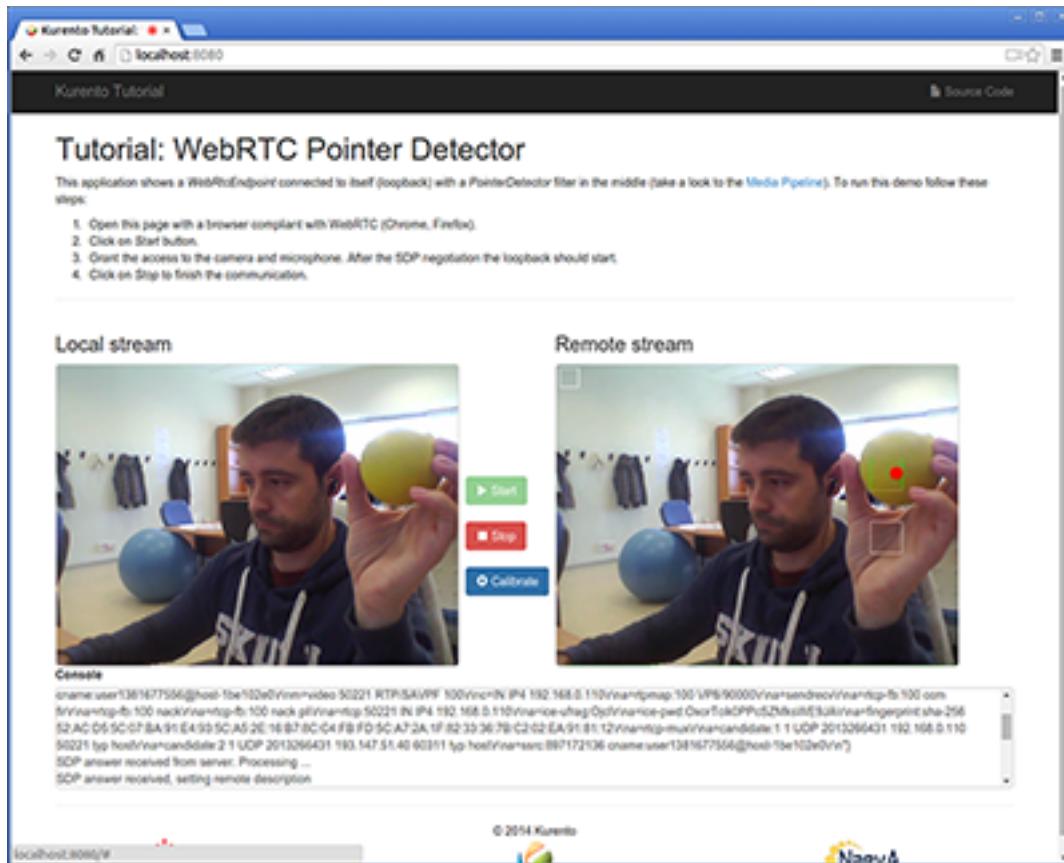


Fig. 16.8: Pointer tracking over a window

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},{  
  "urls":"stun:stun2.example.net"}]  
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",  
  "username":"user","credential":"myPassword"}]
```

Dependencies

The dependencies of this demo has to be obtained using [Bower](#). The definition of these dependencies are defined in the `bower.json` file, as follows:

```
"dependencies": {  
  "kurento-client": "6.7.1",  
  "kurento-utils": "6.7.1"  
  "kurento-module-pointerdetector": "6.7.1"  
}
```

To get these dependencies, just run the following shell command:

```
bower install
```

Note: We are in active development. You can find the latest versions at [Bower](#).

16.1.3 Node.js Module - Pointer Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a pointer tracking filter element.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-pointerdetector` should be also installed:

```
sudo apt-get install kms-pointerdetector
```

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -  
sudo apt-get install -y nodejs  
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git  
cd kurento-tutorial-node/kurento-pointerdetector  
git checkout 6.7.1  
npm install
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules  
npm cache clean
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a pointer in a WebRTC stream based on color tracking.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

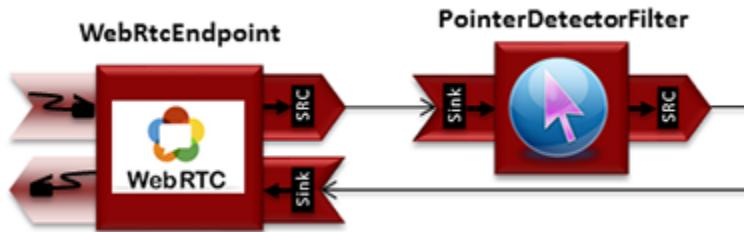


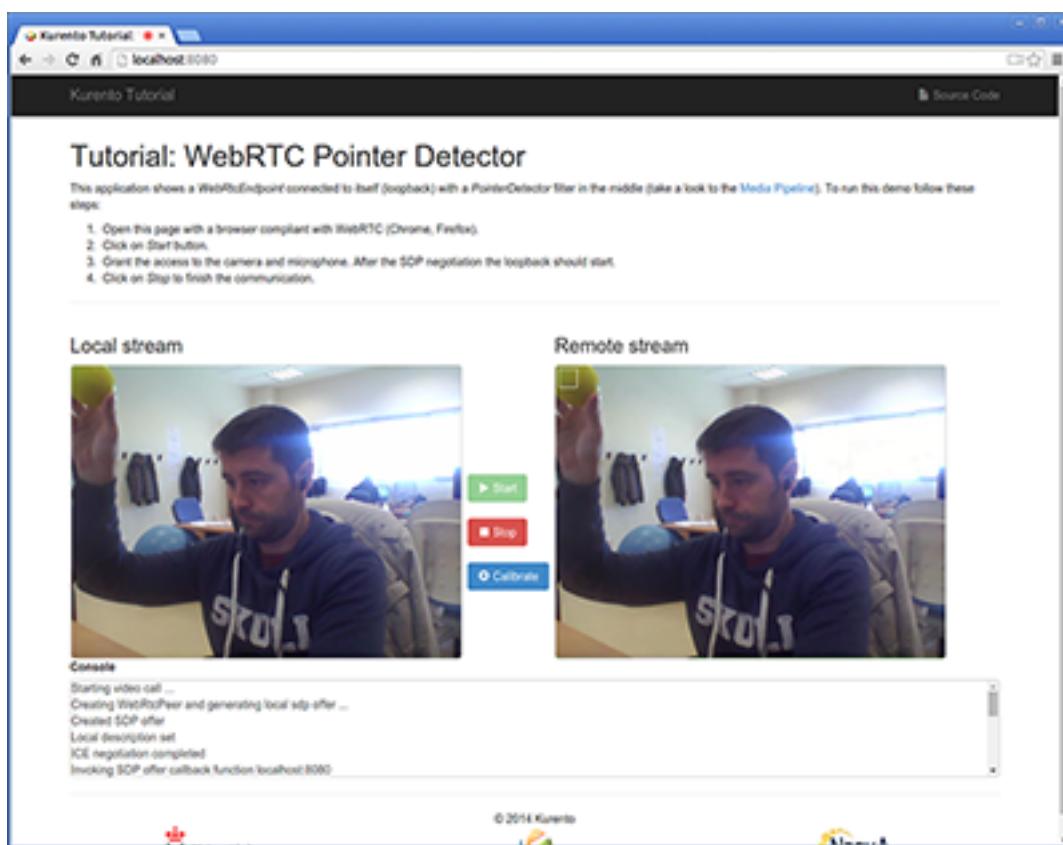
Fig. 16.9: WebRTC with PointerDetector filter in loopback Media Pipeline

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **PointerDetector** instead of **FaceOverlay** filter.

In order to perform pointer detection, there must be a calibration stage, in which the color of the pointer is registered by the filter. To accomplish this step, the pointer should be placed in a square in the upper left corner of the video, as follows:

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

Fig. 16.10: *Pointer calibration stage*

In that precise moment, a calibration operation should be carried out. This is done by clicking on the *Calibrate* blue button of the GUI.

After that, the color of the pointer is tracked in real time by Kurento Media Server. PointerDetectorFilter can also define regions in the screen called *windows* in which some actions are performed when the pointer is detected when the pointer enters (WindowIn event) and exits (WindowOut event) the windows. This is implemented in the JavaScript logic as follows:

```
...
kurento.register('kurento-module-pointerdetector');
const PointerDetectorWindowMediaParam = kurento.getComplexType('pointerdetector.
↪PointerDetectorWindowMediaParam');
const WindowParam = kurento.getComplexType('pointerdetector.
↪WindowParam');
...

function start(sessionId, ws, sdpOffer, callback) {
    if (!sessionId) {
        return callback('Cannot use undefined sessionId');
    }

    getKurentoClient(function(error, kurentoClient) {
        if (error) {
            return callback(error);
        }

        kurentoClient.create('MediaPipeline', function(error, pipeline) {
            if (error) {
                return callback(error);
            }

            createMediaElements(pipeline, ws, function(error, webRtcEndpoint, filter)
↪{
                if (error) {
                    pipeline.release();
                    return callback(error);
                }

                if (candidatesQueue[sessionId]) {
                    while(candidatesQueue[sessionId].length) {
                        var candidate = candidatesQueue[sessionId].shift();
                        webRtcEndpoint.addIceCandidate(candidate);
                    }
                }

                connectMediaElements(webRtcEndpoint, filter, function(error) {
                    if (error) {
                        pipeline.release();
                        return callback(error);
                    }

                    webRtcEndpoint.on('OnIceCandidate', function(event) {
                        var candidate = kurento.getComplexType('IceCandidate')(event.
↪candidate);
                        ws.send(JSON.stringify({
                            id : 'iceCandidate',
                            candidate : candidate
                        }));
                    });
                });
            }
        });
    });
}

function getKurentoClient(callback) {
    const options = {
        host: 'localhost',
        port: 8080
    };

    kurento.connect(options, function(error, kurentoClient) {
        if (error) {
            return callback(error);
        }

        callback(null, kurentoClient);
    });
}
```

```

        filter.on('WindowIn', function (_data) {
            return callback(null, 'WindowIn', _data);
        });

        filter.on('WindowOut', function (_data) {
            return callback(null, 'WindowOut', _data);
        });

        var options1 = PointerDetectorWindowMediaParam({
            id: 'window0',
            height: 50,
            width: 50,
            upperRightX: 500,
            upperRightY: 150
        });
        filter.addWindow(options1, function(error) {
            if (error) {
                pipeline.release();
                return callback(error);
            }
        });

        var options2 = PointerDetectorWindowMediaParam({
            id: 'window1',
            height: 50,
            width: 50,
            upperRightX: 500,
            upperRightY: 250
        });
        filter.addWindow(options2, function(error) {
            if (error) {
                pipeline.release();
                return callback(error);
            }
        });

        webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            sessions[sessionId] = {
                'pipeline' : pipeline,
                'webRtcEndpoint' : webRtcEndpoint,
                'pointerDetector' : filter
            }
            return callback(null, 'sdpAnswer', sdpAnswer);
        });

        webRtcEndpoint.gatherCandidates(function(error) {
            if (error) {
                return callback(error);
            }
        });
    });
});

```

```

        });
    });
}

function createMediaElements(pipeline, ws, callback) {
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            return callback(error);
        }

        var options = {
            calibrationRegion: WindowParam({
                topRightCornerX: 5,
                topRightCornerY: 5,
                width: 30,
                height: 30
            })
        };

        pipeline.create('pointerdetector.PointerDetectorFilter', options,_
        ↵function(error, filter) {
            if (error) {
                return callback(error);
            }

            return callback(null, webRtcEndpoint, filter);
        });
    });
}

```

The following picture illustrates the pointer tracking in one of the defined windows:

In order to carry out the calibration process, this JavaScript function is used:

```

function calibrate() {
    if (webRtcPeer) {
        console.log("Calibrating...");
        var message = {
            id: 'calibrate'
        };
        sendMessage(message);
    }
}

```

Dependencies

Dependencies of this demo are managed using NPM. Our main dependency is the Kurento Client JavaScript (*kurento-client*). The relevant part of the `package.json` file for managing this dependency is:

```

"dependencies": {
    "kurento-client": "6.7.1"
}

```

At the client side, dependencies are managed using Bower. Take a look to the `bower.json` file and pay attention to the following section:

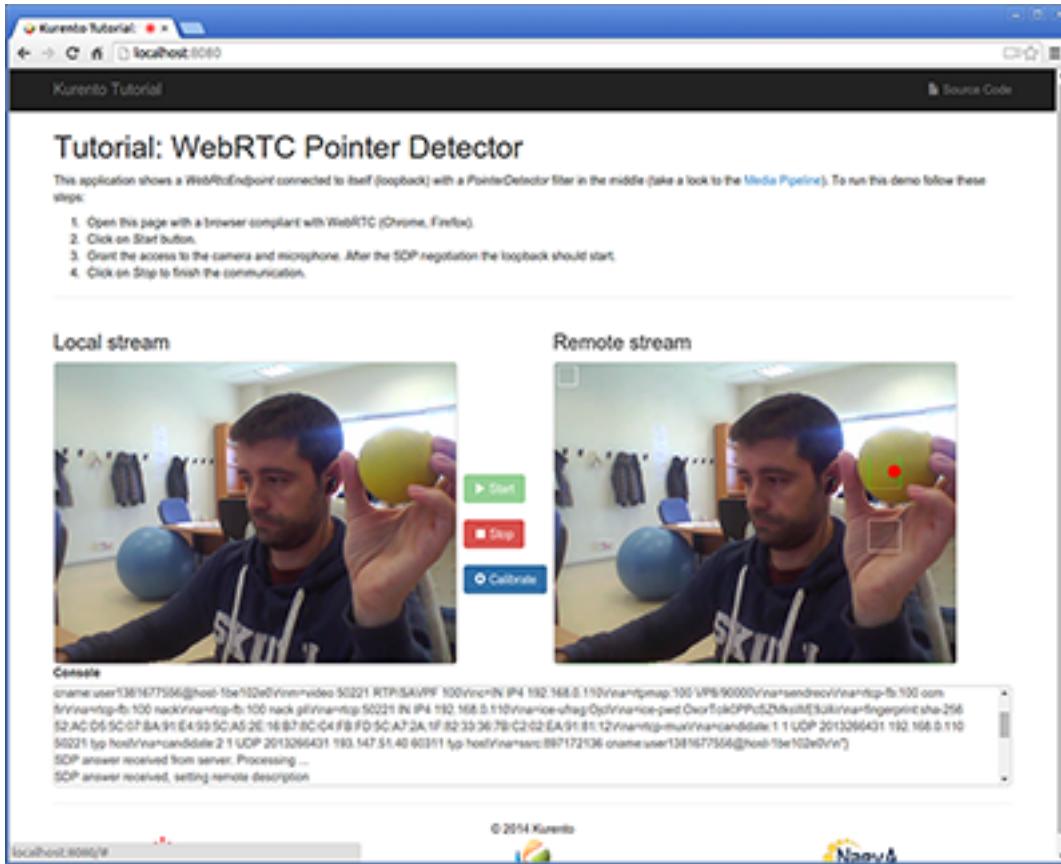


Fig. 16.11: Pointer tracking over a window

```
"dependencies": {
    "kurento-utils" : "6.7.1",
    "kurento-module-pointerdetector": "6.7.1"
}
```

Note: We are in active development. You can find the latest versions at [npm](#) and [Bower](#).

16.2 Module Tutorial - Chroma Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

16.2.1 Java Module - Chroma Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-chroma` should be also installed:

```
sudo apt-get install kms-chroma
```

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-chroma
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a chroma in a WebRTC stream based on color tracking.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

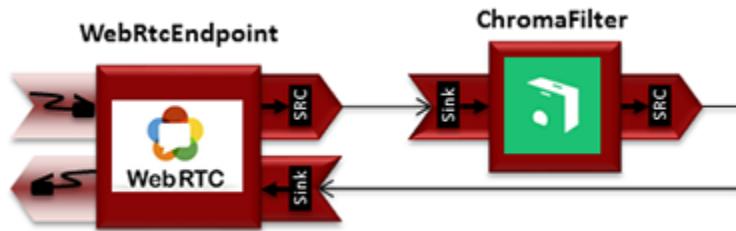


Fig. 16.12: *WebRTC with Chroma filter Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **Chroma** instead of **FaceOverlay** filter.

In order to perform chroma detection, there must be a color calibration stage. To accomplish this step, at the beginning of the demo, a little square appears in upper left of the video, as follows:

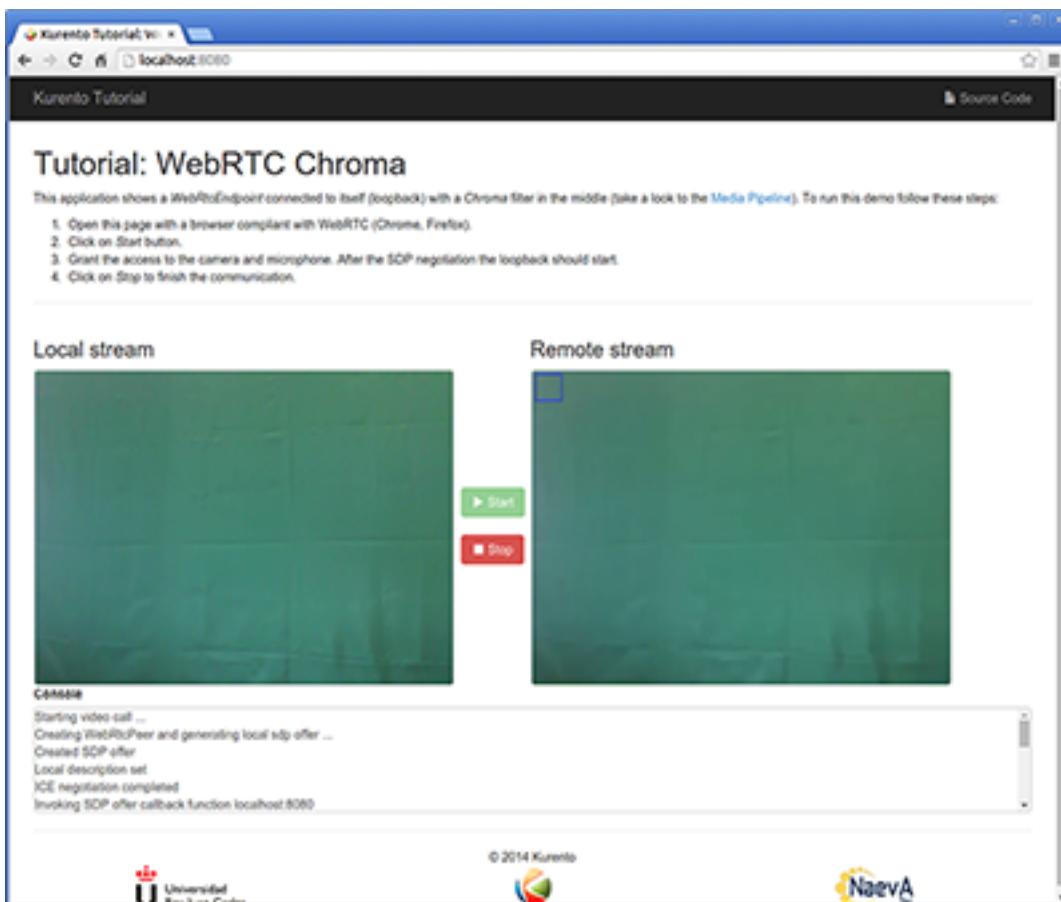


Fig. 16.13: *Chroma calibration stage*

In the first second of the demo, a calibration process is done, by detecting the color inside that square. When the calibration is finished, the square disappears and the chroma is substituted with the configured image. Take into account that this process requires good lighting condition. Otherwise the chroma substitution will not be perfect. This behavior can be seen in the upper right corner of the following screenshot:

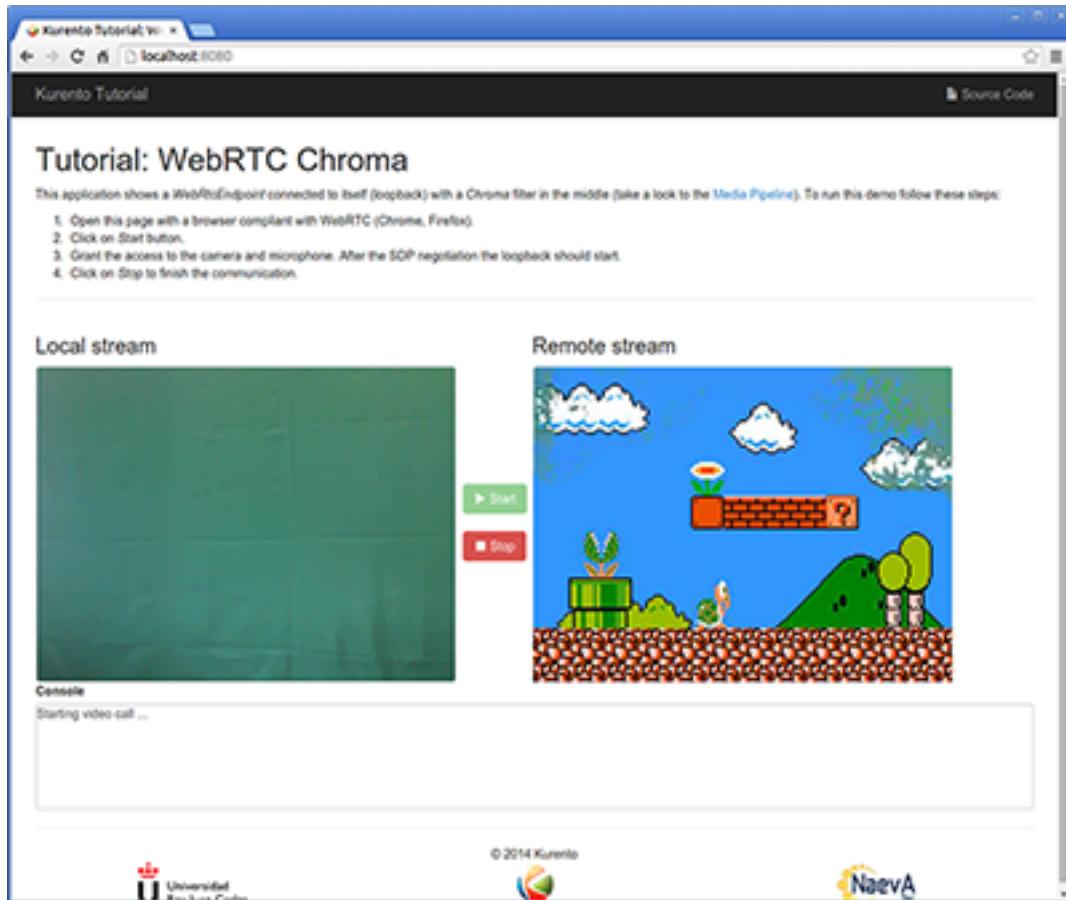


Fig. 16.14: *Chroma filter in action*

The media pipeline of this demo is implemented in the server-side logic as follows:

```
private void start(final WebSocketSession session, JSONObject jsonMessage) {
    try {
        // Media Logic (Media Pipeline and Elements)
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline)
            .build();
        user.setWebRtcEndpoint(webRtcEndpoint);
        users.put(session.getId(), user);

        webRtcEndpoint
            .addIceCandidateFoundListener(new EventListener<IceCandidateFoundEvent>()
        ↵{
            @Override
            public void onEvent(IceCandidateFoundEvent event) {

```

```

        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils
            .toJsonObject(event.getCandidate()));
    try {
        synchronized (session) {
            session.sendMessage(new TextMessage(
                response.toString()));
        }
    } catch (IOException e) {
        log.debug(e.getMessage());
    }
}
});

ChromaFilter chromaFilter = new ChromaFilter.Builder(pipeline,
    new WindowParam(5, 5, 40, 40)).build();
String appServerUrl = System.getProperty("app.server.url",
    ChromaApp.DEFAULT_APP_SERVER_URL);
chromaFilter.setBackground(appServerUrl + "/img/mario.jpg");

webRtcEndpoint.connect(chromaFilter);
chromaFilter.connect(webRtcEndpoint);

// SDP negotiation (offer and answer)
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);

synchronized (session) {
    session.sendMessage(new TextMessage(response.toString()));
}
webRtcEndpoint.gatherCandidates();

} catch (Throwable t) {
    sendError(session, t.getMessage());
}
}
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with webjars:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>

```

```
<groupId>org.kurento</groupId>
<artifactId>kurento-utils-js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>bootstrap</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>demo-console</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>adapter.js</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>jquery</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

16.2.2 JavaScript Module - Chroma Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-chroma` should be also installed:

```
sudo apt-get install kms-chroma
```

Be sure to have installed *Node.js* and *Bower* in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-chroma
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. Kurento Media Server must use WebSockets over SSL/TLS (WSS), so make sure you check [this](#) too. It is possible to locate the KMS in other machine simple adding the parameter `ws_uri` to the URL:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

This application uses computer vision and augmented reality techniques to detect a chroma in a WebRTC stream based on color tracking.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a [Media Pipeline](#) composed by the following [Media Element](#)s:

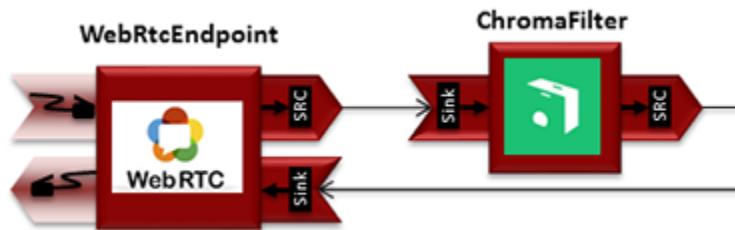


Fig. 16.15: *WebRTC with Chroma filter Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **Chroma** instead of **FaceOverlay** filter.

In order to perform chroma detection, there must be a color calibration stage. To accomplish this step, at the beginning of the demo, a little square appears in upper left of the video, as follows:

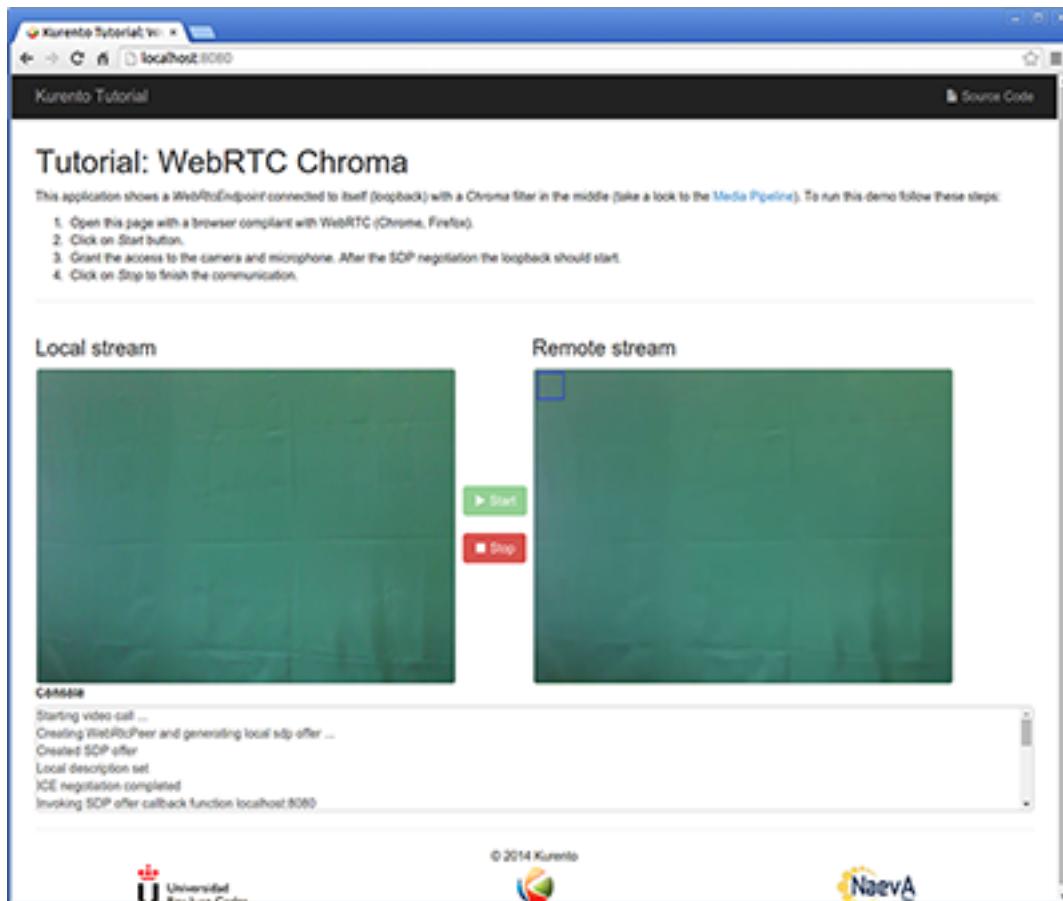


Fig. 16.16: *Chroma calibration stage*

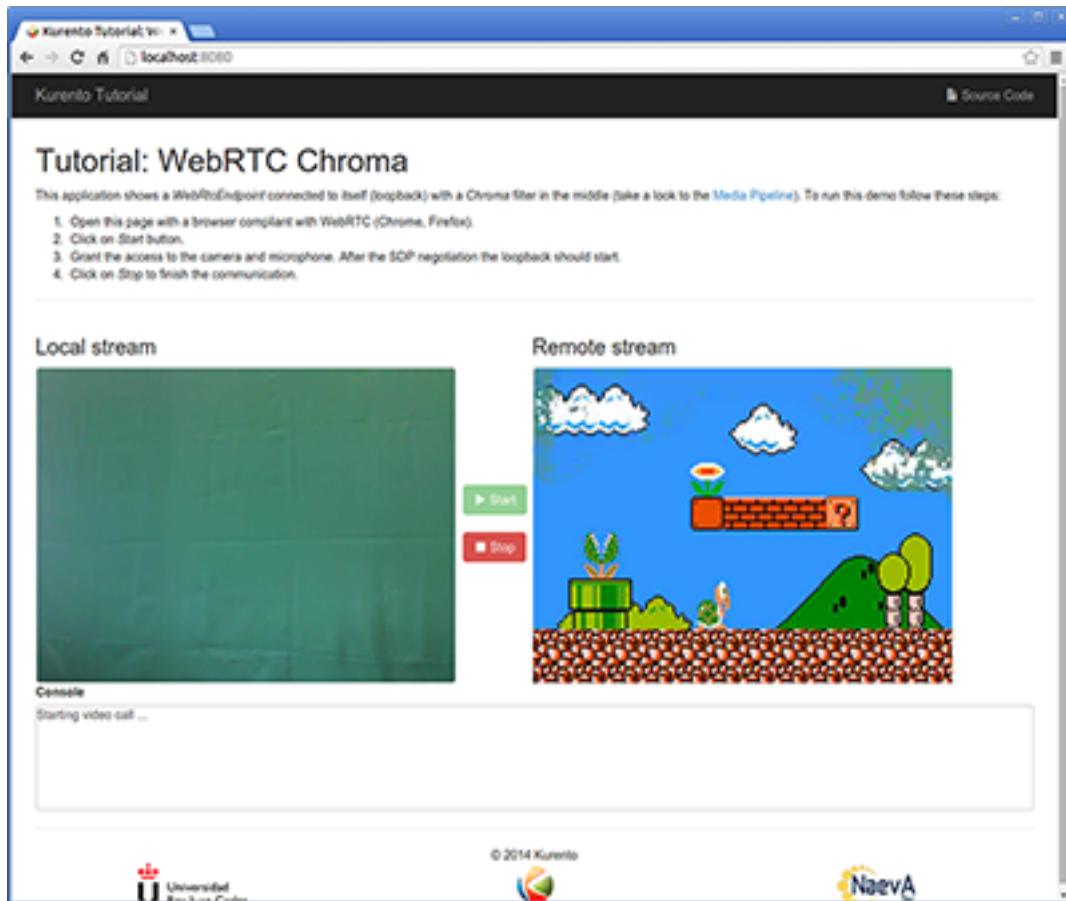
In the first second of the demo, a calibration process is done, by detecting the color inside that square. When the calibration is finished, the square disappears and the chroma is substituted with the configured image. Take into account that this process requires lighting condition. Otherwise the chroma substitution will not be perfect. This behavior can be seen in the upper right corner of the following screenshot:

Note: Modules can have options. For configure these options, you need get the constructor to them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

The media pipeline of this demo is implemented in the JavaScript logic as follows:

```
...
kurentoClient.register('kurento-module-chroma')
const WindowParam = kurentoClient.getComplexType('chroma.WindowParam')
...

kurentoClient(args.ws_uri, function(error, client) {
  if (error) return onError(error);
```

Fig. 16.17: *Chroma filter in action*

```
client.create('MediaPipeline', function(error, _pipeline) {
    if (error) return onError(error);

    pipeline = _pipeline;

    console.log("Got MediaPipeline");

    pipeline.create('WebRtcEndpoint', function(error, webRtc) {
        if (error) return onError(error);

        setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

        webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {
            if (error) return onError(error);

            console.log("SDP answer obtained. Processing...");

            webRtc.gatherCandidates(onError);
            webRtcPeer.processAnswer(sdpAnswer);
        });

        console.log("Got WebRtcEndpoint");

        var options =
        {
            window: WindowParam({
                topRightCornerX: 5,
                topRightCornerY: 5,
                width: 30,
                height: 30
            })
        }

        pipeline.create('chroma.ChromaFilter', options, function(error, filter) {
            if (error) return onError(error);

            console.log("Got Filter");

            filter.setBackground(args.bg_uri, function(error) {
                if (error) return onError(error);

                console.log("Set Image");
            });

            client.connect(webRtc, filter, webRtc, function(error) {
                if (error) return onError(error);

                console.log("WebRtcEndpoint --> filter --> WebRtcEndpoint");
            });
        });
    });
});
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},{  
  "urls":"stun:stun2.example.net"}]  
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",  
  "username":"user","credential":"myPassword"}]
```

Dependencies

The dependencies of this demo has to be obtained using [Bower](#). The definition of these dependencies are defined in the `bower.json` file, as follows:

```
"dependencies": {  
  "kurento-client": "6.7.1",  
  "kurento-utils": "6.7.1"  
  "kurento-module-pointerdetector": "6.7.1"  
}
```

To get these dependencies, just run the following shell command:

```
bower install
```

Note: We are in active development. You can find the latest versions at [Bower](#).

16.2.3 Node.js Module - Chroma Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-chroma` should be also installed:

```
sudo apt-get install kms-chroma
```

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -  
sudo apt-get install -y nodejs  
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git  
cd kurento-tutorial-node/kurento-chroma  
git checkout 6.7.1  
npm install
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules  
npm cache clean
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a chroma in a WebRTC stream based on color tracking.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

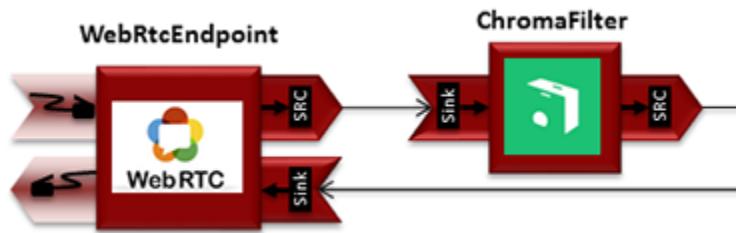


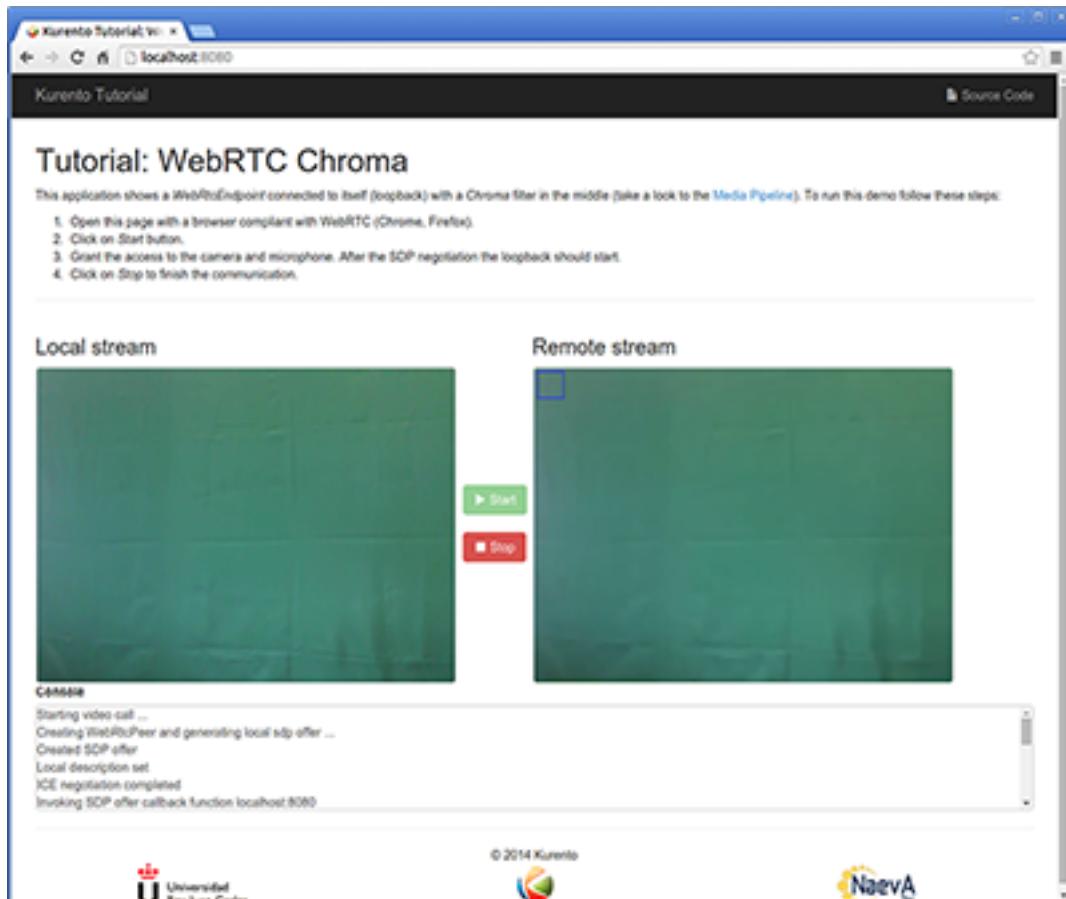
Fig. 16.18: *WebRTC with Chroma filter Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **Chroma** instead of **FaceOverlay** filter.

In order to perform chroma detection, there must be a color calibration stage. To accomplish this step, at the beginning of the demo, a little square appears in upper left of the video, as follows:

In the first second of the demo, a calibration process is done, by detecting the color inside that square. When the calibration is finished, the square disappears and the chroma is substituted with the configured image. Take into account that this process requires lighting condition. Otherwise the chroma substitution will not be perfect. This behavior can be seen in the upper right corner of the following screenshot:

Fig. 16.19: *Chroma calibration stage*

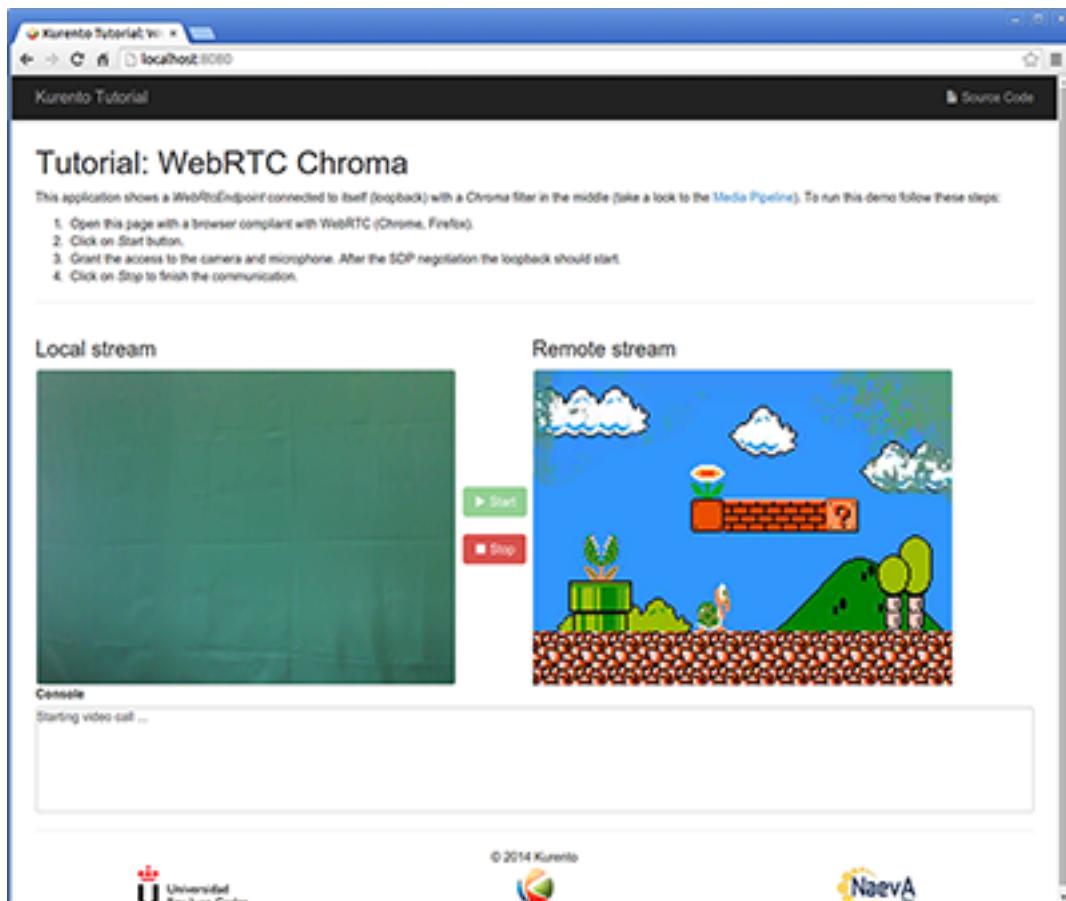


Fig. 16.20: *Chroma filter in action*

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

The media pipeline of this demo is implemented in the JavaScript logic as follows:

```
...
kurento.register('kurento-module-chroma');
...

function start(sessionId, ws, sdpOffer, callback) {
    if (!sessionId) {
        return callback('Cannot use undefined sessionId');
    }

    getKurentoClient(function(error, kurentoClient) {
        if (error) {
            return callback(error);
        }

        kurentoClient.create('MediaPipeline', function(error, pipeline) {
            if (error) {
                return callback(error);
            }

            createMediaElements(pipeline, ws, function(error, webRtcEndpoint, filter)
←{
                if (error) {
                    pipeline.release();
                    return callback(error);
                }

                if (candidatesQueue[sessionId]) {
                    while(candidatesQueue[sessionId].length) {
                        var candidate = candidatesQueue[sessionId].shift();
                        webRtcEndpoint.addIceCandidate(candidate);
                    }
                }

                connectMediaElements(webRtcEndpoint, filter, function(error) {
                    if (error) {
                        pipeline.release();
                        return callback(error);
                    }

                    webRtcEndpoint.on('OnIceCandidate', function(event) {
                        var candidate = kurento.getComplexType('IceCandidate')(event.
←candidate);
                        ws.send(JSON.stringify({
                            id : 'iceCandidate',
                            candidate : candidate
                        }));
                    });
                });

                webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
                    if (error) {
                        pipeline.release();
                
```

```
        return callback(error);
    }

    sessions[sessionId] = {
        'pipeline' : pipeline,
        'webRtcEndpoint' : webRtcEndpoint
    }
    return callback(null, sdpAnswer);
});

webRtcEndpoint.gatherCandidates(function(error) {
    if (error) {
        return callback(error);
    }
});
});

};

function createMediaElements(pipeline, ws, callback) {
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            return callback(error);
        }

        var options = {
            window: kurento.getComplexType('chroma.WindowParam')({
                topRightCornerX: 5,
                topRightCornerY: 5,
                width: 30,
                height: 30
            })
        }
        pipeline.create('chroma.ChromaFilter', options, function(error, filter) {
            if (error) {
                return callback(error);
            }

            return callback(null, webRtcEndpoint, filter);
        });
    });
}

function connectMediaElements(webRtcEndpoint, filter, callback) {
    webRtcEndpoint.connect(filter, function(error) {
        if (error) {
            return callback(error);
        }

        filter.setBackground(url.format(asUrl) + 'img/mario.jpg', function(error) {
            if (error) {
                return callback(error);
            }

            filter.connect(webRtcEndpoint, function(error) {
                if (error) {

```

```

        return callback(error);
    }

    return callback(null);
});;
});;
});
}

```

Dependencies

Dependencies of this demo are managed using NPM. Our main dependency is the Kurento Client JavaScript (*kurento-client*). The relevant part of the `package.json` file for managing this dependency is:

```

"dependencies": {
  "kurento-client" : "6.7.1"
}

```

At the client side, dependencies are managed using Bower. Take a look to the `bower.json` file and pay attention to the following section:

```

"dependencies": {
  "kurento-utils" : "6.7.1",
  "kurento-module-pointerdetector": "6.7.1"
}

```

Note: We are in active development. You can find the latest versions at [npm](#) and [Bower](#).

16.3 Module Tutorial - Crowd Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects people agglomeration in video streams.

16.3.1 Java Module - Crowd Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects clusters of people in video streams.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-crowddetector` should be also installed:

```
sudo apt-get install kms-crowddetector
```

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-crowddetector
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a crowd in a WebRTC stream.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

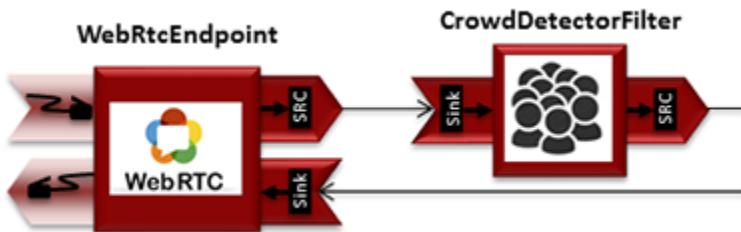


Fig. 16.21: *WebRTC with crowdDetector filter Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **CrowdDetector** instead of **FaceOverlay** filter.

To setup a **CrowdDetectorFilter**, first we need to define one or more *regions of interest* (ROIs). A ROI determines the zone within the video stream, which are going to be monitored and analised by the filter. To define a ROI, we need to configure at least three points. These points are defined in relative terms (0 to 1) to the video width and height.

CrowdDetectorFilter performs two actions in the defined ROIs. On one hand, the detected crowds are colored over the stream. On the other hand, different events are raised to the client.

To understand crowd coloring, we can take a look to an screenshot of a running example of **CrowdDetectorFilter**. In the picture below, we can see that there are two ROIs (bounded with white lines in the video). On these ROIs, we can see two different colors over the original video stream: red zones are drawn over detected static crowds (or moving slowly). Blue zones are drawn over the detected crowds moving fast.



Fig. 16.22: *Crowd detection sample*

Regarding crowd events, there are three types of events, namely:

- CrowdDetectorFluidityEvent. Event raised when a certain level of fluidity is detected in a ROI. Fluidity can be seen as the level of general movement in a crowd.
- CrowdDetectorOccupancyEvent. Event raised when a level of occupancy is detected in a ROI. Occupancy can be seen as the level of agglomeration in stream.
- CrowdDetectorDirectionEvent. Event raised when a movement direction is detected in a ROI by a crowd.

Both fluidity as occupancy are quantified in a relative metric from 0 to 100%. Then, both attributes are qualified into three categories: i) Minimum (min); ii) Medium (med); iii) Maximum (max).

Regarding direction, it is quantified as an angle (0-360°), where 0 is the direction from the central point of the video to the top (i.e., north), 90 correspond to the direction to the right (east), 180 is the south, and finally 270 is the west.

With all these concepts, now we can check out the Java server-side code of this demo. As depicted in the snippet below, we create a ROI by adding `RelativePoint` instances to a list. Each ROI is then stored into a list of `RegionOfInterest` instances.

Then, each ROI should be configured. To do that, we have the following methods:

- `setFluidityLevelMin`: Fluidity level (0-100%) for the category *minimum*.
- `setFluidityLevelMed`: Fluidity level (0-100%) for the category *medium*.
- `setFluidityLevelMax`: Fluidity level (0-100%) for the category *maximum*.
- `setFluidityNumFramesToEvent`: Number of consecutive frames detecting a fluidity level to rise a event.
- `setOccupancyLevelMin`: Occupancy level (0-100%) for the category *minimum*.
- `setOccupancyLevelMed`: Occupancy level (0-100%) for the category *medium*.
- `setOccupancyLevelMax`: Occupancy level (0-100%) for the category *maximum*.
- `setOccupancyNumFramesToEvent`: Number of consecutive frames detecting a occupancy level to rise a event.
- `setSendOpticalFlowEvent`: Boolean value that indicates whether or not directions events are going to be tracked by the filter. Be careful with this feature, since it is very demanding in terms of resource usage (CPU, memory) in the media server. Set to true this parameter only when you are going to need directions events in your client-side.
- `setOpticalFlowNumFramesToEvent`: Number of consecutive frames detecting a direction level to rise a event.
- `setOpticalFlowNumFramesToReset`: Number of consecutive frames detecting a occupancy level in which the counter is reset.
- `setOpticalFlowAngleOffset`: Counterclockwise offset of the angle. This parameters is useful to move the default axis for directions (0°=north, 90°=east, 180°=south, 270°=west).

All in all, the media pipeline of this demo is implemented as follows:

```
// Media Logic (Media Pipeline and Elements)
MediaPipeline pipeline = kurento.createMediaPipeline();
pipelines.put(session.getId(), pipeline);

WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline)
    .build();
webRtcEndpoint
    .addIceCandidateFoundListener(new EventListener<IceCandidateFoundEvent>() {
        @Override
        public void onEvent(IceCandidateFoundEvent event) {
```

```

JsonObject response = new JsonObject();
response.addProperty("id", "iceCandidate");
response.add("candidate",
    JsonUtils.toJsonObject(event.getCandidate()));
try {
    synchronized (session) {
        session.sendMessage(new TextMessage(response
            .toString()));
    }
} catch (IOException e) {
    log.debug(e.getMessage());
}
}
});
});

List<RegionOfInterest> rois = new ArrayList<>();
List<RelativePoint> points = new ArrayList<RelativePoint>();

points.add(new RelativePoint(0, 0));
points.add(new RelativePoint(0.5F, 0));
points.add(new RelativePoint(0.5F, 0.5F));
points.add(new RelativePoint(0, 0.5F));

RegionOfInterestConfig config = new RegionOfInterestConfig();

config.setFluidityLevelMin(10);
config.setFluidityLevelMed(35);
config.setFluidityLevelMax(65);
config.setFluidityNumFramesToEvent(5);
config.setOccupancyLevelMin(10);
config.setOccupancyLevelMed(35);
config.setOccupancyLevelMax(65);
config.setOccupancyNumFramesToEvent(5);
config.setSendOpticalFlowEvent(false);
config.setOpticalFlowNumFramesToEvent(3);
config.setOpticalFlowNumFramesToReset(3);
config.setOpticalFlowAngleOffset(0);

rois.add(new RegionOfInterest(points, config, "roi0"));

CrowdDetectorFilter crowdDetectorFilter = new CrowdDetectorFilter.Builder(
    pipeline, rois).build();

webRtcEndpoint.connect(crowdDetectorFilter);
crowdDetectorFilter.connect(webRtcEndpoint);

// addEventListener to crowddetector
crowdDetectorFilter.addCrowdDetectorDirectionListener(
    new EventListener<CrowdDetectorDirectionEvent>() {
        @Override
        public void onEvent(CrowdDetectorDirectionEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "directionEvent");
            response.addProperty("roiId", event.getRoiID());
            response.addProperty("angle",
                event.getDirectionAngle());
            try {
                session.sendMessage(new TextMessage(response

```

```

        .toString());
    } catch (Throwable t) {
        sendError(session, t.getMessage());
    }
}
});

crowdDetectorFilter.addCrowdDetectorFluidityListener(
    new EventListener<CrowdDetectorFluidityEvent>() {
        @Override
        public void onEvent(CrowdDetectorFluidityEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "fluidityEvent");
            response.addProperty("roiId", event.getRoiID());
            response.addProperty("level",
                event.getFluidityLevel());
            response.addProperty("percentage",
                event.getFluidityPercentage());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });
}

crowdDetectorFilter.addCrowdDetectorOccupancyListener(
    new EventListener<CrowdDetectorOccupancyEvent>() {
        @Override
        public void onEvent(CrowdDetectorOccupancyEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "occupancyEvent");
            response.addProperty("roiId", event.getRoiID());
            response.addProperty("level",
                event.getOccupancyLevel());
            response.addProperty("percentage",
                event.getOccupancyPercentage());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });
}

// SDP negotiation (offer and answer)
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);
session.sendMessage(new TextMessage(response.toString()));

webRtcEndpoint.gatherCandidates();

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side. Other client libraries are managed with webjars:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>jquery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>ekko-lightbox</artifactId>
    </dependency>
</dependencies>
```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

16.3.2 JavaScript Module - Crowd Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects people agglomeration in video streams.

Note: This tutorial has been configured for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-crowddetector` should be also installed:

```
sudo apt-get install kms-crowddetector
```

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-crowddetector
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. Kurento Media Server must use WebSockets over SSL/TLS (WSS), so make sure you check [this](#) too. It is possible to locate the KMS in other machine simple adding the parameter `ws_uri` to the URL:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

This application uses computer vision and augmented reality techniques to detect a crowd in a WebRTC stream.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a [Media Pipeline](#) composed by the following [Media Element](#)s:

The complete source code of this demo can be found in [GitHub](#).

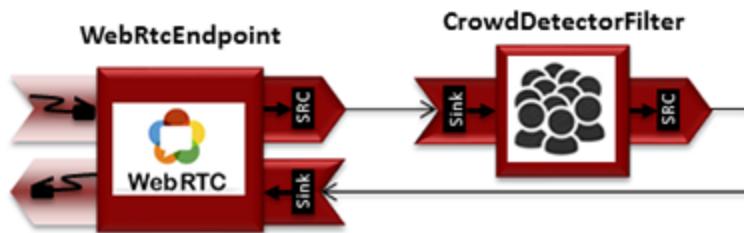


Fig. 16.23: *WebRTC with crowdDetector filter Media Pipeline*

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **CrowdDetector** instead of **FaceOverlay** filter.

To setup a **CrowdDetectorFilter**, first we need to define one or more *region of interests* (ROIs). A ROI delimits the zone within the video stream in which crowd are going to be tracked. To define a ROI, we need to configure at least three points. These points are defined in relative terms (0 to 1) to the video width and height.

CrowdDetectorFilter performs two actions in the defined ROIs. On the one hand, the detected crowd are colored over the stream. On the other hand, different events are raised to the client.

To understand crowd coloring, we can take a look to an screenshot of a running example of **CrowdDetectorFilter**. In the picture below, we can see that there are two ROIs (bounded with white lines in the video). On these ROIs, we can see two different colors over the original video stream: red zones are drawn over detected static crowds (or moving slowly). Blue zones are drawn over the detected crowds moving fast.



Fig. 16.24: *Crowd detection sample*

Regarding crowd events, there are three types of events, namely:

- CrowdDetectorFluidityEvent. Event raised when a certain level of fluidity is detected in a ROI. Fluidity can be seen as the level of general movement in a crowd.
- CrowdDetectorOccupancyEvent. Event raised when a level of occupancy is detected in a ROI. Occupancy can be seen as the level of agglomeration in stream.
- CrowdDetectorDirectionEvent. Event raised when a movement direction is detected in a ROI by a crowd.

Both fluidity as occupancy are quantified in a relative metric from 0 to 100%. Then, both attributes are qualified into three categories: i) Minimum (min); ii) Medium (med); iii) Maximum (max).

Regarding direction, it is quantified as an angle (0-360°), where 0 is the direction from the central point of the video to the top (i.e., north), 90 correspond to the direction to the right (east), 180 is the south, and finally 270 is the west.

With all these concepts, now we can check out the Java server-side code of this demo. As depicted in the snippet below, we create a ROI by adding `RelativePoint` instances to a list. Each ROI is then stored into a list of `RegionOfInterest` instances.

Then, each ROI should be configured. To do that, we have the following methods:

- `fluidityLevelMin`: Fluidity level (0-100%) for the category *minimum*.
- `fluidityLevelMed`: Fluidity level (0-100%) for the category *medium*.
- `fluidityLevelMax`: Fluidity level (0-100%) for the category *maximum*.
- `fluidityNumFramesToEvent`: Number of consecutive frames detecting a fluidity level to rise a event.
- `occupancyLevelMin`: Occupancy level (0-100%) for the category *minimum*.
- `occupancyLevelMed`: Occupancy level (0-100%) for the category *medium*.
- `occupancyLevelMax`: Occupancy level (0-100%) for the category *maximum*.
- `occupancyNumFramesToEvent`: Number of consecutive frames detecting a occupancy level to rise a event.
- `sendOpticalFlowEvent`: Boolean value that indicates whether or not directions events are going to be tracked by the filter. Be careful with this feature, since it is very demanding in terms of resource usage (CPU, memory) in the media server. Set to true this parameter only when you are going to need directions events in your client-side.
- `opticalFlowNumFramesToEvent`: Number of consecutive frames detecting a direction level to rise a event.
- `opticalFlowNumFramesToReset`: Number of consecutive frames detecting a occupancy level in which the counter is reset.
- `opticalFlowAngleOffset`: Counterclockwise offset of the angle. This parameters is useful to move the default axis for directions (0°=north, 90°=east, 180°=south, 270°=west).

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

All in all, the media pipeline of this demo is implemented as follows:

```
...
kurentoClient.register('kurento-module-crowddetector')
const RegionOfInterest = kurentoClient.getComplexType('crowddetector.
˓→RegionOfInterest')
```

```
const RegionOfInterestConfig = kurentoClient.getComplexType('crowddetector.  
RegionOfInterestConfig')  
const RelativePoint = kurentoClient.getComplexType('crowddetector.  
RelativePoint')  
...  
  
kurentoClient(args.ws_uri, function(error, client) {  
    if (error) return onError(error);  
  
    client.create('MediaPipeline', function(error, p) {  
        if (error) return onError(error);  
  
        pipeline = p;  
  
        console.log("Got MediaPipeline");  
  
        pipeline.create('WebRtcEndpoint', function(error, webRtc) {  
            if (error) return onError(error);  
  
            console.log("Got WebRtcEndpoint");  
  
            setIceCandidateCallbacks(webRtcPeer, webRtc, onError)  
  
            webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {  
                if (error) return onError(error);  
  
                console.log("SDP answer obtained. Processing ...");  
  
                webRtc.gatherCandidates(onError);  
  
                webRtcPeer.processAnswer(sdpAnswer);  
            });  
  
            var options =  
            {  
                rois:  
                [  
                    RegionOfInterest({  
                        id: 'roi1',  
                        points:  
                        [  
                            RelativePoint({x: 0, y: 0}),  
                            RelativePoint({x: 0.5, y: 0}),  
                            RelativePoint({x: 0.5, y: 0.5}),  
                            RelativePoint({x: 0, y: 0.5})  
                        ],  
                        regionOfInterestConfig: RegionOfInterestConfig({  
                            occupancyLevelMin: 10,  
                            occupancyLevelMed: 35,  
                            occupancyLevelMax: 65,  
                            occupancyNumFramesToEvent: 5,  
                            fluidityLevelMin: 10,  
                            fluidityLevelMed: 35,  
                            fluidityLevelMax: 65,  
                            fluidityNumFramesToEvent: 5,  
                            sendOpticalFlowEvent: false,  
                            opticalFlowNumFramesToEvent: 3,  
                            opticalFlowNumFramesToReset: 3,  
                        })  
                    })  
                ]  
            };  
        });  
    });  
});
```

```
        opticalFlowAngleOffset: 0
    })
})
]
}

pipeline.create('crowddetector.CrowdDetectorFilter', options, function(error, filter)
{
    if (error) return onError(error);

    console.log("Connecting...");

    filter.on('CrowdDetectorDirection', function (data){
        console.log("Direction event received in roi " + data.roiID +
            " with direction " + data.directionAngle);
    });

    filter.on('CrowdDetectorFluidity', function (data){
        console.log("Fluidity event received in roi " + data.roiID +
            ". Fluidity level " + data.fluidityPercentage +
            " and fluidity percentage " + data.fluidityLevel);
    });

    filter.on('CrowdDetectorOccupancy', function (data){
        console.log("Occupancy event received in roi " + data.roiID +
            ". Occupancy level " + data.occupancyPercentage +
            " and occupancy percentage " + data.occupancyLevel);
    });

    client.connect(webRtc, filter, webRtc, function(error){
        if (error) return onError(error);

        console.log("WebRtcEndpoint --> Filter --> WebRtcEndpoint");
    });
});
});
});
```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```
https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"},  
  ↵ "urls":"stun:stun2.example.net"}]  
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",  
  ↵ "username":"user", "credential":"myPassword"}]
```

Dependencies

The dependencies of this demo has to be obtained using [Bower](#). The definition of these dependencies are defined in the `bower.json` file, as follows:

```
"dependencies": {
  "kurento-client": "6.7.1",
  "kurento-utils": "6.7.1"
  "kurento-module-pointerdetector": "6.7.1"
}
```

To get these dependencies, just run the following shell command:

```
bower install
```

Note: We are in active development. You can find the latest versions at [Bower](#).

16.3.3 Node.js Module - Crowd Detector Filter

This web application consists on a [WebRTC](#) video communication in mirror (*loopback*) with a crowd detector filter. This filter detects people agglomeration in video streams.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-crowddetector` should be also installed:

```
sudo apt-get install kms-crowddetector
```

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-crowddetector
git checkout 6.7.1
npm install
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules
npm cache clean
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a crowd in a WebRTC stream.

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

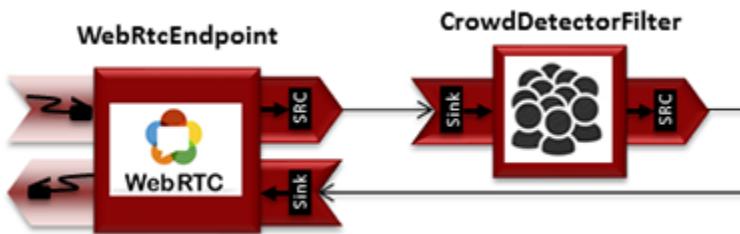


Fig. 16.25: *WebRTC with crowdDetector filter Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **CrowdDetector** instead of **FaceOverlay** filter.

To setup a **CrowdDetectorFilter**, first we need to define one or more *region of interests* (ROIs). A ROI delimits the zone within the video stream in which crowd are going to be tracked. To define a ROI, we need to configure at least three points. These points are defined in relative terms (0 to 1) to the video width and height.

CrowdDetectorFilter performs two actions in the defined ROIs. On the one hand, the detected crowd are colored over the stream. On the other hand, different events are raised to the client.

To understand crowd coloring, we can take a look to an screenshot of a running example of **CrowdDetectorFilter**. In the picture below, we can see that there are two ROIs (bounded with white lines in the video). On these ROIs, we can see two different colors over the original video stream: red zones are drawn over detected static crowds (or moving slowly). Blue zones are drawn over the detected crowds moving fast.

Regarding crowd events, there are three types of events, namely:

- **CrowdDetectorFluidityEvent**. Event raised when a certain level of fluidity is detected in a ROI. Fluidity can be seen as the level of general movement in a crowd.
- **CrowdDetectorOccupancyEvent**. Event raised when a level of occupancy is detected in a ROI. Occupancy can be seen as the level of agglomeration in stream.
- **CrowdDetectorDirectionEvent**. Event raised when a movement direction is detected in a ROI by a crowd.

Both fluidity as occupancy are quantified in a relative metric from 0 to 100%. Then, both attributes are qualified into three categories: i) Minimum (min); ii) Medium (med); iii) Maximum (max).



Fig. 16.26: *Crowd detection sample*

Regarding direction, it is quantified as an angle (0-360°), where 0 is the direction from the central point of the video to the top (i.e., north), 90 correspond to the direction to the right (east), 180 is the south, and finally 270 is the west.

With all these concepts, now we can check out the Java server-side code of this demo. As depicted in the snippet below, we create a ROI by adding `RelativePoint` instances to a list. Each ROI is then stored into a list of `RegionOfInterest` instances.

Then, each ROI should be configured. To do that, we have the following methods:

- `fluidityLevelMin`: Fluidity level (0-100%) for the category *minimum*.
- `fluidityLevelMed`: Fluidity level (0-100%) for the category *medium*.
- `fluidityLevelMax`: Fluidity level (0-100%) for the category *maximum*.
- `fluidityNumFramesToEvent`: Number of consecutive frames detecting a fluidity level to rise a event.
- `occupancyLevelMin`: Occupancy level (0-100%) for the category *minimum*.
- `occupancyLevelMed`: Occupancy level (0-100%) for the category *medium*.
- `occupancyLevelMax`: Occupancy level (0-100%) for the category *maximum*.
- `occupancyNumFramesToEvent`: Number of consecutive frames detecting a occupancy level to rise a event.
- `sendOpticalFlowEvent`: Boolean value that indicates whether or not directions events are going to be tracked by the filter. Be careful with this feature, since it is very demanding in terms of resource usage (CPU, memory) in the media server. Set to true this parameter only when you are going to need directions events in your client-side.
- `opticalFlowNumFramesToEvent`: Number of consecutive frames detecting a direction level to rise a event.
- `opticalFlowNumFramesToReset`: Number of consecutive frames detecting a occupancy level in which the counter is reset.
- `opticalFlowAngleOffset`: Counterclockwise offset of the angle. This parameters is useful to move the default axis for directions (0°=north, 90°=east, 180°=south, 270°=west).

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

All in all, the media pipeline of this demo is implemented as follows:

```
...
kurento.register('kurento-module-crowddetector');
const RegionOfInterest      = kurento.getComplexType('crowddetector.RegionOfInterest
');
const RegionOfInterestConfig = kurento.getComplexType('crowddetector.
RegionOfInterestConfig');
const RelativePoint          = kurento.getComplexType('crowddetector.RelativePoint');
...

function start(sessionId, ws, sdpOffer, callback) {
  if (!sessionId) {
    return callback('Cannot use undefined sessionId');
  }

  getKurentoClient(function(error, kurentoClient) {
    ...
  });
}
```

```

    if (error) {
        return callback(error);
    }

    kurentoClient.create('MediaPipeline', function(error, pipeline) {
        if (error) {
            return callback(error);
        }

        createMediaElements(pipeline, ws, function(error, webRtcEndpoint, filter)
←{
            if (error) {
                pipeline.release();
                return callback(error);
            }

            if (candidatesQueue[sessionId]) {
                while(candidatesQueue[sessionId].length) {
                    var candidate = candidatesQueue[sessionId].shift();
                    webRtcEndpoint.addIceCandidate(candidate);
                }
            }
        }

        connectMediaElements(webRtcEndpoint, filter, function(error) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            filter.on('CrowdDetectorDirection', function (_data){
                return callback(null, 'crowdDetectorDirection', _data);
            });

            filter.on('CrowdDetectorFluidity', function (_data){
                return callback(null, 'crowdDetectorFluidity', _data);
            });

            filter.on('CrowdDetectorOccupancy', function (_data){
                return callback(null, 'crowdDetectorOccupancy', _data);
            });

            webRtcEndpoint.on('OnIceCandidate', function(event) {
                var candidate = kurento.getComplexType('IceCandidate')(event.
←candidate);
                ws.send(JSON.stringify({
                    id : 'iceCandidate',
                    candidate : candidate
                }));
            });
        });

        webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            sessions[sessionId] = {
                'pipeline' : pipeline,

```

```

        'webRtcEndpoint' : webRtcEndpoint
    }
    return callback(null, 'sdpAnswer', sdpAnswer);
});

webRtcEndpoint.gatherCandidates(function(error) {
    if (error) {
        return callback(error);
    }
});
});
});
});
};

function createMediaElements(pipeline, ws, callback) {
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            return callback(error);
        }

        var options = {
            rois: [
                RegionOfInterest({
                    id: 'roi1',
                    points: [
                        RelativePoint({x: 0 , y: 0 }),
                        RelativePoint({x: 0.5, y: 0 }),
                        RelativePoint({x: 0.5, y: 0.5}),
                        RelativePoint({x: 0 , y: 0.5})
                    ],
                    regionOfInterestConfig: RegionOfInterestConfig({
                        occupancyLevelMin: 10,
                        occupancyLevelMed: 35,
                        occupancyLevelMax: 65,
                        occupancyNumFramesToEvent: 5,
                        fluidityLevelMin: 10,
                        fluidityLevelMed: 35,
                        fluidityLevelMax: 65,
                        fluidityNumFramesToEvent: 5,
                        sendOpticalFlowEvent: false,
                        opticalFlowNumFramesToEvent: 3,
                        opticalFlowNumFramesToReset: 3,
                        opticalFlowAngleOffset: 0
                    })
                })
            ]
        }
        pipeline.create('crowddetector.CrowdDetectorFilter', options, function(error, filter) {
            if (error) {
                return callback(error);
            }

            return callback(null, webRtcEndpoint, filter);
        });
    });
}

```

```
}
```

Dependencies

Dependencies of this demo are managed using NPM. Our main dependency is the Kurento Client JavaScript (*kurento-client*). The relevant part of the `package.json` file for managing this dependency is:

```
"dependencies": {
  "kurento-client" : "6.7.1"
}
```

At the client side, dependencies are managed using Bower. Take a look to the `bower.json` file and pay attention to the following section:

```
"dependencies": {
  "kurento-utils" : "6.7.1",
  "kurento-module-pointerdetector": "6.7.1"
}
```

Note: We are in active development. You can find the latest versions at [npm](#) and [Bower](#).

16.4 Module Tutorial - Plate Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

16.4.1 Java Module - Plate Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

Note: This tutorial has been configured to use https. Follow the [instructions](#) to secure your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-platedetector` should be also installed:

```
sudo apt-get install kms-platedetector
```

Warning: Plate detector module is a prototype and its results is not always accurate. Consider this if you are planning to use this module in a production environment.

To launch the application, you need to clone the GitHub project where this demo is hosted, and then run the main class:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-platedetector
git checkout 6.7.1
mvn compile exec:java
```

The web application starts on port 8443 in the localhost by default. Therefore, open the URL <https://localhost:8443/> in a WebRTC compliant browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the flag `kms.url` to the JVM executing the demo. As we'll be using maven, you should execute the following command

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a plate in a WebRTC stream on optical character recognition (OCR).

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

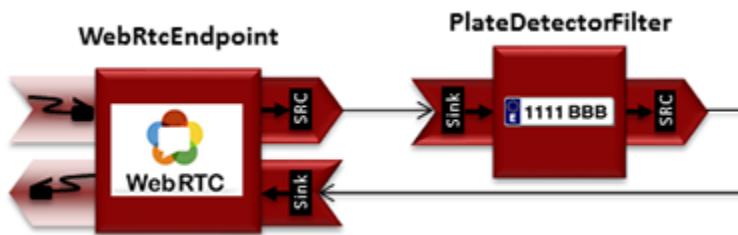


Fig. 16.27: WebRTC with plateDetector filter Media Pipeline

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **PlateDetector** instead of **FaceOverlay** filter. A screenshot of the running example is shown in the following picture:

The following snippet shows how the media pipeline is implemented in the Java server-side code of the demo. An important issue in this code is that a listener is added to the `PlateDetectorFilter` object (`addPlateDetectedListener`). This way, each time a plate is detected in the stream, a message is sent to the client side. As shown in the screenshot below, this event is printed in the console of the GUI.

```
private void start(final WebSocketSession session, JSONObject jsonMessage) {
    try {
        // Media Logic (Media Pipeline and Elements)
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline)
            .build();
```

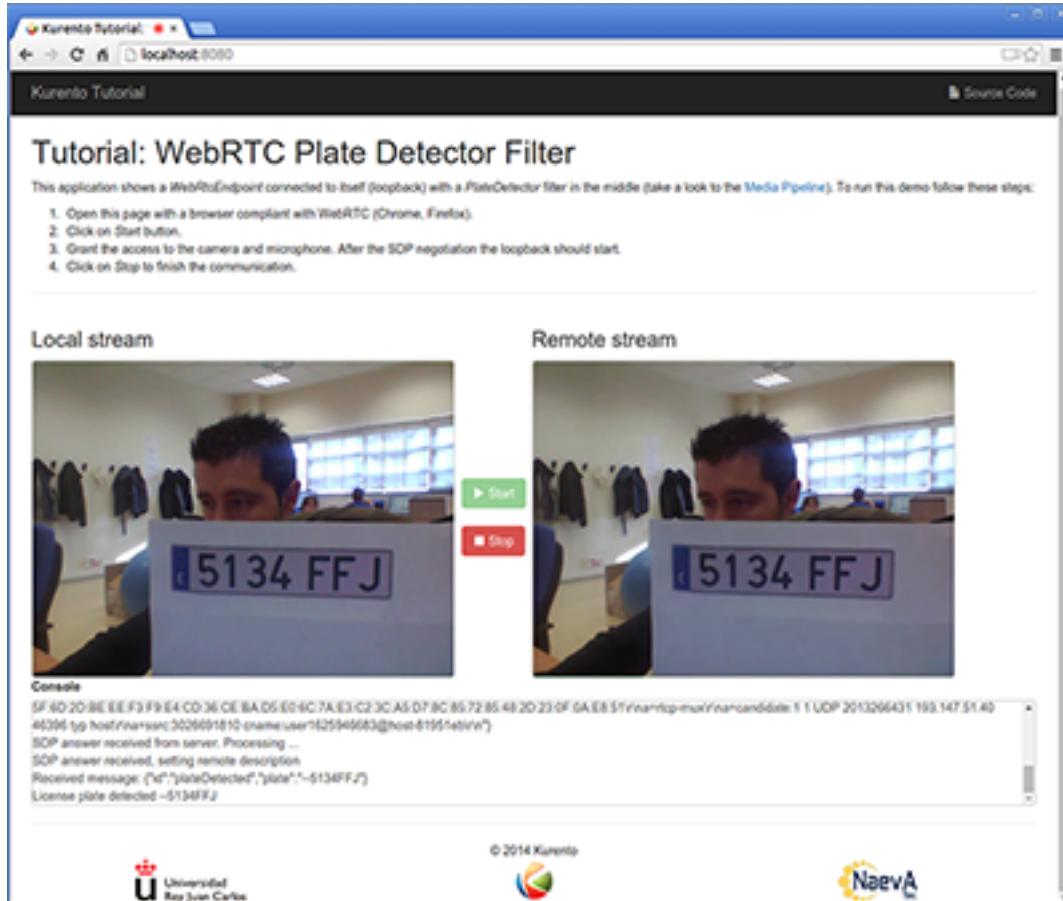


Fig. 16.28: *Plate detector demo in action*

```

user.setWebRtcEndpoint(webRtcEndpoint);
users.put(session.getId(), user);

webRtcEndpoint
    .addIceCandidateFoundListener(new EventListener<IceCandidateFoundEvent>()
→{

    @Override
    public void onEvent(IceCandidateFoundEvent event) {
        JsonObject response = new JsonObject();
        response.addProperty("id", "iceCandidate");
        response.add("candidate", JsonUtils
            .toJsonObject(event.getCandidate()));
        try {
            synchronized (session) {
                session.sendMessage(new TextMessage(
                    response.toString()));
            }
        } catch (IOException e) {
            log.debug(e.getMessage());
        }
    }
});

PlateDetectorFilter plateDetectorFilter = new PlateDetectorFilter.Builder(
    pipeline).build();

webRtcEndpoint.connect(plateDetectorFilter);
plateDetectorFilter.connect(webRtcEndpoint);

plateDetectorFilter
    .addPlateDetectedListener(new EventListener<PlateDetectedEvent>() {
        @Override
        public void onEvent(PlateDetectedEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "plateDetected");
            response.addProperty("plate", event.getPlate());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });
}

// SDP negotiation (offer and answer)
String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);

synchronized (session) {
    session.sendMessage(new TextMessage(response.toString()));
}

```

```

        webRtcEndpoint.gatherCandidates();
    } catch (Throwable t) {
        sendError(session, t.getMessage());
    }
}

```

Dependencies

This Java Spring application is implemented using [Maven](#). The relevant part of the `pom.xml` is where Kurento dependencies are declared. As the following snippet shows, we need two dependencies: the Kurento Client Java dependency (`kurento-client`) and the JavaScript Kurento utility library (`kurento-utils`) for the client-side. Other client libraries are managed with `webjars`:

```

<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-utils-js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>webjars-locator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>bootstrap</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>demo-console</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>adapter.js</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>jquery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars.bower</groupId>
        <artifactId>ekko-lightbox</artifactId>
    </dependency>
</dependencies>

```

Note: We are in active development. You can find the latest version of Kurento Java Client at [Maven Central](#).

Kurento Java Client has a minimum requirement of **Java 7**. Hence, you need to include the following properties in your pom:

```
<maven.compiler.target>1.7</maven.compiler.target>
<maven.compiler.source>1.7</maven.compiler.source>
```

16.4.2 JavaScript Module - Plate Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-platedetector` should be also installed:

```
sudo apt-get install kms-platedetector
```

Warning: Plate detector module is a prototype and its results is not always accurate. Consider this if you are planning to use this module in a production environment.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Due to [Same-origin policy](#), this demo has to be served by an HTTP server. A very simple way of doing this is by means of an HTTP Node.js server which can be installed using [npm](#):

```
sudo npm install http-server -g
```

You also need the source code of this demo. You can clone it from GitHub. Then start the HTTP server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-platedetector
git checkout 6.7.1
bower install
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. Kurento Media Server must use WebSockets over SSL/TLS (WSS), so make sure you check [this](#) too. It is possible to locate the KMS in other machine simple adding the parameter `ws_uri` to the URL:

```
https://localhost:8443/index.html?ws_uri=wss://kms_host:kms_port/kurento
```

Notice that the Kurento Media Server must be connected using a **Secure WebSocket** (i.e., the KMS URI starts with `wss://`). For this reason, the support for secure WebSocket must be enabled in the Kurento Media Server you are using to run this tutorial. For further information about securing applications, please visit the following [page](#).

Understanding this example

This application uses computer vision and augmented reality techniques to detect a plate in a WebRTC stream on optical character recognition (OCR).

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

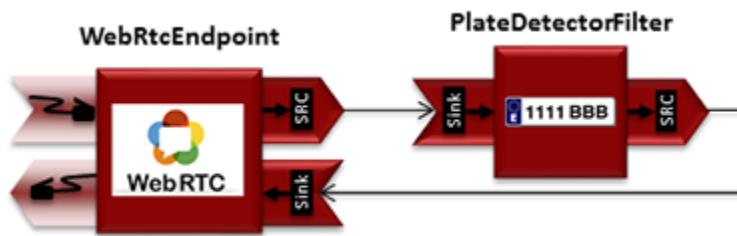


Fig. 16.29: WebRTC with plateDetector filter Media Pipeline

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **PlateDetector** instead of **FaceOverlay** filter. An screenshot of the running example is shown in the following picture:

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

The following snippet shows how the media pipeline is implemented in the Java server-side code of the demo. An important issue in this code is that a listener is added to the `PlateDetectorFilter` object (`addPlateDetectedListener`). This way, each time a plate is detected in the stream, a message is sent to the client side. As shown in the screenshot below, this event is printed in the console of the GUI.

```

...
kurentoClient.register('kurento-module-platedetector')
...

kurentoClient(args.ws_uri, function(error, client) {
  if (error) return onError(error);

  client.create('MediaPipeline', function(error, _pipeline) {
    if (error) return onError(error);

    pipeline = _pipeline;

    console.log("Got MediaPipeline");
  });
}
  
```

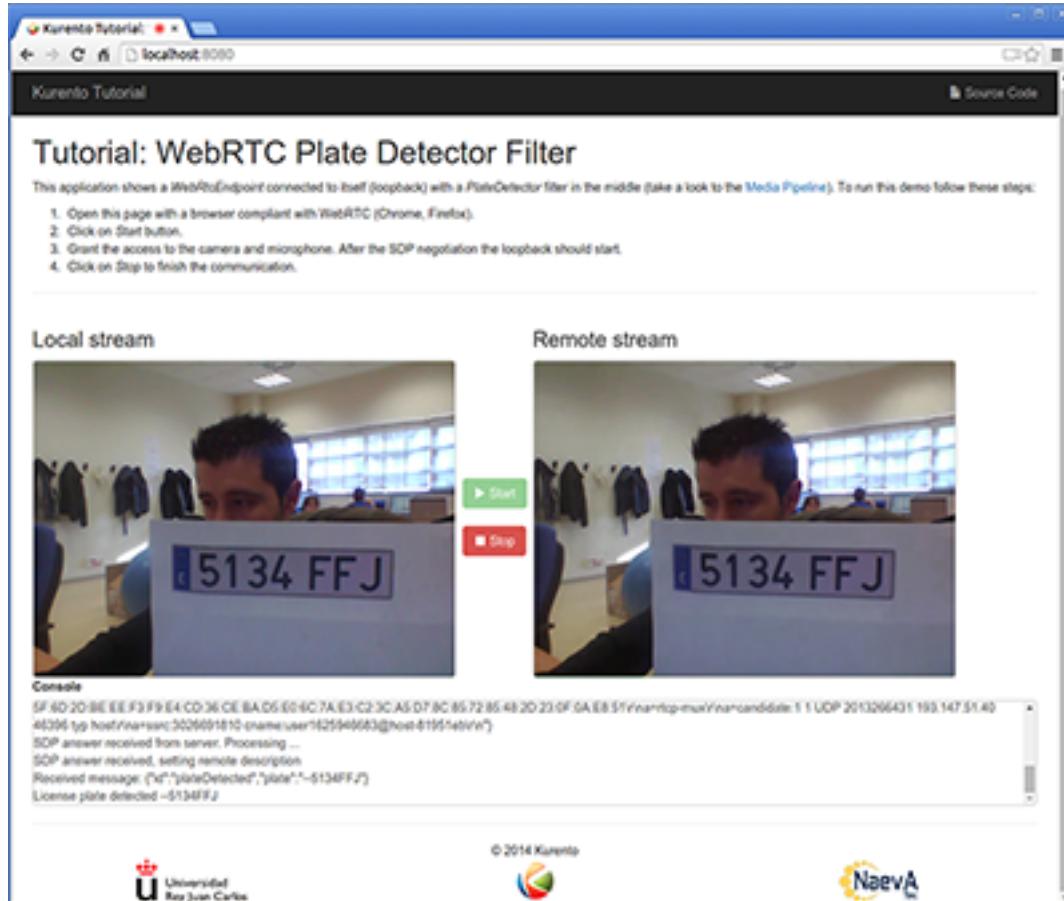


Fig. 16.30: *Plate detector demo in action*

```

pipeline.create('WebRtcEndpoint', function(error, webRtc) {
    if (error) return onError(error);

    console.log("Got WebRtcEndpoint");

    setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

    webRtc.processOffer(sdpOffer, function(error, sdpAnswer) {
        if (error) return onError(error);

        console.log("SDP answer obtained. Processing...");

        webRtc.gatherCandidates(onError);
        webRtcPeer.processAnswer(sdpAnswer);
    });

    pipeline.create('platedetector.PlateDetectorFilter', function(error, filter) {
        if (error) return onError(error);

        console.log("Got Filter");

        filter.on('PlateDetected', function (data){
            console.log("License plate detected " + data.plate);
        });

        client.connect(webRtc, filter, webRtc, function(error) {
            if (error) return onError(error);

            console.log("WebRtcEndpoint --> filter --> WebRtcEndpoint");
        });
    });
});
});

```

Note: The *TURN* and *STUN* servers to be used can be configured simple adding the parameter `ice_servers` to the application URL, as follows:

```

https://localhost:8443/index.html?ice_servers=[{"urls":"stun:stun1.example.net"}, {
    ↵"urls":"stun:stun2.example.net"}]
https://localhost:8443/index.html?ice_servers=[{"urls":"turn:turn.example.org",
    ↵"username":"user","credential":"myPassword"}]

```

Dependencies

The dependencies of this demo has to be obtained using *Bower*. The definition of these dependencies are defined in the `bower.json` file, as follows:

```

"dependencies": {
    "kurento-client": "6.7.1",
    "kurento-utils": "6.7.1"
    "kurento-module-pointerdetector": "6.7.1"
}

```

To get these dependencies, just run the following shell command:

```
bower install
```

Note: We are in active development. You can find the latest versions at [Bower](#).

16.4.3 Node.js Module - Plate Detector Filter

This web application consists on a [WebRTC](#) video communication in mirror (*loopback*) with a plate detector filter element.

Note: This tutorial has been configurated for using https. Follow these [instructions](#) for securing your application.

For the impatient: running this example

First of all, you should install Kurento Media Server to run this demo. Please visit the [installation guide](#) for further information. In addition, the built-in module `kms-platedetector` should be also installed:

```
sudo apt-get install kms-platedetector
```

Warning: Plate detector module is a prototype and its results is not always accurate. Consider this if you are planning to use this module in a production environment.

Be sure to have installed [Node.js](#) and [Bower](#) in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

To launch the application, you need to clone the GitHub project where this demo is hosted, install it and run it:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-platedetector
git checkout 6.7.1
npm install
```

If you have problems installing any of the dependencies, please remove them and clean the npm cache, and try to install them again:

```
rm -r node_modules
npm cache clean
```

Finally, access the application connecting to the URL <https://localhost:8443/> through a WebRTC capable browser (Chrome, Firefox).

Note: These instructions work only if Kurento Media Server is up and running in the same machine as the tutorial. However, it is possible to connect to a remote KMS in other machine, simply adding the argument `ws_uri` to the npm execution command, as follows:

```
npm start -- --ws_uri=ws://kms_host:kms_port/kurento
```

In this case you need to use npm version 2. To update it you can use this command:

```
sudo npm install npm -g
```

Understanding this example

This application uses computer vision and augmented reality techniques to detect a plate in a WebRTC stream on optical character recognition (OCR).

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the video camera stream (the local client-side stream) and other for the mirror (the remote stream). The video camera stream is sent to Kurento Media Server, which processes and sends it back to the client as a remote stream. To implement this, we need to create a *Media Pipeline* composed by the following *Media Element*s:

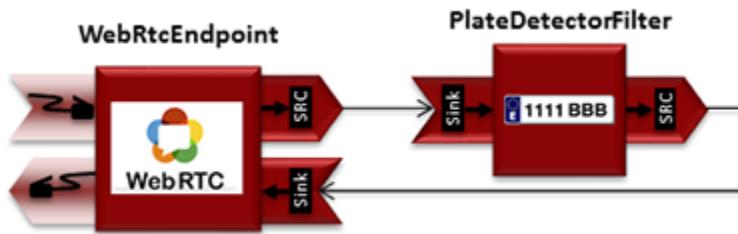


Fig. 16.31: *WebRTC with plateDetector filter Media Pipeline*

The complete source code of this demo can be found in [GitHub](#).

This example is a modified version of the [Magic Mirror](#) tutorial. In this case, this demo uses a **PlateDetector** instead of **FaceOverlay** filter. An screenshot of the running example is shown in the following picture:

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In Javascript and Node, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

The following snippet shows how the media pipeline is implemented in the Java server-side code of the demo. An important issue in this code is that a listener is added to the `PlateDetectorFilter` object (`addPlateDetectedListener`). This way, each time a plate is detected in the stream, a message is sent to the client side. As shown in the screenshot below, this event is printed in the console of the GUI.

```
...
kurento.register('kurento-module-platedetector');
...

function start(sessionId, ws, sdpOffer, callback) {
  if (!sessionId) {
    return callback('Cannot use undefined sessionId');
  }

  getKurentoClient(function(error, kurentoClient) {
    if (error) {
      ...
    }
  });
}
```

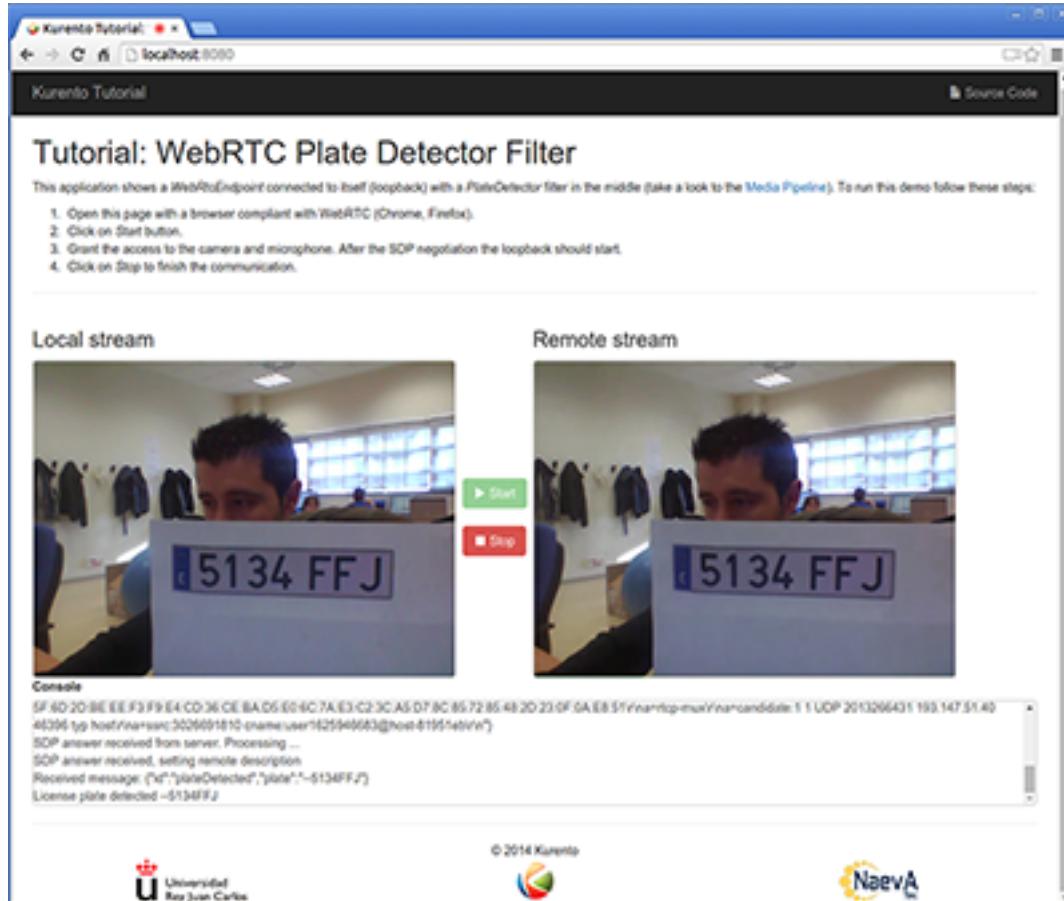


Fig. 16.32: *Plate detector demo in action*

```

        return callback(error);
    }

    kurentoClient.create('MediaPipeline', function(error, pipeline) {
        if (error) {
            return callback(error);
        }

        createMediaElements(pipeline, ws, function(error, webRtcEndpoint, filter)
←{
            if (error) {
                pipeline.release();
                return callback(error);
            }

            if (candidatesQueue[sessionId]) {
                while(candidatesQueue[sessionId].length) {
                    var candidate = candidatesQueue[sessionId].shift();
                    webRtcEndpoint.addIceCandidate(candidate);
                }
            }
        }

        connectMediaElements(webRtcEndpoint, filter, function(error) {
            if (error) {
                pipeline.release();
                return callback(error);
            }

            webRtcEndpoint.on('OnIceCandidate', function(event) {
                var candidate = kurento.getComplexType('IceCandidate')(event.
←candidate);
                ws.send(JSON.stringify({
                    id : 'iceCandidate',
                    candidate : candidate
                }));
            });

            filter.on('PlateDetected', function (data){
                return callback(null, 'plateDetected', data);
            });

            webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer) {
                if (error) {
                    pipeline.release();
                    return callback(error);
                }

                sessions[sessionId] = {
                    'pipeline' : pipeline,
                    'webRtcEndpoint' : webRtcEndpoint
                }
                return callback(null, 'sdpAnswer', sdpAnswer);
            });

            webRtcEndpoint.gatherCandidates(function(error) {
                if (error) {
                    return callback(error);
                }
            })
        });
    });
}

```

```
        });
    });
});
});
});

function createMediaElements(pipeline, ws, callback) {
    pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
        if (error) {
            return callback(error);
        }

        pipeline.create('platedetector.PlateDetectorFilter', function(error, filter) {
            if (error) {
                return callback(error);
            }

            return callback(null, webRtcEndpoint, filter);
        });
    });
}
```

Dependencies

Dependencies of this demo are managed using NPM. Our main dependency is the Kurento Client JavaScript (*kurento-client*). The relevant part of the `package.json` file for managing this dependency is:

```
"dependencies": {
    "kurento-client" : "6.7.1"
}
```

At the client side, dependencies are managed using Bower. Take a look to the `bower.json` file and pay attention to the following section:

```
"dependencies": {
    "kurento-utils" : "6.7.1",
    "kurento-module-pointerdetector": "6.7.1"
}
```

Note: We are in active development. You can find the latest versions at [npm](#) and [Bower](#).

Kurento Utils JS

[TODO full review]

17.1 Overview

Kurento Utils is a wrapper object of an [RTCPeerConnection](#). This object is aimed to simplify the development of WebRTC-based applications.

The source code of this project can be cloned from the [GitHub repository](#).

17.2 How to use it

- **Minified file** - Download the file from [here](#).
- **NPM** - Install and use library in your NodeJS files.

```
npm install kurento-utils
```

```
var utils = require('kurento-utils');
```

- **Bower** - Generate the bundled script file

```
bower install kurento-utils
```

Import the library in your *html* page

```
<script  
src="bower_components/kurento-utils/js/kurento-utils.js"></script>
```

17.3 Examples

There are several tutorials that show kurento-utils used in complete WebRTC applications developed on Java, Node and JavaScript. These tutorials are in GitHub, and you can download and run them at any time.

- **Java** - <https://github.com/Kurento/kurento-tutorial-java>
- **Node** - <https://github.com/Kurento/kurento-tutorial-node>
- **JavaScript** - <https://github.com/Kurento/kurento-tutorial-js>

In the following lines we will show how to use the library to create an `RTCPeerConnection`, and how to negotiate the connection with another peer. The library offers a `WebRtcPeer` object, which is a wrapper of the browser's `RTCPeerConnection` API. Peer connections can be of different types: unidirectional (send or receive only) or bidirectional (send and receive). The following code shows how to create the latter, in order to be able to send and receive media (audio and video). The code assumes that there are two video tags in the page that loads the script. These tags will be used to show the video as captured by your own client browser, and the media received from the other peer. The constructor receives a property that holds all the information needed for the configuration.

```
var videoInput = document.getElementById('videoInput');
var videoOutput = document.getElementById('videoOutput');

var constraints = {
  audio: true,
  video: {
    width: 640,
    framerate: 15
  }
};

var options = {
  localVideo: videoInput,
  remoteVideo: videoOutput,
  onicecandidate : onIceCandidate,
  mediaConstraints: constraints
};

var webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
  if(error) return onError(error)

  this.generateOffer(onOffer)
});
```

With this little code, the library takes care of creating the `RTCPeerConnection`, and invoking `getUserMedia` in the browser if needed. The constraints in the property are used in the invocation, and in this case both microphone and webcam will be used. However, this does not create the connection. This is only achieved after completing the SDP negotiation between peers. This process implies exchanging SDPs offer and answer and, since *Trickle ICE* is used, a number of candidates describing the capabilities of each peer. How the negotiation works is out of the scope of this document. More info can be found in [this link](#).

In the previous piece of code, when the `webRtcPeer` object gets created, the SDP offer is generated with `this.generateOffer(onOffer)`. The only argument passed is a function, that will be invoked once the browser's peer connection has generated that offer. The `onOffer` callback method is responsible for sending this offer to the other peer, by any means devised in your application. Since that is part of the signaling plane and business logic of each particular application, it won't be covered in this document.

Assuming that the SDP offer has been received by the remote peer, it must have generated an SDP answer, that should

be received in return. This answer must be processed by the `webRtcEndpoint`, in order to fulfill the negotiation. This could be the implementation of the `onOffer` callback function. We've assumed that there's a function somewhere in the scope, that allows sending the SDP to the remote peer.

```
function onOffer(error, sdpOffer) {
    if (error) return onError(error);

    // We've made this function up sendOfferToRemotePeer(sdpOffer,
    function(sdpAnswer) {
        webRtcPeer.processAnswer(sdpAnswer);
    });
}
```

As we've commented before, the library assumes the use of *Trickle ICE* to complete the connection between both peers. In the configuration of the `webRtcPeer`, there is a reference to a `onIceCandidate` callback function. The library will use this function to send ICE candidates to the remote peer. Since this is particular to each application, we will just show the signature

```
function onIceCandidate(candidate) {
    // Send the candidate to the remote peer
}
```

In turn, our client application must be able to receive ICE candidates from the remote peer. Assuming the signaling takes care of receiving those candidates, it is enough to invoke the following method in the `webRtcPeer` to consider the ICE candidate

```
webRtcPeer.addIceCandidate(candidate);
```

Following the previous steps, we have:

- Sent and SDP offer to a remote peer
- Received an SDP answer from the remote peer, and have the `webRtcPeer` process that answer.
- Exchanged ICE candidates between both peer, by sending the ones generated in the browser, and processing the candidates received by the remote peer.

This should complete the negotiation process, and should leave us with a working bidirectional WebRTC media exchange between both peers.

17.4 Using data channels

WebRTC data channels lets you send text or binary data over an active WebRTC connection. The `WebRtcPeer` object can provide access to this functionality by using the `RTCDATAChannel` form the wrapped `RTCPeerConnection` object. This allows you to inject into and consume data from the pipeline. This data can be treated by each endpoint differently. For instance, a `WebRtcPeer` object in the browser, will have the same behavior as the `RTCDATAChannel` (you can see a description [here](#)). Other endpoints could make use of this channel to send information: a filter that detects QR codes in a video stream, could send the detected code to the clients through a data channel. This special behavior should be specified in the filter.

The use of data channels in the `WebRtcPeer` object is indicated by passing the `dataChannels` flag in the options bag, along with the desired options.

```
var options = {
    localVideo : videoInput,
    remoteVideo : videoOutput,
    dataChannels : true,
```

```
dataChannelConfig: {
  id : getChannelName(),
  onmessage : onMessage,
  onopen : onOpen,
  onclose : onClosed,
  onbufferedamountlow : onbufferedamountlow,
  onerror : onerror
},
onicecandidate : onIceCandidate
}

webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,  
→onWebRtcPeerCreated);
```

The values in `dataChannelConfig` are all optional. Once the `webRtcPeer` object is created, and after the connection has been successfully negotiated, users can send data through the data channel

```
webRtcPeer.send('your data stream here');
```

The format of the data you are sending, is determined by your application, and the definition of the endpoints that you are using.

The lifecycle of the underlying `RTCDataChannel`, is tied to that of the `webRtcPeer`: when the `webRtcPeer.dispose()` method is invoked, the data channel will be closed and released too.

17.5 Reference documentation

17.5.1 WebRtcPeer

The constructor for `WebRtcPeer` is `WebRtcPeer(mode, options, callback)` where:

- **mode**: Mode in which the PeerConnection will be configured. Valid values are
 - `recv`: receive only media.
 - `send`: send only media.
 - `sendRecv`: send and receive media.
- **options** : It is a group of parameters and they are optional. It is a json object.
 - `localVideo`: Video tag in the application for the local stream.
 - `remoteVideo`: Video tag in the application for the remote stream.
 - `videoStream`: Provides an already available video stream that will be used instead of using the media stream from the local webcam.
 - `audioStreams`: Provides an already available audio stream that will be used instead of using the media stream from the local microphone.
 - `mediaConstraints`: Defined the quality for the video and audio
 - `connectionConstraints`: Defined the connection constraint according with browser like googIPv6, DtlsSrtpKeyAgreement...
 - `peerConnection`: Use a peerConnection which was created before
 - `sendSource`: Which source will be used

- * *webcam*
- * *screen*
- * *window*
- *onstreamended*: Method that will be invoked when stream ended event happens
- *onicecandidate*: Method that will be invoked when ice candidate event happens
- *oncandidategatheringdone*: Method that will be invoked when all candidates have been harvested
- *dataChannels*: Flag for enabling the use of data channels. If *true*, then a data channel will be created in the *RTCPeerConnection* object.
- *dataChannelConfig*: It is a JSON object with the configuration passed to the DataChannel when created. It supports the following keys:
 - * *id*: Specifies the *id* of the data channel. If none specified, the same *id* of the *WebRtcPeer* object will be used.
 - * *options*: Options object passed to the data channel constructor.
 - * *onopen*: Function invoked in the *onopen* event of the data channel, fired when the channel is open.
 - * *onclose*: Function invoked in the *onclose* event of the data channel, fired when the data channel is closed.
 - * *onmessage*: Function invoked in the *onmessage* event of the data channel. This event is fired every time a message is received.
 - * *onbufferedamountlow*: Is the event handler called when the *bufferedamountlow* event is received. Such an event is sent when *RTCDataChannel.bufferedAmount* drops to less than or equal to the amount specified by the *RTCDataChannel.bufferedAmountLowThreshold* property.
 - * *onerror*: Callback function invoked when an error in the data channel is produced. If none is provided, an error trace message will be logged in the browser console.
- *simulcast*: Indicates whether simulcast is going to be used. Value is *true|false*
- *configuration*: It is a JSON object where ICE Servers are defined using
 - * *iceServers*: The format for this variable is like:

```
[{"urls": "turn:turn.example.org", "username": "user", "credential": "myPassword"}]
[{"urls": "stun:stun1.example.net"}, {"urls": "stun:stun2.example.net"}]
```

- **callback**: It is a callback function which indicate, if all worked right or not

Also there are 3 specific methods for creating *WebRtcPeer* objects without using *mode* parameter:

- **WebRtcPeerRecvonly(options, callback)**: Create a *WebRtcPeer* as receive only.
- **WebRtcPeerSendonly(options, callback)**: Create a *WebRtcPeer* as send only.
- **WebRtcPeerSendrecv(options, callback)**: Create a *WebRtcPeer* as send and receive.

MediaConstraints

Constraints provide a general control surface that allows applications to both select an appropriate source for a track and, once selected, to influence how a source operates. *getUserMedia()* uses constraints to help select an appro-

priate source for a track and configure it. For more information about media constraints and its values, you can check [here](#).

By default, if the mediaConstraints is undefined, this constraints are used when `getUserMedia` is called:

```
{  
  audio: true,  
  video: {  
    width: 640,  
    framerate: 15  
  }  
}
```

If `mediaConstraints` has any value, the library uses this value for the invocation of `getUserMedia`. It is up to the browser whether those constraints are accepted or not.

In the examples section, there is one example about the use of media constraints.

Methods

getPeerConnection

Using this method the user can get the peerConnection and use it directly.

showLocalVideo

Use this method for showing the local video.

getLocalStream

Using this method the user can get the local stream. You can use **muted** property to silence the audio, if this property is *true*.

getRemoteStream

Using this method the user can get the remote stream.

getCurrentFrame

Using this method the user can get the current frame and get a canvas with an image of the current frame.

processAnswer

Callback function invoked when a SDP answer is received. Developers are expected to invoke this function in order to complete the SDP negotiation. This method has two parameters:

- **sdpAnswer**: Description of sdpAnswer
- **callback**: It is a function with *error* like parameter. It is called when the remote description has been set successfully.

processOffer

Callback function invoked when a SDP offer is received. Developers are expected to invoke this function in order to complete the SDP negotiation. This method has two parameters:

- **sdpOffer**: Description of sdpOffer
- **callback**: It is a function with *error* and *sdpAnswer* like parameters. It is called when the remote description has been set successfully.

dispose

This method frees the resources used by WebRtcPeer.

addIceCandidate

Callback function invoked when an ICE candidate is received. Developers are expected to invoke this function in order to complete the SDP negotiation. This method has two parameters:

- **iceCandidate**: Literal object with the ICE candidate description
- **callback**: It is a function with *error* like parameter. It is called when the ICE candidate has been added.

getLocalSessionDescriptor

Using this method the user can get peerconnection's local session descriptor.

getRemoteSessionDescriptor

Using this method the user can get peerconnection's remote session descriptor.

generateOffer

Creates an offer that is a request to find a remote peer with a specific configuration.

17.5.2 How to do screen share

Screen and window sharing depends on the privative module *kurento-browser-extensions*. To enable its support, you'll need to install the package dependency manually or provide a *getScreenConstraints* function yourself on runtime. The option **sendSource** could be *window* or *screen* before create a WebRtcEndpoint. If it's not available, when trying to share the screen or a window content it will throw an exception.

17.6 Souce code

The code is at [github](#).

Be sure to have *Node.js* and *Bower* installed in your system:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

To install the library, it is recommended to do that from the [NPM repository](#):

```
npm install kurento-utils
```

Alternatively, you can download the code using Git and install manually its dependencies:

```
git clone https://github.com/Kurento/kurento-utils
cd kurento-utils
npm install
```

17.7 Build for browser

After you download the project, to build the browser version of the library you'll only need to execute the [grunt](#) task runner. The file needed will be generated on the *dist* folder. Alternatively, if you don't have it globally installed, you can run a local copy by executing:

```
cd kurento-utils
node_modules/.bin/grunt
```

CHAPTER 18

Endpoint Events

This is a list of all events that can be emitted by an instance of `WebRtcEndpoint`. This class belongs to a chain of inherited classes, so this list includes events from all of them, starting from the topmost class in the inheritance tree:

Table of Contents

- *Endpoint Events*
 - *MediaObject events*
 - * *Error*
 - *MediaElement events*
 - * *ElementConnected*
 - * *ElementDisconnected*
 - * *MediaFlowInStateChange*
 - * *MediaFlowOutStateChange*
 - * *MediaTranscodingStateChange*
 - *BaseRtpEndpoint events*
 - * *ConnectionStateChanged*
 - * *MediaStateChanged*
 - *WebRtcEndpoint events*
 - * *DataChannelClose*
 - * *DataChannelOpen*
 - * *IceCandidateFound*
 - * *IceComponentStateChange*

- * *IceGatheringDone*
 - * *NewCandidatePairSelected*
 - *Sample sequence of events: WebRtcEndpoint*

18.1 MediaObject events

This is the base interface used to manage capabilities common to all Kurento elements, including both *MediaElement* and *MediaPipeline*.

18.1.1 Error

Some error has occurred. Check the event parameters (such as *description*, *errorCode*, and *type*) to get information about what happened.

18.2 MediaElement events

These events indicate some low level information about the state of GStreamer, the underlying multimedia framework.

Note that the *MediaFlowInStateChange* and *MediaFlowOutStateChange* events are not 100% reliable to check if a RTP connection is flowing: RTCP packets do not usually flow at a constant rate. For example, minimizing a browser window with an *RTCPeerConnection* might affect this interval.

18.2.1 ElementConnected

[TODO - add contents]

18.2.2 ElementDisconnected

[TODO - add contents]

18.2.3 MediaFlowInStateChange

- State = *Flowing*: Data is arriving from the KMS Pipeline, and flowing **into** the Endpoint. Technically, this means that there are GStreamer Buffers flowing from the Pipeline to the Endpoint's *sink* pad. For example, with a Recorder element this event would fire when media arrives from the Pipeline to be written to disk.
- State = *NotFlowing*: The element is not receiving any input data from the Pipeline.

18.2.4 MediaFlowOutStateChange

- State = *Flowing*: There is data flowing **from** the Endpoint towards the KMS Pipeline. Technically, this means that there are GStreamer Buffers flowing from the Endpoint's *src* pad to the Pipeline. For example, with a Player element this event would fire when media is read from disk and is pushed to the Pipeline.
- State = *NotFlowing*: The element is not sending any output data to the Pipeline.

18.2.5 MediaTranscodingStateChange

All Endpoint objects in Kurento Media Server embed a custom-made GStreamer element called *agnosticbin*. This element is used to provide seamless interconnection of components in the *MediaPipeline*, regardless of the format and codec configuration of the input and output media streams.

When media starts flowing through any *MediaElement*-derived object, an internal dynamic configuration is done in order to match the incoming media format with the requested output media format. If both input and output formats are compatible (at the codec level), then the media can be transferred directly without any extra processing. However, if the input and output media formats are not compatible, the internal transcoding module will get enabled to convert the input media format to be compatible with the required output.

For example, if a WebRtcEndpoint receives a *VP8* video stream from a Chrome browser, and then has to send the stream to a Safari browser which only accepts *H.264*, then the media will need to be transcoded.

- State = *Transcoding*: The *MediaElement* will transcode the incoming media, because its format is not compatible with the requested output format.
- State = *NotTranscoding*: The *MediaElement* will *not* transcode the incoming media, because its format is compatible with the requested output format.

18.3 BaseRtpEndpoint events

These events provide information about the state of the RTP connection for each stream in the WebRTC call.

18.3.1 ConnectionStateChanged

- State = *Connected*: All of the `KmsIRtpConnection` objects have been created [TODO: explain what this means].
- State = *Disconnected*: At least one of the `KmsIRtpConnection` objects is not created yet.

Call sequence:

```
signal KmsIRtpConnection::"connected"
-> signal KmsSdpSession::"connection-state-changed"
-> signal KmsBaseRtpEndpoint::"connection-state-changed"
-> BaseRtpEndpointImpl::updateConnectionState
```

18.3.2 MediaStateChanged

- State = *Connected*: At least *one* of the audio or video RTP streams in the session is still alive (receiving RTCP packets).
- State = *Disconnected*: None of the RTP streams belonging to the session is alive (ie. no RTCP packets are received for any of them).

These signals from `GstRtpBin` will trigger the `MediaStateChanged` event:

- `GstRtpBin::"on-bye-ssrc"`: State = *Disconnected*.
- `GstRtpBin::"on-bye-timeout"`: State = *Disconnected*.
- `GstRtpBin::"on-timeout"`: State = *Disconnected*.
- `GstRtpBin::"on-ssrc-active"`: State = *Connected*.

Call sequence:

```
signal GstRtpBin::"on-bye-ssrc"
|| signal GstRtpBin::"on-bye-timeout"
|| signal GstRtpBin::"on-timeout"
|| signal GstRtpBin::"on-ssrc-active"
-> signal KmsBaseRtpEndpoint::"media-state-changed"
-> BaseRtpEndpointImpl::updateMediaState
```

Note: MediaStateChanged (State = *Connected*) will happen after these other events have been emitted:

1. NewCandidatePairSelected.
 2. IceComponentStateChanged (State: *Connected*).
 3. MediaFlowOutStateChange (State: *Flowing*).
-

18.4 WebRtcEndpoint events

These events provide information about the state of libnice, the underlying library in charge of the ICE Gathering process. The ICE Gathering is typically done before attempting any WebRTC call.

For further reference, see the libnice's [Agent documentation](#) and [source code](#).

18.4.1 DataChannelClose

[TODO - add contents]

18.4.2 DataChannelOpen

[TODO - add contents]

18.4.3 IceCandidateFound

A new local candidate has been found, after the ICE Gathering process was started. Equivalent to the signal `NiceAgent::"new-candidate-full"`.

18.4.4 IceComponentStateChange

This event carries the state values from the signal `NiceAgent::"component-state-changed"`.

- State = *Disconnected*: There is no active connection, and the ICE process is stopped.
NiceAgent state: `NICE_COMPONENT_STATE_DISCONNECTED`, “*No activity scheduled*”.
- State = *Gathering*: The Endpoint has started finding all possible local candidates, which will be notified through the event `IceCandidateFound`.
NiceAgent state: `NICE_COMPONENT_STATE_GATHERING`, “*Gathering local candidates*”.

- State = *Connecting*: The Endpoint has started the connectivity checks between at least one pair of local and remote candidates.

NiceAgent state: NICE_COMPONENT_STATE_CONNECTING, “*Establishing connectivity*”.

- State = *Connected*: At least one candidate pair resulted in a successful connection. This happens right after the event NewCandidatePairSelected.

NiceAgent state: NICE_COMPONENT_STATE_CONNECTED, “*At least one working candidate pair*”.

- State = *Ready*: All local candidates have been gathered, all pairs of local and remote candidates have been tested for connectivity, and a successful connection was established.

NiceAgent state: NICE_COMPONENT_STATE_READY, “*ICE concluded, candidate pair selection is now final*”.

- State = *Failed*: All local candidates have been gathered, all pairs of local and remote candidates have been tested for connectivity, but still none of the connection checks was successful, so no connectivity was reached to the remote peer.

NiceAgent state: NICE_COMPONENT_STATE_FAILED, “*Connectivity checks have been completed, but connectivity was not established*”.

This graph shows the possible state changes:

Note: The states *Ready* and *Failed* indicate that the ICE transport has completed gathering and is currently idle. However, since events such as adding a new interface or a new TURN server will cause the state to go back, *Ready* and *Failed* are **not** terminal states.

18.4.5 IceGatheringDone

All local candidates have been found, all remote candidates have been received from the remote peer, and all pairs of local-remote candidates have been tested for connectivity. When this happens, all activity of the ICE agent stops. Equivalent to the signal NiceAgent::”candidate-gathering-done”.

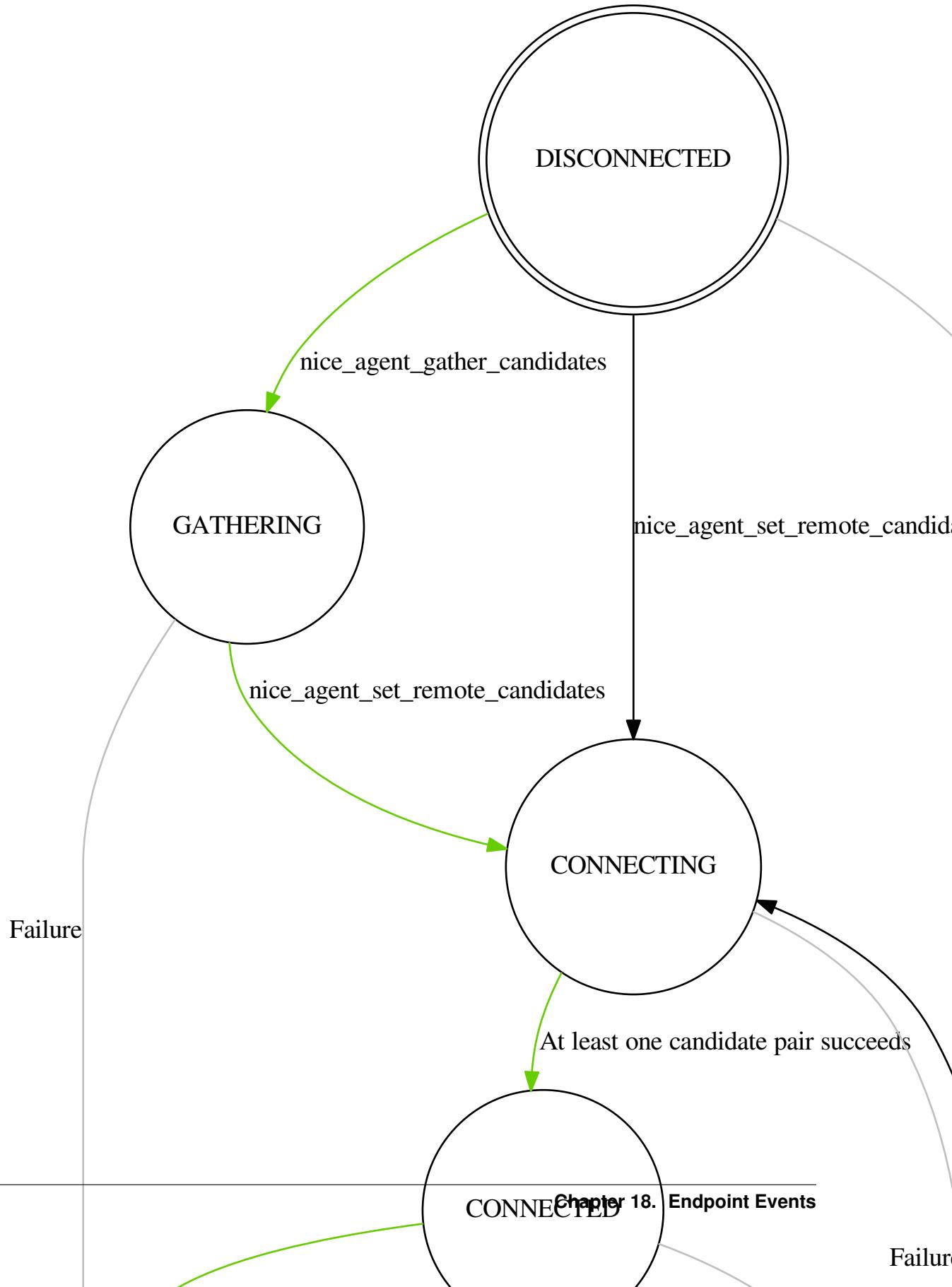
18.4.6 NewCandidatePairSelected

During the connectivity checks one of the pairs happened to provide a successful connection, and the pair had a higher preference than the previously selected one (or there was no previously selected pair yet). Equivalent to the signal NiceAgent::”new-selected-pair”.

18.5 Sample sequence of events: WebRtcEndpoint

Once an instance of *WebRtcEndpoint* is created inside a Media Pipeline, an event handler should be added for each one of the events that can be emitted by the endpoint. Later, the endpoint should be instructed to do one of either:

- Generate an SDP Offer, when KMS is the caller. Later, the remote peer will generate an SDP Answer as a reply, which must be provided to the endpoint.
- Process an SDP Offer generated by the remote peer, when KMS is the callee. This will in turn generate an SDP Answer, which should be provided to the remote peer.



As a last step, the *WebRtcEndpoint* should be instructed to start the ICE Gathering process.

You can see a working example of this in [Kurento Java Tutorial - Hello World](#). This example code shows the typical usage for the *WebRtcEndpoint*:

```
KurentoClient kurento;
MediaPipeline pipeline = kurento.createMediaPipeline();
WebRtcEndpoint webRtcEp = new WebRtcEndpoint.Builder(pipeline).build();
webRtcEp.addIceCandidateFoundListener(...);
webRtcEp.addIceComponentStateChangedListener(...);
webRtcEp.addIceGatheringDoneListener(...);
webRtcEp.addNewCandidatePairSelectedListener(...);

// Receive an SDP Offer, via the application's custom signaling mechanism
String sdpOffer = recvMessage();

// Process the SDP Offer, generating an SDP Answer
String sdpAnswer = webRtcEp.processOffer(sdpOffer);

// Send the SDP Answer, via the application's custom signaling mechanism
sendMessage(sdpAnswer);

// Start gathering candidates for ICE
webRtcEp.gatherCandidates();
```

The application's custom signaling mechanism could be as simple as some ad-hoc messaging protocol built upon WebSocket endpoints.

When a *WebRtcEndpoint* instance has been created, and all event handlers have been added, starting the ICE process will generate a sequence of events very similar to this one:

1. Event(s): *IceCandidateFound*.

Typically, candidates of type host (corresponding to the LAN, local network) are almost immediately found after starting the ICE gathering, and this event can arrive even before the event *IceComponentStateChanged* is emitted.

2. Event: *IceComponentStateChanged* (State: *Gathering*).

At this point, the local peer is gathering more candidates, and it is also waiting for the candidates gathered by the remote peer, which could start arriving at any time.

3. Function call: *AddIceCandidate*.

The remote peer found some initial candidates, and started sending them. Typically, the first candidate received is of type host, because those are found the fastest.

4. Event: *IceComponentStateChanged* (State: *Connecting*).

After receiving the very first of the remote candidates, the ICE agent starts with the connectivity checks.

5. Function call(s): *AddIceCandidate*.

The remote peer will continue sending its own gathered candidates, of any type: host, srflx (*STUN*), relay (*TURN*).

6. Event: *IceCandidateFound*.

The local peer will also continue finding more of the available local candidates.

7. *NewCandidatePairSelected*.

The ICE agent makes local and remote candidate pairs. If one of those pairs pass the connectivity checks, it is selected for the WebRTC connection.

8. `IceComponentStateChanged` (State: *Connected*).

After selecting a candidate pair, the connection is established. *At this point, the media stream(s) can start flowing.*

9. `NewCandidatePairSelected`.

Typically, better candidate pairs will be found over time. The old pair will be abandoned in favor of the new one.

10. `IceGatheringDone`.

When all candidate pairs have been tested, no more work is left to do for the ICE agent. The gathering process is finished.

11. `IceComponentStateChanged` (State: *Ready*).

As a consequence of finishing the ICE gathering, the component state gets updated.

CHAPTER 19

NAT Traversal

NAT Traversal, also known as *Hole Punching*, is the procedure of opening an inbound port in the NAT tables of the routers which implement this technology (which are the vast majority of home and corporate routers).

There are different types of NAT, depending on how they behave: **Full Cone**, **Address-Restricted Cone**, **Port-Restricted Cone**, and **Symmetric**. For a comprehensive explanation of NAT and the different types that exist, please read our Knowledge Base document: [NAT Types and NAT Traversal](#).

19.1 WebRTC with ICE

ICE is the standard method used by *WebRTC* to solve the issue of NAT Traversal. Kurento supports ICE by means of a 3rd-party library: [libnice, The GLib ICE implementation](#).

Refer to the [logging documentation](#) if you need to enable the debug logging for this library.

19.2 RTP without ICE

KMS is able to automatically infer what is the public IP and port of any remote peer which is communicating with it through an RTP connection. This removes the need to use ICE in some specific situations, where that complicated mechanism is not desired. This new automatic port discovery was inspired by the **Connection-Oriented Media Transport** (COMEDIA) as presented by the early Drafts of what finally would become the RFC 4145.

TCP-Based Media Transport in the Session Description Protocol (SDP) ([IETF RFC 4145](#)) defines an SDP extension which adds TCP connections and procedures, such as how a passive machine would wait for connections from a remote active machine and be able to obtain connection information from the active one, upon reception of an initial connection.

Early Drafts of RFC 4145 (up to [Draft 05](#)) also contemplated the usage of this same concept of “Connection-Oriented Media Transport in SDP” with UDP connections, as a way of aiding NAT traversal. This is what has been used as a basis for the implementation of automatic port discovery in KMS.

It works as follows:

1. The machine behind a NAT router acts as the active peer. It sends an SDP Offer to the other machine, the passive peer.
 - (a) Sending an SDP Offer from behind a NAT means that the IP and port specified in the SDP message are actually just the private IP and port of that machine, instead of the public ones. The passive peer won't be able to use these to communicate back to the active peer. Due to this, the SDP Offer states the port 9 (*Discard port*) instead of whatever port the active machine will be using.
 - (b) The SDP Offer includes the media-level attribute `a=direction:active`, so the passive peer is able to acknowledge that the Connection-Oriented Media Transport is being used for that media, and it writes `a=direction:passive` in its SDP Answer.
2. The passive peer receives the SDP Offer and answers it as usual, indicating the public IP and port where it will be listening for incoming packets. Besides that, it must ignore the IP and port indicated in the received SDP Offer. Instead, it must enter a wait state, until the active peer starts sending some packets.
3. When the active peer sends the first RTP/RTCP packets to the IP and port specified in the SDP Answer, the passive peer will be able to analyze them on reception and extract the public IP and reception port of the active peer.
4. The passive peer is now able to send RTP/RTCP packets to the discovered IP and port values of the active peer.

This mechanism has the following requisites and/or limitations:

- Only the active peer can be behind a NAT router. The passive peer must have a publicly accessible IP and port for RTP.
- The active peer must be able to receive RTP/RTCP packets at the same ports that are used to send RTP/RTCP packets. In other words, the active peer must be compatible with *Symmetric RTP and RTCP* as defined in [IETF RFC 4961](#).
- The active peer must actually do send some RTP/RTCP packets before the passive peer is able to send any data back. In other words, it is not possible to establish a one-way stream where only the passive peer sends data to the active peer.

This is how to enable the Connection-Oriented Media Transport mode:

- The SDP Offer must be sent from the active peer to the passive peer.
- The IP stated in the SDP Offer can be anything (as it will be ignored), so `0.0.0.0` can be used.
- The Port stated in the SDP Offer should be 9 (*Discard port*).
- The active peer must include the media-level attribute `a=direction:active` in the SDP Offer, for each media that requires automatic port discovery.
- The passive peer must acknowledge that it supports the automatic port discovery mode, by including the media-level attribute `a=direction:passive` in its SDP Answer. As per normal rules of the SDP Offer/Answer Model ([IETF RFC 3264](#)), if this attribute is not present in the SDP Answer, then the active peer must assume that the passive peer is not compatible with this functionality and should react to this fact as whatever is deemed appropriate by the application developer.

19.2.1 Example

This is a minimal example of an *SDP Offer/Answer* negotiation that a machine would perform with KMS from behind a NAT router. The highlighted lines are those relevant to NAT Traversal:

Listing 19.1: SDP Offer

```
v=0
o=- 0 0 IN IP4 0.0.0.0
s=Example sender
c=IN IP4 0.0.0.0
t=0 0
m=audio 9 RTP/AVPF 96
a=rtpmap:96 opus/48000/2
a=sendonly
a=direction:active
a=ssrc:111111 cname:active@example.com
m=video 9 RTP/AVPF 103
a=rtpmap:103 H264/90000
a=sendonly
a=direction:active
a=ssrc:222222 cname:active@example.com
```

This is what KMS would answer:

Listing 19.2: SDP Answer

```
v=0
o=- 3696336115 3696336115 IN IP4 80.28.30.32
s=Kurento Media Server
c=IN IP4 80.28.30.32
t=0 0
m=audio 56740 RTP/AVPF 96
a=rtpmap:96 opus/48000/2
a=recvonly
a=direction:passive
a=ssrc:4061617641 cname:user885892801@host-b546a6e8
m=video 37616 RTP/AVPF 103
a=rtpmap:103 H264/90000
a=recvonly
a=direction:passive
a=ssrc:1363449382 cname:user885892801@host-b546a6e8
```

In this particular example, KMS is installed in a server with the public IP 80.28.30.32; also, it won't be sending media to the active peer, only receiving it (as requested by the application with `a=sendonly`, and acknowledged by KMS with `a=recvonly`).

Note that even in this case, KMS still needs to know on what port the sender is listening for RTCP feedback packets, which are a mandatory part of the RTP protocol. So, in this example, KMS will learn the public IP and port of the active machine, and will use those to send the Receiver Report RTCP packets to the sender.

CHAPTER 20

Securing Kurento Applications

[TODO full review]

Starting with Chrome 47, WebRTC is only allowed from SECURE ORIGINS (HTTPS or localhost). Check their [release notes](#) for further information about this issue.

Note: Keep in mind that serving your application through HTTPS, forces you to use WebSockets Secure (WSS) if you are using websockets to control your application server.

20.1 Securing client applications

20.1.1 Configure Java applications to use HTTPS

- The application needs a certificate in order to enable HTTPS:
 - Request a certificate from a local certification authority.
 - Create an self-signed certificate.

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore \
keystore.jks -storepass password -validity 360 -keysize 2048
```

- Use the certificate in your application:
 - Include a valid keystore in the *jar* file:

File *keystore.jks* must be in the project's root path, and a file named *application.properties* must exist in *src/main/resources/*, with the following content:

```
server.port: 8443
server.ssl.key-store: keystore.jks
server.ssl.key-store-password: yourPassword
```

```
server.ssl.keyStoreType: JKS
server.ssl.keyAlias: yourKeyAlias
```

- You can also specify the location of the properties file. When launching your Spring-Boot based app, issue the flag `-Dspring.config.location=<path-to-properties>`.

- Start application

```
mvn compile exec:java -Dkms.url=ws://kms_host:kms_port/kurento
```

Note: If you plan on using a webserver as proxy, like Nginx or Apache, you'll need to `setAllowedOrigins` when registering the handler. Please read the official Spring documentation entry for more info.

20.1.2 Configure Node applications to use HTTPS

- The application requires a valid SSL certificate in order to enable HTTPS:
 - Request a certificate from a local certification authority.
 - Create your own self-signed certificate as explained [here](#). This will show you how to create the required files: `server.crt`, `server.key` and `server.csr`.

Add the following changes to `server.js` in order to enable HTTPS:

```
...
var express = require('express');
var ws = require('ws');
var fs    = require('fs');
var https = require('https');
...

var options =
{
  key:  fs.readFileSync('key/server.key'),
  cert: fs.readFileSync('keys/server.crt')
};

var app = express();

var server = https.createServer(options, app).listen(port, function() {
...
});

var wss = new ws.Server({
  server : server,
  path : '/'
});

wss.on('connection', function(ws) {
...
});
```

- Start application

```
npm start
```

20.1.3 Configure Javascript applications to use HTTPS

- You'll need to provide a valid SSL certificate in order to enable HTTPS:
 - Request a certificate from a local certification authority.
 - Create your own self-signed certificate as explained [here](#). This will show you how to create the required files: *server.crt*, *server.key* and *server.csr*.
- Start the application using the certificates:

```
http-server -p 8443 -S -C keys/server.crt -K keys/server.key
```

20.2 Securing server applications

20.2.1 Configure Kurento Media Server to use Secure WebSocket (WSS)

First, you need to change the configuration file of Kurento Media Server, i.e. `/etc/kurento/kurento.conf.json`, uncommenting the following lines:

```
"secure": {
  "port": 8433,
  "certificate": "defaultCertificate.pem",
  "password": ""
},
```

If this PEM certificate is a signed certificate (by a Certificate Authority such as Verisign), then you are done. If you are going to use a self-signed certificate (suitable for development), then there is still more work to do.

You can generate a self signed certificate by doing this:

```
certtool --generate-privkey --outfile defaultCertificate.pem
echo 'organization = your organization name' > certtool tmpl
certtool --generate-self-signed --load-privkey defaultCertificate.pem \
--template certtool tmpl >> defaultCertificate.pem
sudo chown kurento defaultCertificate.pem
```

Due to the fact that the certificate is self-signed, applications will reject it by default. For this reason, you'll need to force them to accept it.

- Browser applications: You'll need to manually accept the certificate as trusted one before secure WebSocket connections can be established. By default, this can be done by connecting to <https://localhost:8433/kurento> and accepting the certificate in the browser.
- Java applications, follow the instructions of this [link](#) (get `InstallCert.java` from [here](#)). You'll need to instruct the `KurentoClient` needs to be configured to allow the use of certificates. For this purpose, we need to create our own `JsonRpcClient`:

```
SslContextFactory sec = new SslContextFactory(true);
sec.setValidateCerts(false);
JsonRpcClientWebSocket rpcClient = new JsonRpcClientWebSocket(uri, sec);
KurentoClient kuretoClient = KurentoClient.createFromJsonRpcClient(rpcClient);
```

- Node applications, please take a look to this [page](#).

Second, you have to change the WebSocket URI in your application logic. For instance, in the *hello-world* application within the tutorials, this would be done as follows:

- Java: Changing this line in `HelloWorldApp.java`:

```
final static String DEFAULT_KMS_WS_URI = "wss://localhost:8433/kurento";
```

- Browser JavaScript: Changing this line in `index.js`:

```
const ws_uri = 'wss://' + location.hostname + ':8433/kurento';
```

- Node.js: Changing this line in `server.js`:

```
const ws_uri = "wss://localhost:8433/kurento";
```

CHAPTER 21

WebRTC Statistics

[TODO full review]

21.1 Introduction

WebRTC streams (audio, video, or data) can be lost, and experience varying amounts of network delay. In order to assess the performance of WebRTC applications, it could be required to be able to monitor the WebRTC features of the underlying network and media pipeline.

To that aim, Kurento provides WebRTC statistics gathering for the server-side (Kurento Media Server, KMS). The implementation of this capability follows the guidelines provided in the [W3C WebRTC's Statistics API](#). Therefore, the statistics gathered in the KMS can be divided into two groups:

- `inboundrtp`: statistics on the stream received in the KMS.
- `outboundrtp`: statistics on the stream sent by KMS.

21.2 API description

As usual, WebRTC statistics gathering capability is provided by the KMS and is consumed by means of the different Kurento client implementations (Java, JavaScript clients are provided out of the box). To read these statistics, first it should be enabled using the method `setLatencyStats` of a Media Pipeline object. Using the Kurento Java client this is done as follows:

```
String kmsWsUri = "ws://localhost:8888/kurento";
KurentoClient kurentoClient = KurentoClient.create(kmsWsUri);
MediaPipeline mediaPipeline = kurentoClient.createMediaPipeline();
mediaPipeline.setLatencyStats(true);

// ...
```

... and using the JavaScript client:

```
var kmsWsUri = "ws://localhost:8888/kurento";
kurentoClient(kmsWsUri, function(error, kurentoClient) {
    kurentoClient.create("MediaPipeline", function(error, mediaPipeline) {
        mediaPipeline.setLatencyStats(true, function(error) {
            // ...
        });
    });
});
```

Once WebRTC statistics are enabled, the second step is reading the statistics values using the method `getStats` of a Media Element. For example, to read the statistics of a `WebRtcEndpoint` object in Java:

```
WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(mediaPipeline).build();
MediaType mediaType = ... // it can be MediaType.VIDEO, MediaType.AUDIO, or MediaType.
                           ↪DATA
Map<String, Stats> statsMap = webRtcEndpoint.getStats(mediaType);

// ...
```

... and in JavaScript:

```
mediaPipeline.create("WebRtcEndpoint", function(error, webRtcEndpoint) {
    var mediaType = ... // it can be 'VIDEO', 'AUDIO', or 'DATA'
    webRtcEndpoint.getStats(mediaType, function(error, statsMap) {
        // ...
    });
});
```

Notice that the WebRTC statistics are read as a map. Therefore, each entry of this collection has a key and a value, in which the key is the specific statistic, with a given value at the reading time. Take into account that these values make reference to real-time properties, and so these values vary in time depending on multiple factors (for instance network performance, KMS load, and so on). The complete description of the statistics are defined in the [KMD interface](#) description. The most relevant statistics are listed below:

- `ssrc`: The synchronized source (SSRC).
- `firCount`: Count the total number of Full Intra Request (FIR) packets received by the sender. This metric is only valid for video and is sent by receiver.
- `pliCount`: Count the total number of Packet Loss Indication (PLI) packets received by the sender and is sent by receiver.
- `nackCount`: Count the total number of Negative ACKnowledgement (NACK) packets received by the sender and is sent by receiver.
- `sliCount`: Count the total number of Slice Loss Indication (SLI) packets received by the sender. This metric is only valid for video and is sent by receiver.
- `remb`: The Receiver Estimated Maximum Bitrate (REMB). This metric is only valid for video.
- `packetsLost`: Total number of RTP packets lost for this SSRC.
- `packetsReceived`: Total number of RTP packets received for this SSRC.
- `bytesReceived`: Total number of bytes received for this SSRC.
- `jitter`: Packet Jitter measured in seconds for this SSRC.

- `packetsSent`: Total number of RTP packets sent for this SSRC.
- `bytesSent`: Total number of bytes sent for this SSRC.
- `targetBitrate`: Presently configured bitrate target of this SSRC, in bits per second.
- `roundTripTime`: Estimated round trip time (seconds) for this SSRC based on the RTCP timestamp.
- `audioE2ELatency`: End-to-end audio latency measured in nano seconds.
- `videoE2ELatency`: End-to-end video latency measured in nano seconds.

All in all, the process for gathering WebRTC statistics in the KMS can be summarized in two steps: 1) Enable WebRTC statistics; 2) Read WebRTC. This process is illustrated in the following picture. This diagram also describes the [JSON-RPC](#) messages exchanged between Kurento client and KMS following the [Kurento Protocol](#):

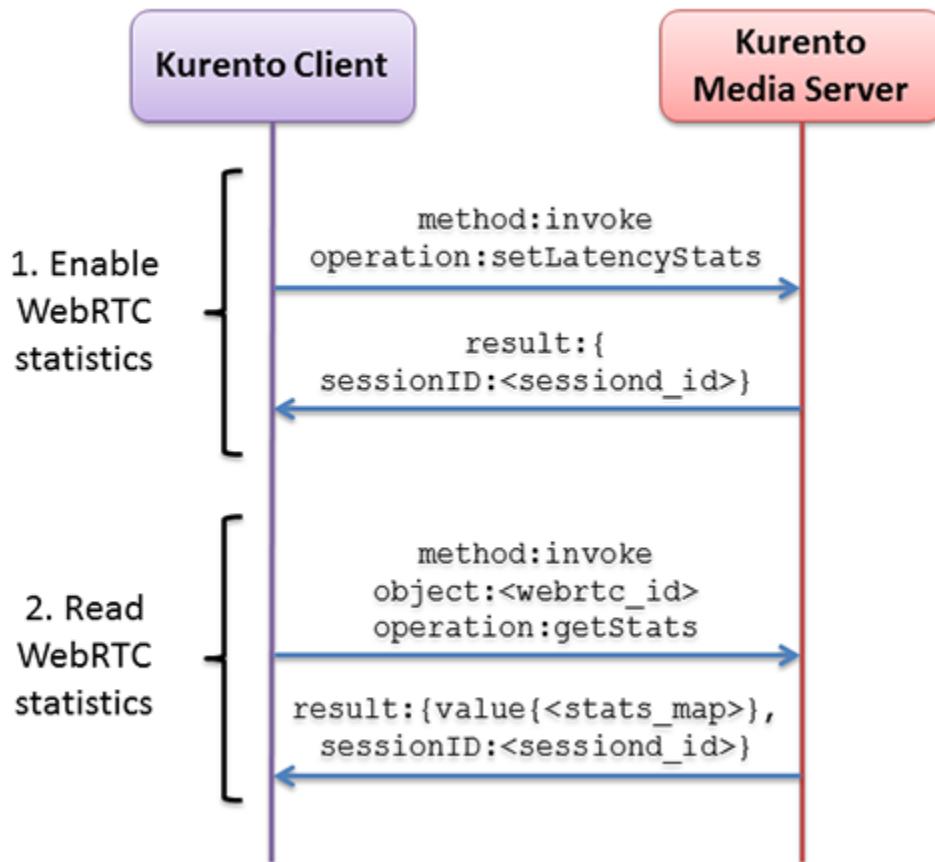


Fig. 21.1: Sequence diagram for gathering WebRTC statistics in KMS

21.3 Example

There is a running tutorial which uses the WebRTC gathering as described before. This demo has been implemented using the JavaScript client and it is available on GitHub: [kurento-loopback-stats](#).

From a the Media Pipeline point of view, this demo application consists in a `WebRtcEndpoint` in loopback. Once the demo is up and running, WebRTC are enabled and gathered with a rate of 1 second.

In addition to the KMS WebRTC statistics, the client-side (i.e. browser WebRtc peer) are also gathered by the application. This is done using the standard method provided by the `peerConnection` object, i.e using its method `getStats`. Please check out the JavaScript logic located in the `index.js` file for implementation details.

Both kinds of WebRTC statistics values (i.e. browser and KMS side) are updated and shown each second in the application GUI, as follows:

Stat	Browser	KMS
SSRC	1725500000	1725500000
Bytes send (browser)	12703404	12461324
Packets send (browser)	12104	12104
PLIs received (browser)	0	0
FIRs received (browser)	1	1
NACKs received (browser)	0	0
RTT	1	--
Jitter	--	0.008811111561954021
Packets Lost	0	0
Fraction lost	--	0
REMB	--	500000

KMS e2e latency: 0.278746 seconds

Stat	Browser	KMS
SSRC	842464803	842464803
Bytes received (browser)	12631812	12423332
Packets received (browser)	10424	10424
PLIs sent (browser)	0	0
FIRs sent (browser)	0	0
NACKs sent (browser)	0	0
Jitter	89	--
RTT	--	0.0006256103515625
Packet lost (browser)	0	--
REMB	--	823624

Fig. 21.2: Statistics results in the kurento-loopback-stats demo GUI

CHAPTER 22

Debug Logging

Kurento Media Server generates log files that are stored in `/var/log/kurento-media-server/`. The content of this folder is as follows:

- `media-server_<timestamp>.<log_number>.<kms_pid>.log`: Output log of a currently running instance of KMS.
- `media-server_error.log`: Errors logged by third-party libraries.
- `logs`: Folder that contains older KMS logs. The logs in this folder are rotated, so they don't fill up all the space available in the disk.

Each line in a log produced by KMS has a fixed structure:

```
[timestamp] [pid] [memory] [level] [component] [filename:loc] [method] [message]
```

- `[timestamp]`: Date and time of the logging message (e.g. `2017-12-31 23:59:59,493295`).
- `[pid]`: Process Identifier of `kurento-media-sever` (e.g. `17521`).
- `[memory]`: Memory address in which the `kurento-media-sever` component is running (e.g. `0x00007fd59f2a78c0`).
- `[level]`: Logging level. This value typically will be `INFO` or `DEBUG`. If unexpected error situations happen, the `WARN` and `ERROR` levels will contain information about the problem.
- `[component]`: Name of the component that generated the log line. E.g. `KurentoModuleManager`, `webrtcendpoint`, or `qtmux`, among others.
- `[filename:loc]`: Source code file name (e.g. `main.cpp`) followed by the line of code number.
- `[method]`: Name of the function in which the log message was generated (e.g. `loadModule()`, `doGarbageCollection()`, etc).
- `[message]`: Specific log information.

For example, when KMS starts correctly, this trace is written in the log file:

```
[timestamp] [pid] [memory] info KurentoMediaServer main.cpp:255 main() Kurento_Media Server started
```

22.1 Logging levels and components

Each different **component** of KMS is able to generate its own logging messages. Besides that, each individual logging message has a severity **level**, which defines how critical (or superfluous) the message is.

These are the different message levels, as defined by the [GStreamer logging library](#):

- **(1) ERROR:** Logs all *fatal* errors. These are errors that do not allow the core or elements to perform the requested action. The application can still recover if programmed to handle the conditions that triggered the error.
- **(2) WARNING:** Logs all warnings. Typically these are *non-fatal*, but user-visible problems that *are expected to happen*.
- **(3) FIXME:** Logs all “fixme” messages. Fixme messages are messages that indicate that something in the executed code path is not fully implemented or handled yet. The purpose of this message is to make it easier to spot incomplete/unfinished pieces of code when reading the debug log.
- **(4) INFO:** Logs all informational messages. These are typically used for events in the system that *happen only once*, or are important and rare enough to be logged at this level.
- **(5) DEBUG:** Logs all debug messages. These are general debug messages for events that *happen only a limited number of times* during an object’s lifetime; these include setup, teardown, change of parameters, etc.
- **(6) LOG:** Logs all log messages. These are messages for events that *happen repeatedly* during an object’s lifetime; these include streaming and steady-state conditions.
- **(7) TRACE:** Logs all trace messages. These messages for events that *happen repeatedly* during an object’s lifetime such as the ref/unref cycles.
- **(8) MEMDUMP:** Log all memory dump messages. Memory dump messages are used to log (small) chunks of data as memory dumps in the log. They will be displayed as hexdump with ASCII characters.

Logging categories and levels can be set by two methods:

- Use the specific command-line argument while launching KMS. For example, run:

```
/usr/bin/kurento-media-server \
--gst-debug-level=3 \
--gst-debug=Kurento*:4,kms*:4
```

- Use the environment variable *GST_DEBUG*. For example, run:

```
export GST_DEBUG="3,Kurento*:4,kms*:4"
/usr/bin/kurento-media-server
```

22.2 Suggested levels

Here are some tips on what logging components and levels could be most useful depending on what is the issue to be analyzed. They are given in the environment variable form, so they can be copied directly into the KMS configuration file, */etc/default/kurento-media-server*:

- Default suggested levels:

```
export GST_DEBUG="3,Kurento*:4,kms*:4"
```

- COMEDIA port discovery:

```
export GST_DEBUG="3, rtpendpoint:4"
```

- ICE candidate gathering:

```
export GST_DEBUG="3,kmsiceniceagent:5,kmswebrtcsession:5,webrtcendpoint:4"
```

Notes:

- *kmscicenseagent* shows messages from the Nice Agent (handling of candidates).
 - *kmswebrtcsession* shows messages from the KMS WebRtcSession (decision logic).
 - *webrtcendpoint* shows messages from the WebRtcEndpoint (very basic logging).

- Event MediaFlow{In|Out} state changes:

```
export GST_DEBUG="3,KurentoMediaElementImpl:5"
```

- Player:

```
export GST_DEBUG="3,playerendpoint:5"
```

- Recorder:

```
export GST_DEBUG="3, KurentoRecorderEndpointImpl:4, recorderendpoint:5, qtmux:5"
```

- REMB congestion control:

```
export GST_DEBUG="3, kmsremb:5"
```

Notes:

- *kmsremb:5* (debug level 5) shows only effective REMB send/recv values.
 - *kmsremb:6* (debug level 6) shows full handling of all source SSRCs.

- RPC calls:

```
export GST_DEBUG="3,KurentoWebSocketTransport:5"
```

- RTP Sync:

```
export GST_DEBUG="3,kmsutils:5,rtpsynchonizer:5,rtpsynccontext:5,  
↳ basertpendpoint:5"
```

- SDP processing:

```
export GST_DEBUG="3,kmssdpsession:4"
```

- Transcoding of media:

```
export GST_DEBUG="3,Kurento*:5,kms*:4,agnosticbin*:7"
```

- Unit tests:

```
export GST_DEBUG="3,check:5"
```

22.3 3rd-party libraries

22.3.1 libnice

libnice is the GLib implementation of *ICE*, the standard method used by *WebRTC* to solve the issue of *NAT Traversal*.

This library has its own logging system that comes disabled by default, but can be enabled very easily. This can prove useful in situations where a developer is studying an issue with the ICE process. However, the debug output of libnice is very verbose, so it makes sense that it is left disabled by default for production systems.

Run KMS with these environment variables defined: `G_MESSAGES_DEBUG` and `NICE_DEBUG`. They must have one or more of these values, separated by commas:

- libnice
- libnice-stun
- libnice-tests
- libnice-socket
- libnice-pseudotcp
- libnice-pseudotcp-verbose
- all

Example:

```
export G_MESSAGES_DEBUG="libnice,libnice-stun"
export NICE_DEBUG="$G_MESSAGES_DEBUG"
/usr/bin/kurento-media-server
```

CHAPTER 23

Contribution Guide

You can contribute to the Kurento project through bug reports or bug fixes in the [Issue Tracker](#), code for new features, or improvements to the [documentation](#). To contribute, write in the the Kurento Public Mailing List providing full information about your contribution and its value.

You must comply with the following contribution guidelines:

- Specify the contents of your contribution either through a detailed bug description, through a Pull Request, or through a patch.
- Specify the licensing restrictions of the code you contribute.
- For newly created code to be incorporated in the Kurento codebase, you must accept Kurento to own the code copyright, so that its open source nature is guaranteed.
- Justify appropriately the need and value of your contribution. The Kurento project has no obligations in relation to accepting contributions from third parties.

The Kurento project leaders have the right of asking for further explanations, tests or validations of any code contributed to the community before it being incorporated into the Kurento codebase. You must be ready to address all these kind of concerns before having your code approved.

CHAPTER 24

Team

The Kurento development team is formed under the [CodeURJC Research Group](#), which belongs to the spanish [Rey Juan Carlos University](#), located in Madrid. This team is financed by the University and by [Naeva Tec](#).

- Micael Gallego is the current lead of the Kurento project. He is involved mainly in the development of the Kurento Java client, Java tutorials, testing infrastructure and Kurento Module Creator.
- Boni García is involved in testing infrastructure and also in writing tutorials and documentation.
- Pablo Fuente works in Kurento Room, OpenVidu, Kurento Java client and the JavaScript client libraries.
- Juan Navarro is our Kurento Media Server guy. He loves working close to the metal with C/C++ and GStreamer.
- Fede Díaz is our DevOps, helping the project with CI infrastructure, performance testing, and so on.
- Patxi Gortázar is the original DevOps of Kurento platform and he's still working across several areas of the project.

CHAPTER 25

Code of Conduct

Open Source Software communities are complex structures where different interests, expectations and visions need to converge and find some kind of equilibrium. In this process, all stakeholders need to understand and comply with a minimal set of rules that guarantee that things happen to the benefit of the community as a whole and that efforts are invested in the most optimal way for that to happen.

Of course, the Kurento team would be happy to have the appropriate resources that allowed providing full and detailed answers to all issues that may arise. Unfortunately this is not the case, and as happens in most OSS projects out there, we need to optimize how efforts are invested and think on the benefit of the community as a whole, instead of ending up satisfying the specific needs of a specific user.

Having said this, it is also clear that complying with a minimum set of netiquette rules is a plus for having questions and issues answered. Most of these rules are common sense, but it may be worthy to state them in a more explicit way so that Kurento users are able to check if they are doing their best to have their issues and questions addressed. Here they go:

- **Be courteous.** Any kind of insult, threat or undervaluation of other people's efforts will only contribute to having your request ignored.
- **Make your homework.** Asking questions such as "*I want to create a system like Skype, please explain me the steps*" may require very extensive answers and you'll probably find that nobody in the community is willing to invest the time to write them, save the case that someone happens to be writing a book on the topic. In general, *don't ask others to make your work*.
- **Follow the [reporting guidelines](#).** When creating a new bug report, following these guidelines will greatly help others to study your issue and look for solutions, which in the end is a positive net for you.
- **Read the documentation first.** Requesting help on issues that are clearly addressed in [the documentation](#) is, in general, a bad practice.
- **Check the [Community Support](#).** Things like opening a new discussion thread on the mailing list dealing with a problem that has already been discussed in another thread, will be probably perceived as slackness by the rest of the community. Avoid this and remember that Google is your friend.
- **Beware of cross-posting.** In general, cross-posting is not considered as a good practice. If for some reason you need to send the same request to different mailing lists, inform in all of them about that providing links to the corresponding threads in the other lists so that the rest of users can check where answers finally arrived.

- **Be constructive.** Claims of the kind “*this design is bad*” or “*you are doing it wrong*” are not particularly useful. If you don’t like something, provide specific suggestions (or better code) showing how things should be improved.
- **Maintain the focus.** Kurento Community Support places have the objective of discussing Kurento-related issues. If you want to have information related to other different projects or to WebRTC in general, contact the corresponding community. Of course, spam shall be punished with immediate banning from the mailing lists.

Complying with these rules will contribute to improve the quality of the Kurento Community as a whole, making it the most helpful source of help and support.

Bests, and have a nice coding time.

CHAPTER 26

Kurento Business Features

26.1 Commercial Support

Kurento is formed by a small team of people. This means that our task pipeline is quite restricted, and most feature or support requests end up being stored in the backlog for a long time. We advance as fast as we can, but time and resources are limited and at the end of the day there is so much that we can do.

If you have some needs that require urgent attention, or want to help with funding development on the Kurento project, we offer consultancy and support services on demand. Please contact us at openvidu@gmail.com and let us know about your project.

CHAPTER 27

Developer Guide

This section is a comprehensive guide for development of *Kurento itself*. The intended reader of this text is any person who wants to get involved in writing code for the Kurento project, or to understand how the source code of this project is structured.

If you are looking to write applications that make use of Kurento, then you should read [Writing Kurento Applications](#).

Table of Contents

- *Developer Guide*
 - *Introduction*
 - *Development tools*
 - *Source code repositories*
 - * *Module dependency graph*
 - * *Module dependency list*
 - *Development 101*
 - * *Libraries*
 - * *Debian packages*
 - * *Build tools*
 - *Working with KMS sources*
 - * *Developing KMS*
 - *Add Kurento repository*
 - *Install development packages*
 - *Download KMS*
 - *Build KMS*

- *Launch KMS*
- *Build and run KMS unit tests*
- *Clean your system*
- * *Working on a forked library*
 - *Full cycle*
 - *In-place linking*
- * *Generating Debian packages*
 - *Dependency resolution: to repo or not to repo*
 - *Package generation script*
 - *Building KMS on Ubuntu 14.04 (Trusty)*
- *How-To*
 - * *How to add or update external libraries*
 - * *How to add new fork libraries*
 - * *Known problems*

27.1 Introduction

Kurento offers a multimedia framework that eases the task of building multimedia applications with the following features:

- **Dynamic WebRTC Media pipelines:** Kurento allows custom media pipelines connected to WebRTC peers like web browsers and mobile apps. These media pipelines can be composed by players, recorders, mixers, etc. and can be changed dynamically when the media is flowing.
- **Client/Server Architecture:** Apps developed with Kurento follow a client/server architecture. Kurento Media Server (KMS) is the server and offers a WebSocket interface implementing the Kurento Protocol, which allows Client Applications to define pipeline topologies.
- **Java and JavaScript Client Applications:** The typical use case of a KMS deployment consists of a three-layer architecture, where the user's browser interacts with the KMS server by means of an intermediate Client Application. There are several official Kurento Client Libraries, supporting the use of Java and JavaScript for the Client Applications. Clients for other languages can be easily implemented following the WebSocket protocol.
- **Third party Modules:** KMS has an extensible architecture based on plugins, which allows third parties to implement modules that can be combined with other built-in or third party modules in the same pipeline. For example, there are modules for Computer Vision with features such as face detection, barcode reading, etc.

This document contains a high level explanation of how to become a KMS developer. Development of *Kurento Client Applications* is out of the scope for this document, and won't be explained here.

27.2 Development tools

This is an overview of the tools and technologies used by KMS:

- The code is written in C and C++ languages.

- The code style is heavily influenced by that of Gtk and GStreamer projects.
- CMake is the construction tool.
- The source code is versioned in several GitHub repositories.
- The officially supported platforms are Ubuntu LTS distributions: 14.04 (Trusty) and 16.04 (Xenial).
- The heart of KMS is the GStreamer multimedia framework.
- In addition to GStreamer, KMS uses other libraries like boost, jsoncpp, libnice, etc.

27.3 Source code repositories

Kurento source code is stored in several GitHub repositories at <https://github.com/Kurento>. Each one of these repositories has a specific purpose and usually contains the code required to build a shared library of the same name.

There are several types of repositories:

- **Fork Repositories:** KMS depends on several open source libraries, the main one being GStreamer. Sometimes these libraries show specific behaviors that need to be tweaked in order to be useful for KMS; other times there are bugs that have been fixed but the patch is not accepted at the upstream source for whatever reason. In these situations, while the official path of feature requests and/or patch submit is still tried, we have created a fork of the affected libraries. The repositories that contain these forked libraries are called “Fork Repositories”.

These are the current Fork Repositories, as of KMS version 6.7:

- [gstreamer](#) (libgstreamer1.5)
- [gst-plugins-base](#)
- [gst-plugins-good](#)
- [gst-plugins-bad](#)
- [gst-plugins-ugly](#)
- [gst-libav](#)
- [jsoncpp](#)
- [libsrtsp](#)
- [libnice](#) (gstreamer1.0-nice, gstreamer1.5-nice)
- [openwebrtc-gst-plugins](#)
- [openh264](#)
- [usrstcp](#)

- **Main Repositories:** The core of KMS is located in Main Repositories. As of version 6.7, these repositories are:

- [kurento-module-creator](#): It is a code generation tool for generating code scaffolding for plugins. This code includes KMS code and Kurento client code. It has mainly Java code.
- [kms-cmake-utils](#): Contains a set of utilities for building KMS with CMake.
- [kms-core](#): Contains the core GStreamer code. This is the base library that is needed for other libraries. It has 80% C code and a 20% C++ code.
- [kms-elements](#): Contains the main elements offering pipeline capabilities like WebRtc, Rtp, Player, Recorder, etc. It has 80% C code and a 20% C++ code.
- [kms-filters](#): Contains the basic video filters included in KMS. It has 65% C code and a 35% C++ code.

- [kms-jsonrpc](#): Kurento protocol is based on JsonRpc, and makes use of a JsonRpc library contained in this repository. It has C++ code.
 - [kurento-media-server](#): Contains the main entry point of KMS. That is, the main() function for the server executable code. This application depends on libraries located in the above repositories. It has mainly C++ code.
- **Omni-Build Repository:** The [kms-omni-build](#) repository is a dummy umbrella for the other KMS Main Repositories. It has no actual code; instead, it only has the required CMake code to allow building the whole KMS project in one go. For this, it gets a copy of the required repositories via Git submodules.
 - **Module Repositories:** KMS is distributed with some basic GStreamer pipeline elements, but other elements are available in form of modules. These modules are stored individually in Module Repositories. Currently, we have the following ones:
 - [kms-chroma](#)
 - [kms-crowddetector](#)
 - [kms-platedetector](#)
 - [kms-pointerdetector](#)
 - **Client Repositories:** Client Applications can be developed in Java, JavaScript with Node.js, or JavaScript directly in the browser. Each of these languages have their support tools made available in their respective repositories.
 - **Tutorial or demo repositories:** There are several repositories that contain sample code for developers that use Kurento or want to develop a custom Kurento module. Currently these are:
 - [kms-datachannelexample](#)
 - [kms-plugin-sample](#)
 - [kms-opencv-plugin-sample](#)
 - [kurento-tutorial-java](#)
 - [kurento-tutorial-js](#)
 - [kurento-tutorial-node](#)

A KMS developer must know how to work with KMS Fork and Main Repositories and understand that each of these have a different development life cycle. The majority of development for KMS will occur at the KMS Main Repositories, while it's unusual to make changes in Fork Repositories except for updating their upstream versions.

27.3.1 Module dependency graph

This graph shows the dependencies between all modules that form part of Kurento:

27.3.2 Module dependency list

As the dependency graph is not strictly linear, there are multiple possible ways to order all modules into a linear dependency list; one possible order would be this one:

Externals:

1. gstreamer
2. libsrtp
3. openh264

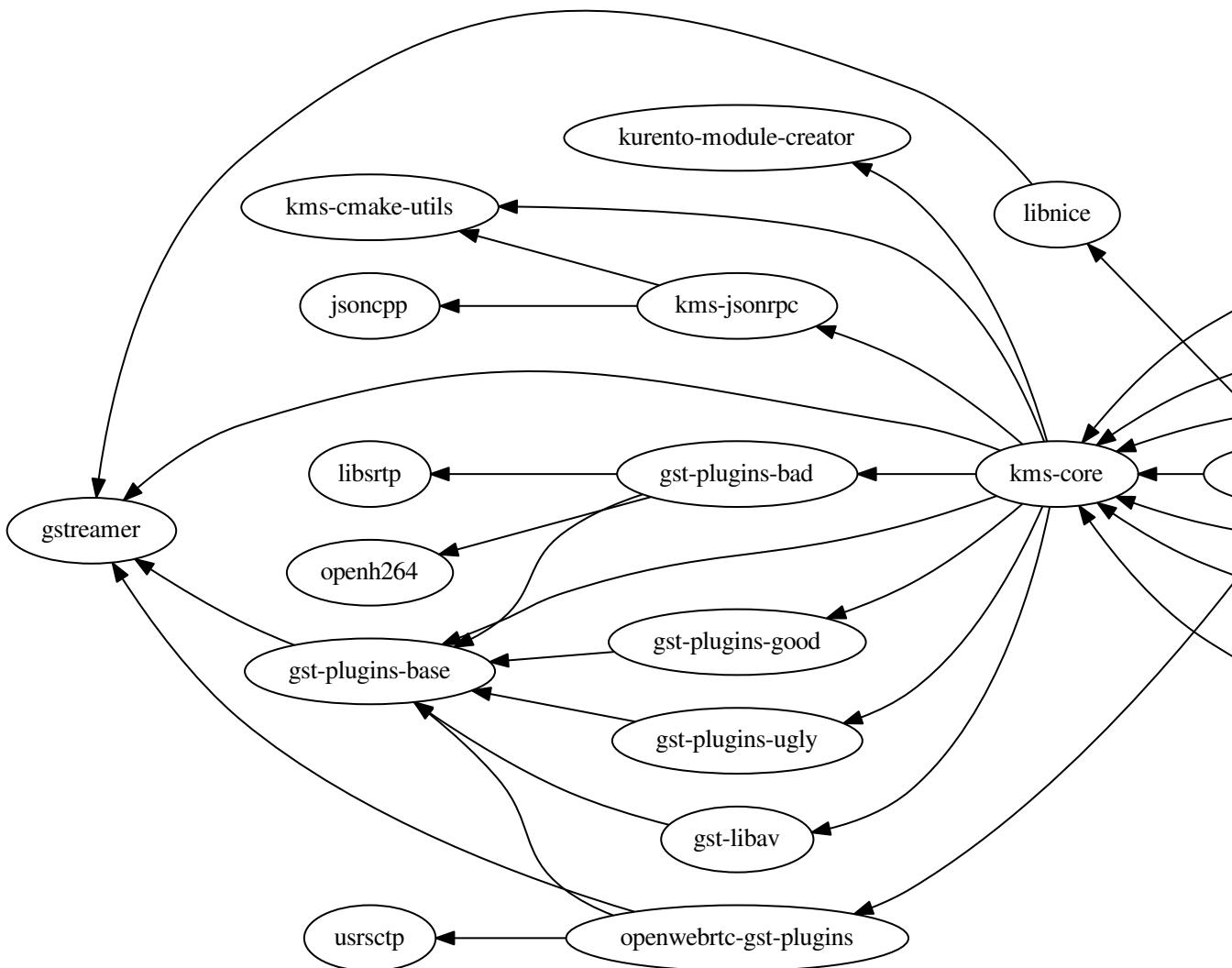


Fig. 27.1: All dependency relationships

4. usrsctp
5. jsoncpp
6. gst-plugins-base
7. gst-plugins-good
8. gst-plugins-ugly
9. gst-plugins-bad
10. gst-libav
11. openwebrtc-gst-plugins
12. libnice

KMS Main + Extra:

1. kurento-module-creator
2. kms-cmake-utils
3. kms-jsonrpc
4. kms-core
5. kms-elements
6. kms-filters
7. kurento-media-server
8. kms-chroma
9. kms-crowddetector
10. kms-datachannelexample
11. kms-platedetector
12. kms-pointerdetector

27.4 Development 101

KMS is a C/C++ project developed with an Ubuntu system as main target, which means that its dependency management and distribution is based on the Debian package system.

27.4.1 Libraries

It is not a trivial task to configure the compiler to use a set of libraries because a library can be composed of several *.so* and *.h* files. To make this task easier, `pkg-config` is a helper tool used when compiling applications and libraries. In short: when a library is installed in a system, it registers itself in the `pkg-config` database with all its required files, which allows to later query those values in order to compile with the library in question.

For example, if you want to compile a C program which depends on GLib 2.0, you can run:

```
gcc -o program program.c $(pkg-config --libs --cflags glib-2.0)
```

27.4.2 Debian packages

In a Debian/Ubuntu system, development libraries are distributed as Debian packages which are made available in public package repositories. When a C or C++ project is developed in these systems, it is usual to distribute it also in Debian packages. It is then possible to install them with the command `apt-get install`, which will handle automatically all the package's dependencies.

When a library is packaged, the result usually consists of several packages. These are some pointers on the most common naming conventions for packages, although they are not always strictly enforced by Debian or Ubuntu maintainers:

- **bin package:** Package containing the binary files for the library itself. Applications are linked against them during development, and they are also loaded in production. The package name starts with *lib*, followed by the name of the library.
- **dev package:** Contains files needed to link with the library during development. The package name starts with *lib* and ends with *-dev*. For example: *libboost-dev* or *libglib2.0-dev*.
- **dbg package:** Contains debug symbols to ease error debugging during development. The package name starts with *lib* and ends with *-dbg*. For example: *libboost-dbg*.
- **doc package:** Contains documentation for the library. Used in development. The package name starts with *lib* and ends with *-doc*. For example: *libboost-doc*.
- **src package:** Package containing the source code for the library. It uses the same package name as the bin version, but it is accessed with the command `apt-get source` instead of `apt-get install`.

27.4.3 Build tools

There are several tools for building C/C++ projects: Autotools, Make, CMake, Gradle, etc. The most prominent tool for building projects is the Makefile, and all the other tools tend to be simply wrappers around this one. KMS uses CMake, which generates native Makefiles to build and package the project. There are some IDEs that recognize CMake projects directly, such as [JetBrains CLion](#) or [Qt Creator](#).

A CMake project consists of several *CMakeLists.txt* files, which define how to compile and package native code into binaries and shared libraries. These files also contain a list of the libraries (dependencies) needed to build the code.

To specify a dependency it is necessary to know how to configure this library in the compiler. The already mentioned `pkg-config` tool is the standard de-facto for this task, so CMake comes with the ability to use `pkg-config` under the hood. There are also some libraries built with CMake that use some specific CMake-only utilities.

27.5 Working with KMS sources

KMS uses CMake to build KMS Main Repositories. Fork repositories contain its own build system (typically Autotools or native Make). This depends on the preferences of the original creators of each project.

KMS Main Repositories declare libraries in CMake, assuming they are or can be installed in the system. For example, **kms-elements** depends on the following items:

- **kms-core**, a library located in a Main Repository.
- **libnice**, a library located in a Fork Repository.
- **ffmpeg**, a public library.

Thus *kms-core*, *ffmpeg* and *libnice* libraries have to be installed in the system before building the project **kms-elements**.

In KMS, we have developed a custom CMake command to search a library in several places. This command is called `generic_find` and it is located in the `kms-cmake-utils` repository.

kms-omni-build is an special project because it is designed to build all KMS Main Repositories from a single entry point. This repo brings the other KMS Main Repositories as Git submodules: it makes KMS development easier because if you build this project, you don't need to manually install the libraries of the other KMS Main Repositories. However, all other development and support libraries must still be installed manually.

To build KMS from sources you first have to decide on which part you want to work:

- **Main KMS development:** You want to make code changes in Main Repositories and test them in your development machine, to see how the changes affect KMS. Or maybe you want to debug KMS with GDB or analyze it with Valgrind.
- **Change a forked library:** You want to update a Fork Repository and check if all is working as expected. In this case, you have two options:
 - Change code in the current fork.
 - Synchronize the fork with a new release of forked library.
- **Generate Debian packages:** To distribute KMS it is necessary to generate Debian packages from KMS Fork and Main Repositories.

As you can see, there are a lot of possibilities. In the next sections we'll explain the best way to build KMS in these different contexts.

27.5.1 Developing KMS

To work directly with KMS source code, or to just build KMS from sources, the easiest way is using the module **kms-omni-build**. Just follow these steps:

- Add the Kurento repository to your system configuration.
- Install development packages: tools like Git, GCC, CMake, etc., and KMS development libraries.
- Clone **kms-omni-build**.
- Build with CMake and Make.
- Run the newly compiled KMS.
- Run KMS tests.

Add Kurento repository

These steps are pretty much the same as those explained in [Local Installation](#), with the only change of using a different package repository.

First Step. Define what version of Ubuntu is installed in your system. Open a terminal and copy **only one** of these lines:

```
# Choose one:  
DISTRO="trusty" # KMS for Ubuntu 14.04 (Trusty)  
DISTRO="xenial" # KMS for Ubuntu 16.04 (Xenial)
```

Second Step. Add the Kurento repository to your system configuration. Run these two commands in the same terminal you used in the previous step:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 5AFA7A83

sudo tee "/etc/apt/sources.list.d/kurento.list" >/dev/null <<EOF
# Kurento Media Server - Pre-Release packages
deb [arch=amd64] http://ubuntu.openvidu.io/dev $DISTRO kms6
EOF
```

Install development packages

Run:

```
PACKAGES=(

    # Development tools
    build-essential
    cmake
    debhelper
    default-jdk
    gdb
    git
    maven
    pkg-config
    valgrind
    wget

    # 'maven-debian-helper' installs an old Maven version in Ubuntu 14.04 (Trusty),
    # so this ensures that the effective version is the one from 'maven'.
    maven-debian-helper-

    # System development libraries
    libboost-dev
    libboost-filesystem-dev
    libboost-log-dev
    libboost-program-options-dev
    libboost-regex-dev
    libboost-system-dev
    libboost-test-dev
    libboost-thread-dev
    libevent-dev
    libglib2.0-dev
    libglibmm-2.4-dev
    libopencv-dev
    libsigc++-2.0-dev
    libsoup2.4-dev
    libssl-dev
    libvpx-dev
    libxml2-utils
    uuid-dev

    # Kurento external libraries
    gstreamer1.5-plugins-base
    gstreamer1.5-plugins-good
    gstreamer1.5-plugins-ugly
    gstreamer1.5-plugins-bad
    gstreamer1.5-libav
    gstreamer1.5-nice
    gstreamer1.5-tools
    gstreamer1.5-x
```

```
libgstreamer1.5-dev
libgstreamer-plugins-base1.5-dev
libnice-dev
openh264-gst-plugins-bad-1.5
openwebrtc-gst-plugins-dev
kmsjsoncpp-dev
ffmpeg
)

sudo apt-get update
sudo apt-get install "${PACKAGES[@]}"
```

Optionally, install the debugging symbols if you will be using a debugger to troubleshoot bugs in KMS. Run:

```
PACKAGES=(
    # Third-party libraries
    libglib2.0-0-dbg
    libssl1.0.0-dbg

    # Kurento external libraries
    gstreamer1.5-plugins-base-dbg
    gstreamer1.5-plugins-good-dbg
    gstreamer1.5-plugins-ugly-dbg
    gstreamer1.5-plugins-bad-dbg
    gstreamer1.5-libav-dbg
    libgstreamer1.5-0-dbg
    libnice-dbg
    libsrtp1-dbg
    openwebrtc-gst-plugins-dbg
    kmsjsoncpp-dbg

    # KMS main components
    kms-jsonrpc-dbg
    kms-core-dbg
    kms-elements-dbg
    kms-filters-dbg
    kurento-media-server-dbg

    # KMS extra modules
    kms-chroma-dbg
    kms-crowddetector-dbg
    kms-platedetector-dbg
    kms-pointerdetector-dbg
)

sudo apt-get update
sudo apt-get install "${PACKAGES[@]}"
```

Download KMS

Run:

```
git clone https://github.com/Kurento/kms-omni-build.git \
&& cd kms-omni-build \
&& git submodule init \
&& git submodule update --recursive --remote
```

Optionally, change to the master branch of each submodule, if you will be developing on each one of those:

```
REF=master
for d in $(find . -maxdepth 1 -mindepth 1 -type d)
do pushd $d ; git checkout "$REF" ; popd ; done
```

You can also set REF to any other branch or tag, such as REF=6.7.1. This will bring the code to the state it had in that version.

Build KMS

Run:

```
TYPE=Debug
mkdir build-$TYPE \
&& cd build-$TYPE \
&& cmake -DCMAKE_BUILD_TYPE=$TYPE -DCMAKE_VERBOSE_MAKEFILE=ON .. \
&& make
```

CMake accepts the following build types: *Debug*, *Release*, *RelWithDebInfo*. So, for a Release build, you would run TYPE=Release instead of TYPE=Debug.

Note: The standard way of compiling a project with CMake is to create a *build* directory and run the `cmake` and `make` commands from there. This allows the developer to have different build folders for different purposes. However **do not use this technique** if you are trying to compile a subdirectory of **kms-omni-build**. For example, if you do this to build *kms-omni-build/kms-core*, no more than one build folder can be present at a time in *kms-omni-build/kms-core/build*. If you want to keep several builds of a single module, it is better to just work on a separate Git clone of that repository.

It is also possible to enable GCC's AddressSanitizer or ThreadSanitizer with these flags:

```
-DENABLE_ANALYZER_ASAN=ON # Enable the AddressSanitizer (aka ASan) memory error detector. Implies ``CMAKE_BUILD_TYPE=Release``.
-DSANITIZE_ADDRESS=ON
-DSANITIZE_THREAD=ON
-DSANITIZE_LINK_STATIC=ON
```

[TODO: finish testing that these modes do actually work]

Launch KMS

Run:

```
export GST_DEBUG='3,Kurento*:4,kms*:4,rtpendpoint:4,webrtcendpoint:4'

kurento-media-server/server/kurento-media-server \
--modules-path=. \
--modules-config-path=./config \
--conf-file=./config/kurento.conf.json \
--gst-plugin-path=.
```

You can set the logging level of specific categories with the option `--gst-debug`, which can be used multiple times, once for each category. Besides that, the global logging level is specified with `--gst-debug-level`. These values can also be defined in the environment variable `GST_DEBUG` (see *Debug Logging*).

Other launch options that could be useful:

```
--logs-path, -d <Path> : Path where rotating log files will be stored  
--log-file-size, -s <Number> : Maximum file size for log files, in MB  
--number-log-files, -n <Number> : Maximum number of log files to keep
```

More launch options, handled by GStreamer: <https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/gst-running.html>

Build and run KMS unit tests

KMS uses the Check unit testing framework for C (<https://libcheck.github.io/check/>). To build and run all tests, change the last one of the build commands from make to make check.

To build and run one specific test, use make <TestName>.check. For example:

```
make test_agnosticbin.check
```

If you want to analyze memory usage with Valgrind, use make <TestName>.valgrind. For example:

```
make test_agnosticbin.valgrind
```

Each test has some amount of debug logging which will get printed; check these messages in the file *.Testing/Temporary/LastTest.log* after running a test suite. To find the starting point of each individual test in this log file, look for the words “*test start*”. Example:

```
webrtcendpoint.c:1848:test_vp8_sendrecv: test start
```

Clean your system

To leave the system in a clean state, remove all KMS packages and related development libraries. Run this command and, for each prompted question, visualize the packages that are going to be uninstalled and press Enter if you agree. This command is used on a daily basis by the development team at Kurento with the option --yes (which makes the process automatic and unattended), so it should be fairly safe to use. However we don’t know what is the configuration of your particular system, and running in manual mode is the safest bet in order to avoid uninstalling any unexpected package.

Run:

```
PACKAGES=(  
    # KMS main components + extra modules  
    '^^(kms|kurento).*'  
  
    # Kurento external libraries  
    ffmpeg  
    '^gir1.2-gst.*1.5'  
    gir1.2-nice-0.1  
    '^^(lib)?gstreamer.*1.5.*'  
    '^lib(nice|s3-2|srtp|usrctp).*'br/>    '^srtp-*'  
    '^openh264(-gst-plugins-bad-1.5)?'  
    '^openwebrtc-gst-plugins.*'  
  
    # System development libraries  
    '^libboost-?(filesystem|log|program-options|regex|system|test|thread) ?-dev'  
    '^lib(glib2.0|glibmm-2.4|opencv|sigc++-2.0|soup2.4|ssl|tesseract|vpx)-dev'
```

```

    uuid-dev
)

# Run a loop over all package names and uninstall them.
for PACKAGE in "${PACKAGES[@]}"; do
    sudo apt-get purge --auto-remove "$PACKAGE" || { echo "Skip unexisting"; }
done

```

27.5.2 Working on a forked library

These are the two typical workflows used to work with fork libraries:

Full cycle

This workflow has the easiest and fastest setup, however it also is the slowest one. To make a change, you would edit the code in the library, then build it, generate Debian packages, and lastly install those packages over the ones already installed in your system. It would then be possible to run KMS and see the effect of the changes in the library.

This is of course an extremely cumbersome process to follow during anything more complex than a couple of edits in the library code.

In-place linking

The other work method consists on changing the system library path so it points to the working copy where the fork library is being modified. Typically, this involves building the fork with its specific tool (which often is Automake), changing the environment variable `LD_LIBRARY_PATH`, and running KMS with such configuration that any required shared libraries will load the modified version instead of the one installed in the system.

This allows for the fastest development cycle, however the specific instructions to do this are very project-dependent. For example, when working on the GStreamer fork, maybe you want to run GStreamer without using any of the libraries installed in the system (see <https://cgit.freedesktop.org/gstreamer/gstreamer/tree/scripts/gst-uninstalled>).

[TODO: Add concrete instructions for every forked library]

27.5.3 Generating Debian packages

You can easily create Debian packages for KMS itself and for any of the forked libraries. Packages are generated by a Python script called `compile_project.py`, which can be found in the `adm-scripts` repository, and you can use it to generate Debian packages locally in your machine. Versions number of all packages are timestamped, so a developer is able to know explicitly which version of each package has been installed at any given time.

Follow these steps to generate Debian packages from any of the Kurento repositories:

1. **(Optional)** Make sure the system is in a clean state. The section [Clean your system](#) explains how to do this.
2. **(Optional)** Add Kurento Packages Repository. The section about [Dependency resolution](#) explains what is the effect of adding the repo, and the section [Add Kurento repository](#) explains how to do this.
3. Install system tools and Python modules. Run:

```

PACKAGES=
# Packaging tools
build-essential
debhelper

```

```
curl
fakeroot
flex
git
libcommons-validator-java
python
python-apt
python-debian
python-git
python-requests
python-yaml
realpath
subversion
wget
)

sudo apt-get update
sudo apt-get install "${PACKAGES[@]}"
```

Note:

- `subversion` (`svn`) is used by `compile_project.py` due to GitHub's lack of support for the `git-archive` protocol (see <https://github.com/isaacs/github/issues/554>).
 - `flex` should be automatically installed by gstreamer, but a bug in package version detection needs to get fixed.
 - `realpath` is used by `adm-scripts/kurento_check_version.sh`.
-

4. Download the Kurento CI tools. Run:

```
git clone https://github.com/Kurento/adm-scripts.git
export PATH="$PWD/adm-scripts:$PATH"
```

5. Download and build packages for the desired module. Run:

```
git clone https://github.com/Kurento/kms-core.git
cd kms-core
sudo PATH="$PWD/.../adm-scripts:$PATH" PYTHONUNBUFFERED=1 \
  ./adm-scripts/kms/compile_project.py \
  --base_url https://github.com/Kurento compile
```

Dependency resolution: to repo or not to repo

The script `compile_project.py` is able to resolve all dependencies for any given module. For each dependency, the following process will happen:

1. If the dependency is already available to `apt-get` from the Kurento Packages Repository, it will get downloaded and installed. This means that the dependency will not get built locally.
2. If the dependency is not available to `apt-get`, its corresponding project will be cloned from the Git repo, built, and packaged itself. This triggers a recursive call to `compile_project.py`, which in turn will try to satisfy all the dependencies corresponding to that sub-project.

It is very important to keep in mind the dependency resolution mechanism that happens in the Python script, because it can affect which packages get built in the development machine. **If the Kurento Packages Repository has been**

configured for “apt-get“, then all dependencies for a given module will be downloaded and installed from the repo, instead of being built. On the other hand, if the Kurento repo has not been configured, then all dependencies will be built from source.

This can have a very big impact on the amount of modules that need to be built to satisfy the dependencies of a given project. The most prominent example is **kurento-media-server**: it basically depends on *everything* else. If the Kurento repo is available to apt-get, then all of KMS libraries will be downloaded and installed. If the repo is not available, then all source code of KMS will get downloaded and built, including the whole GStreamer libraries and other forked libraries.

Note: This only applies to Ubuntu 16.04 (Xenial), for which the official package repositories already contain all required development libraries to build the whole KMS. However, for Ubuntu 14.04 (Trusty) the official repos are missing some required packages, so the Kurento Packages Repository must be configured in the system in order to build all of KMS. Refer to the following sections.

Package generation script

This is the full procedure followed by the *compile_project.py* script:

1. Check if all development dependencies for the given module are installed in the system. This check is done by parsing the file *debian/control* of the project.
2. If some dependencies are not installed, apt-get tries to install them.
3. For each dependency defined in the file *.build.yaml*, the script checks if it got installed during the previous step. If it wasn't, then the script checks if these dependencies can be found in the source code repository given as argument. The script then proceeds to find this dependency's real name and requirements by checking its online copy of the *debian/control* file.
4. Every dependency with source repository, as found in the previous step, is cloned and the script is run recursively with that module.
5. When all development dependencies are installed (either from package repositories or compiling from source code), the initially requested module is built, and its Debian packages are generated and installed.

Building KMS on Ubuntu 14.04 (Trusty)

KMS cannot be built in Trusty without adding the Kurento Packages Repository, because some of the system development libraries are required in a more recent version than the one available by default in the official Ubuntu Trusty repos. This is a non exhaustive list of those required libraries, compared with the versions available in Xenial and in the Kurento repo:

Name	Requirement	In Trusty repo	In Xenial repo	In Kurento repo	Notes
kms-core	libglib2.0-dev (>= 2.46)	2.40	2.48	2.46	[1]
gst-plugins-base	libsoup2.4-dev (>= 2.48)	2.44	2.52	2.50	
libsrtp	libssl-dev (>= 1.0.2)	1.0.1f	1.0.2g	1.0.2g	
gst-plugins-bad	libde265-dev (any)	none	1.0.2	0.9	
gst-plugins-bad	libx265-dev (any)	none	1.9	1.7	
gst-plugins-bad	libaass-dev (>= 0.10.2)	0.10.1	0.13.1	0.10.2	
gst-plugins-bad	libgnutls28-dev, librtmp-dev				[2]
kms-elements	libnice-dev (>= 0.1.13)	0.1.4	0.1.13	0.1.13	
libnice	libgupnp-igd-1.0-dev (>= 0.2.4)	0.2.2	0.2.4	0.2.4	

[1] It actually builds and works fine with 2.40, but the required version of GLib was first raised from 2.40 to 2.42 and later to 2.46 in commits `b10d318b` and `7f703bed`, justified as providing huge performance improvements in `mutex` and `g_object_ref`.

[2] The latter depends on `libgnutls-dev`, which conflicts with the former (only in 14.04). Solution: use `librtmp-dev` from Kurento repo, which doesn't depend on `libgnutls-dev`.

This list of dependencies means that it is not possible to build the whole KMS on Ubuntu Trusty, at least not without the Kurento Packages Repository already configured in the system. But as we mentioned in the previous section, the mere presence of this repo will skip building as many packages as possible if the build script is able to find them already available for install with `apt-get`.

In case that we want to force building the whole KMS libraries and modules -as opposed to downloading them from the repo- the solution to this problem is to clone each module separately, and manually build them one by one, in the order given by their [Module dependency list](#).

27.6 How-To

27.6.1 How to add or update external libraries

Add or change it in these files:

- `debian/control`.
- `CMakeLists.txt`.

27.6.2 How to add new fork libraries

1. Fork the repository.
2. Create a `.build.yaml` file in this repository, listing its project dependencies (if any).
3. Add dependency to `debian/control` in the project that uses it.
4. Add dependency to `CMakeLists.txt` in the project that uses it.

27.6.3 Known problems

- Sometimes the GStreamer fork doesn't compile correctly. Try again.

- Some unit tests can fail, especially if the storage server (which contains some required input files) is having connectivity issues. If tests fail, packages are not generated. To skip tests, edit the file *debian/rules* and change `-DGENERATE_TESTS=TRUE` to `-DGENERATE_TESTS=FALSE -DDISABLE_TESTS=TRUE`.

CHAPTER 28

Continuous Integration

We have two types of repositories containing Debian packages: either Release or Nightly builds of the KMS Main Repositories and KMS Fork Repositories. Each of these types of repos are made available for the currently supported Ubuntu editions, which as of this writing are Ubuntu 14.04 (Trusty) and Ubuntu 16.04 (Xenial).

After some exploration of the different options we had, in the end we settled on the option to have self-contained repos, where all Kurento packages are stored and no dependencies with additional repositories are needed (*).

There is an independent repository for each released version of Kurento, and one single repository for nightly builds:

- Repositories for Ubuntu 14.04 (Trusty):
 - Release: deb [arch=amd64] http://ubuntu.openvidu.io/<Version> trusty kms6
 - Nightly: deb [arch=amd64] http://ubuntu.openvidu.io/dev trusty kms6
- Repositories for Ubuntu 16.04 (Xenial):
 - Release: deb [arch=amd64] http://ubuntu.openvidu.io/<Version> xenial kms6
 - Nightly: deb [arch=amd64] http://ubuntu.openvidu.io/dev xenial kms6

We also have several Continuous-Integration (*CI*) jobs such that every time a patch is accepted in Git's master branch, a new package build of that repository is generated, and uploaded to the nightly repositories.

All scripts used by CI are stored in the Git repo [adm-scripts](#).

CHAPTER 29

Release Procedures

[WORK-IN-PROGRESS]

Table of Contents

- *Release Procedures*
 - *Introduction*
 - *General considerations*
 - *Project Inventory*
 - * *Media Server*
 - *KMS main components*
 - *KMS extra modules*
 - *Kurento external libraries*
 - *C/C++ modules*
 - *Java modules*

29.1 Introduction

Kurento is a project composed of multiple kinds of modules, spanning a multitude of different technologies, languages and sets of *best practices*. Each one of those modules have their own unique style of administration, and the procedures needed to publish a new release can vary a lot.

This document aims to summarize all release procedures that apply to each one of the modules the Kurento project is composed of. The main form of categorization is by technology type: C/C++ based modules, Java modules, JavaScript modules, and others.

29.2 General considerations

- Kurento components to be released must use development versions.
- Dependencies to Kurento libraries can use release or development versions, as needed.
 - In Maven (Java), development versions are indicated by the suffix `-SNAPSHOT` after the version number.
Example: `6.7.0-SNAPSHOT`.
 - In CMake (C/C++), development versions are indicated by the suffix `-dev` after the version number.
Example: `6.7.0-dev`.
- All dependencies to development versions will be changed to a release version during the release procedure. Concerning people will be asked to choose an appropriate release version for each development dependency.
- Kurento uses semantic versioning. Please refer to www.semver.org for more information.
- Tags will be named with the version number of the release. Example: `6.7.0`.

29.3 Project Inventory

Where applicable, these lists are presented in the order given by their module dependencies, such as the one indicated in `development-dependency-list`.

29.3.1 Media Server

KMS main components

- `kurento-module-creator`
- `kms-cmake-utils`
- `kms-jsonrpc`
- `kms-core`
- `kms-elements`
- `kms-filters`
- `kurento-media-server`

KMS extra modules

- `kms-chroma`
- `kms-crowddetector`
- `kms-datachannelexample`
- `kms-platedetector`
- `kms-pointerdetector`

Kurento external libraries

- gstreamer
- libsrtp
- openh264
- usrsctp
- jsoncpp
- gst-plugins-base
- gst-plugins-good
- gst-plugins-ugly
- gst-plugins-bad
- gst-libav
- openwebrtc-gst-plugins
- libnice

Java Public kurento-qa-pom kurento-java kurento-room kurento-tutorial-java kurento-maven-plugin doc-kurento Private kurento-sfu kurento-tree kurento-demo kurento-tutorial-test Javascript Public kurento-client-js kurento-jsonrpc-js kurento-utils-js kurento-tutorial-js kurento-tutorial-node KurentoForks/demo-console Private kurento-demo-js kurento-demo-node kurento-developer-portal

29.4 C/C++ modules

29.5 Java modules

CHAPTER 30

Security Hardening

Hardening is a set of mechanisms that can be activated by turning on several compiler flags, and are commonly used to protect resulting programs against memory corruption attacks. These mechanisms have been standard practice since at least 2011, when the Debian project set on the goal of releasing all their packages with the security hardening build flags enabled¹. Ubuntu has also followed the same policy regarding their own build procedures².

Kurento Media Server had been lagging in this respect, and old releases only implemented the standard [Debian hardening options](#) that are applied by the `dpkg-buildflags` tool by default:

- **Format string checks** (`-Wformat -Werror=format-security`)³. These options instruct the compiler to make sure that the arguments supplied to string functions such as `printf` and `scanf` have types appropriate to the format string specified, and that the conversions specified in the format string make sense.
- **Fortify Source** (`-D_FORTIFY_SOURCE=2`)⁴. When this macro is defined at compilation time, several compile-time and run-time protections are enabled around unsafe use of some glibc string and memory functions such as `memcpy` and `strcpy`, with get replaced by their safer counterparts. This feature can prevent some buffer overflow attacks, but it requires optimization level `-O1` or higher so it is not enabled in Debug builds (which use `-O0`).
- **Stack protector** (`-fstack-protector-strong`)⁵. This compiler option provides a randomized stack canary that protects against *stack smashing* attacks that could lead to buffer overflows, and reduces the chances of arbitrary code execution via controlling return address destinations.
- **Read-Only Relocations (RELRO)** (`-Wl,-z,relro`). This linker option marks any regions of the relocation table as “read-only” if they were resolved before execution begins. This reduces the possible areas of memory in a program that can be used by an attacker that performs a successful *GOT-overwrite* memory corruption exploit. This option works best with the linker’s *Immediate Binding* mode, which forces *all* regions of the relocation table to be resolved before execution begins. However, immediate binding is disabled by default.

Starting from version **6.7**, KMS also implements these extra hardening measurements:

¹ https://wiki.debian.org/Hardening#Notes_on_Memory_Corruption_Mitigation_Methods

² https://wiki.ubuntu.com/Security/Features#Userspace_Hardening

³ <https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

⁴ http://man7.org/linux/man-pages/man7/feature_test_macros.7.html

⁵ <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

- **Position Independent Code (-fPIC) / Position Independent Executable (-fPIE -pie)**⁶. Allows taking advantage of the protection offered by the Kernel. This protects against attacks and generally frustrates memory corruption attacks. This option was initially made the default in Ubuntu 16.10 for some selected architectures, and in Ubuntu 17.10 was finally enabled by default across all architectures supported by Ubuntu.

Note: The *PIC/PIE* feature adds a very valuable protection against attacks, but has one important requisite: *all shared objects must be compiled as position-independent code*. If your shared library has stopped linking with KMS, or your plugin stopped loading at run-time, try recompiling your code with the `-fPIC` option.

- **Immediate Binding (-Wl,-z,now).** This linker option improves upon the *Read-Only Relocations* (RELRO) option, by forcing that all dynamic symbols get resolved at start-up (instead of on-demand). Combined with the RELRO flag, this means that the GOT can be made entirely read-only, which prevents even more types of *GOT-overwrite* memory corruption attacks.

30.1 Hardening validation

Debian-based distributions provide the *hardening-check* tool (package *hardening-includes*), which can be used to check if a binary file (either an executable or a shared library) was properly hardened:

```
$ hardening-check /usr/sbin/sshd
/usr/sbin/sshd:
Position Independent Executable: yes
Stack protected: yes
Fortify Source functions: yes
Read-only relocations: yes
Immediate binding: yes
```

30.2 Hardening in Kurento

Since version 6.7, Kurento Media Server is built with all the mentioned hardening measurements. All required flags are added in the Debian package generation step, by setting the environment variable *DEB_BUILD_MAINT_OPTIONS* to *hardening=all*, as described by [Debian hardening options](#). This variable is injected into the build environment by the CMake module *kms-cmake-utils/CMake/CommonBuildFlags.cmake*, which is included by all modules of KMS.

30.3 PIC/PIE in GCC

This section explains how the Position Independent Code (PIC) and Position Independent Executable (PIE) features are intended to be used (in GCC). Proper use of these is required to achieve correct application of ASLR by the Kernel.

- PIC must be enabled in the compiler for compilation units that will end up linked into a shared library. Note that this includes objects that get packed as a static library before being linked into a shared library.
- PIC must be enabled in the linker for shared libraries.
- PIE or PIC (the former being the recommended one) must be enabled in the compiler for compilation units that will end up linked into an executable file. Note that this includes objects that get packed as a static library before being linked into the executable.

⁶ <https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html>

- PIE must be enabled in the linker for executable files.

Now follows some examples of applying these rules into an hypothetical project composed of one shared library and one executable file:

- The shared library (*SHARED.so*) is composed of 4 source files:
 - *A.c* and *B.c* are compiled first into a static library: *AB.a*. GCC flags: `-fPIC`.
 - *C.c* and *D.c* are compiled into object files *C.o* and *D.o*. GCC flags: `-fPIC`.
 - *AB.a*, *C.o*, and *D.o* are linked into a shared library: *SHARED.so*. GCC flags: `-shared -fPIC`.
- The executable file (*PROGRAM*) is composed of 4 source files:
 - *E.c* and *F.c* are compiled first into a static library: *EF.a*. GCC flags: `-fPIE (*)`.
 - *G.c* and *H.c* are compiled into object files *G.o* and *H.o*. GCC flags: `-fPIE (*)`.
 - *EF.a*, *G.o*, and *H.o* are linked into an executable file: *PROGRAM*. GCC flags: `-pie -fPIE (... -lSHARED)`.

(*): In these cases, it is also possible to compile these files with `-fPIC`, although `-fPIE` is recommended. It is also possible to mix both; for example *E.c* and *F.c* can be compiled with `-fPIC`, while *G.c* and *H.c* are compiled with `-fPIE` (empirically tested, it works fine).

See also:

Options for Code Generation Conventions See `-fPIC`, `-fPIE`.

Options for Linking See `-shared`, `-pie`.

dpkg-buildflags See FEATURE AREAS > hardening > pie.

30.4 PIC/PIE in CMake

CMake has *partial* native support to enable PIC/PIE in a project, via the *POSITION_INDEPENDENT_CODE* and *CMAKE_POSITION_INDEPENDENT_CODE* variables. We consider it “partial” because these variables add the corresponding flags for the compilation steps, but the flag `-pie` is not automatically added to the linker step.

We raised awareness about this issue in their bug tracker: [POSITION_INDEPENDENT_CODE does not add -pie](#).

The effect of setting *POSITION_INDEPENDENT_CODE* to *ON* for a CMake target (or setting *CMAKE_POSITION_INDEPENDENT_CODE* for the whole project), is the following:

- If the target is a library, the flag `-fPIC` is added by CMake to the compilation and linker steps.
- If the target is an executable, the flag `-fPIE` is added by CMake to the compilation and linker steps.

However, CMake is lacking that it *does not* add the flag `-pie` to the linker step of executable targets, so final executable programs are *not* properly hardened for ASLR protection by the Kernel.

Kurento Media Server works around this limitation of CMake by doing this in the CMake configuration:

```
# Use "-fPIC" / "-fPIE" for all targets by default, including static libs
set(CMAKE_POSITION_INDEPENDENT_CODE ON)

# CMake doesn't add "-pie" by default for executables (CMake issue #14983)
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -pie")
```

It would be nice if CMake took over the whole process of generating valid PIC/PIE libraries and executables, by ensuring that all needed flags are added in the correct places. It’s actually very close to that, by only missing the `-pie` flag while linking executable programs.

CHAPTER 31

Writing this documentation

Although each of the Kurento repositories contains a *README* file, the main source of truth for up-to-date information about Kurento is *this* documentation you are reading right now, hosted at Read The Docs.

The final deliverable form of the documentation is obtained by following a 3-step process:

1. The source files are written in a markup language called *reStructuredText* (reST). This format is less known than the popular Markdown, but it is much more powerful and adequate for long-form documentation writing.
2. The source files, written in *reST* format, are processed and converted to other deliverable formats by [Sphinx](#), a documentation processing tool which adds some layers of useful syntax to the *reST* baseline, and also takes care of generating all the documents in their final form.
3. Finally, the generated HTML files are hosted in Read The Docs, the service of choice for lots of open-source projects. Actually Read The Docs is not only the hosting, but they also perform the Sphinx generation step itself: they provide a Continuous Integration system that watches a Git repository and triggers a new documentation build each time it detects changes.

Kurento documentation files are written using both the basic features of *reST*, and the extra features that are provided by Sphinx. The *reST* language itself can be learned by checking any reference documents such as the [reStructuredText quick reference](#) and the [reStructuredText Primer](#).

Sphinx adds its own set of useful markup elements, to make *reST* even more useful for writing documentation. To learn more about this, the most relevant section in their documentation is [Sphinx Markup Constructs](#).

Besides the extra markup added by Sphinx, there is also the possibility to use *Sphinx Extensions*, which each one does in turn add its own markup to extend even more the capabilities of the language. For example, as of this writing we are using [sphinx.ext.graphviz](#) and [sphinx-ext-wikipedia](#) extensions, to easily insert links to Wikipedia articles and embedded diagrams in the documents.

31.1 Kurento documentation Style Guide

31.1.1 Paragraph conventions

- **Line breaks:** *Don't* break the lines. The documentation is a prose text, and not source code, so the typical restrictions of line length don't apply here. Use automatic line breaks in your editor, if you want. The overall flow of the text should be dictated by the width of the screen where the text is being presented, and not by some arbitrary line length limit.

31.1.2 Inline markup

- Paths, file names, package names, acronyms, and in general human-oriented words are emphasized with single asterisks (as in `*word*`). Sample phrases:

```
This document talks about Kurento Media Server (*KMS*).  
All dependency targets are defined in the *CMakeLists.txt* file.  
You need to install *libboost-dev* for development.
```

- Code, commands, arguments, environment variables, commit hashes, and in general machine-oriented keywords are emphasized with double backquotes (as in ```word```). Sample phrases:

```
Use ``apt-get install`` to set up all required packages.  
Set ``CMAKE_BUILD_TYPE=Debug`` to build with debugging symbols.  
The argument ``--gst-debug`` can be used to control the logging level.
```

- There is no difference between using *single asterisks* (`*word*`), and *single backquotes* (``word``); they get rendered as *italic text*. So, always use asterisks when wanting to emphasize some text.
- As opposed to Markdown, underscores (as in `_word_`) *don't get rendered*, so don't use them to emphasize text.

31.1.3 Header conventions

- **Header separation:** Always separate each header from the preceding paragraph, by using **3** empty lines. The only exception to this rule is when two headers come together (e.g. a document title followed by a section title); in that case, they are separated by just **1** empty line.
- **Header shape:** *reST* allows to express section headers with any kind of characters that form an underline shape below the section title. We follow these conventions for Kurento documentation files:

1. Level 1 (Document title). Use = above and below:

```
=====  
Level 1  
=====
```

2. Level 2. Use = below:

```
Level 2  
=====
```

3. Level 3. Use -:

```
Level 3  
-----
```

4. Level 4. Use ~:

```
Level 4
~~~~~
```

5. Level 5. Use ":

```
Level 5
*****
```

31.2 Sphinx documentation generator

Our Sphinx-based project is hosted in the [doc-kurento](#) repository. Here, the main entry point for running Sphinx is the Makefile, based on the template that is provided for new projects by Sphinx itself. This Makefile is customized to attend our particular needs, and implements several targets:

- **init-workdir**. This target constitutes the first step to be run before most other targets. Our documentation source files contain substitution keywords in some parts, in the form `| KEYWORD |`, which is expected to be substituted by some actual value during the generation process. Currently, the only keyword in use is `VERSION`, which must be expanded to the actual version of the documentation being built.

For example, here is the `VERSION` keyword when substituted with its final value: `6.7.1`.

Yes, Sphinx does already include a substitutions feature by itself, and the keyword `VERSION` is precisely one of the supported substitutions. Sadly, this feature of Sphinx is very unreliable. For example, it won't work if the keyword is located inside a literal code block, or inside an URL. So, we must resort to performing the substitutions by ourselves if we want reliable results.

The `source` folder is copied into the `build` directory, and then the substitutions take place over this copy.

- **langdoc**. This target creates the automatically generated reference documentation for each [Kurento Client](#). Currently, this means the Javadoc and Jsdoc documentations for Java and Js clients, respectively. The Kurento client repositories are checked out in the same version as specified by the documentation's version file, or in the master branch if no such version tag exists. Then, the client stubs of the [Kurento API](#) are automatically generated, and from the resulting source files, the appropriate documentation is automatically generated too.

The `langdoc` target is usually run before the `html` target, in order to end up with a complete set of HTML documents that include all the reST documentation with the Javadoc/Jsdoc sections.

- **dist**. This target is a convenience shortcut to generate the documentation in the most commonly requested formats: HTML, PDF and EPUB. All required sub-targets will be run and the resulting files will be left as a compressed package in the `dist/` subfolder.
- **ci-readthedocs**. This is a special target that is meant to be called exclusively by our Continuous Integration system. The purpose of this job is to manipulate all the documentation into a state that is a valid input for the Read The Docs CI system. Check the next section for more details.

31.3 Read The Docs builds

It would be great if Read The Docs worked by simply calling the command `make html`, as then we would be able to craft a Makefile that would build the complete documentation in one single step (by making the Sphinx's `html` target dependent on our `init-workdir` and `langdoc`). But alas, they don't work like this; instead, they run Sphinx directly from their Python environment, rendering our Makefile as useless in their CI.

In order to overcome this limitation, we opted for the simple solution of handling RTD a specifically-crafted Git repository, with the contents that they expect to find. This works as follows:

1. Read The Docs has been configured to watch for changes in the [doc-kurento-readthedocs](#) repo, instead of *doc-kurento*.
2. The *init-workdir* and *langdoc* targets run locally from our *doc-kurento* repo.
3. The resulting files from those targets are copied as-is to the *doc-kurento-readthedocs* repository.
4. Everything is then committed and pushed to this later repo, thus triggering a new RTD build.

CHAPTER 32

Congestion Control (RMCAT)

RTP Media Congestion Avoidance Techniques (RMCAT) is an [IETF Working Group](#) that aims to develop new protocols which can manage network congestion in the context of RTP streaming. The goals for any congestion control algorithm are:

- Preventing network collapse due to congestion.
- Allowing multiple flows to share the network fairly.

A good introduction to RMCAT, its history and its context can be found in this blog post by Mozilla: [What is RMCAT congestion control, and how will it affect WebRTC?](#).

As a result of this Working Group, several algorithms have been proposed so far by different vendors:

- By Cisco: [NADA, A Unified Congestion Control Scheme for Real-Time Media](#).
- By Ericsson: [SCReAM, Self-Clocked Rate Adaptation for Multimedia](#).
- By Google: [GCC, Google Congestion Control Algorithm for Real-Time Communication](#).

32.1 Google Congestion Control

Google's GCC is the RMCAT algorithm of choice for WebRTC, and it's used by WebRTC-compatible web browsers. In GCC, both sender and receiver of an RTP session collaborate to find out a realistic estimation of the network bandwidth that is available:

- The RTP sender generates special timestamps called `abs-send-time`, and sends them as part of the RTP packet's Header Extension.
- The RTP receiver generates a new type of RTCP feedback message called *Receiver Estimated Maximum Bitrate* (REMB).

32.2 REMB

There has been some misconceptions about what is the exact meaning of the value that is carried by REMB messages. In short, REMB is used for reporting the aggregate bandwidth estimates of the receiver, across all media streams that are sharing the same RTP session.

Sources:

- In [\[rmcat\]](#) comments on [draft-alvestrand-rmcat-remb-02](#), this question is responded:

```
> - the maximum bitrate, is it defined as the maximum bitrate for  
> a particular stream, or the maximum bitrate for the whole "Session" ?
```

As per GCC, REMB can be used for reporting the sum of the receiver estimate across all media streams that share the same end-to-end path. I supposed the SSRCs of the multiple media streams that make up the aggregate estimate are reported in the block.

- In [\[rtcweb\]](#) REMB with many ssrc entries, a similar question is answered, talking about an explicit example:

```
> If Alice is sending Bob an audio stream (SSRC 1111) and a video stream  
> (SSRC 2222) and Bob sends a REMB feedback message with:  
> - bitrate: 100000 (100kbits/s)  
> - ssrccs: 1111, 2222  
>  
> Does it mean that Alice should limit the sum of her sending audio and  
> video bitrates to 100kbits/s? or does it mean that Alice can send  
> 100kbits/s of audio and 100kbits/s of video (total = 200)?
```

The way it was originally designed, it meant that the total should be 100 Kbits/sec. REMB did not take a position on how the sender chose to allocate bits between the SSRCs, only the total amount of bits sent.

CHAPTER 33

NAT Types and NAT Traversal

Table of Contents

- *NAT Types and NAT Traversal*
 - *Basic Concepts*
 - * *IP connection*
 - * *Inbound NAT connection*
 - *Types of NAT*
 - * *Full Cone NAT*
 - * *(Address-)Restricted Cone NAT*
 - * *Port-Restricted Cone NAT*
 - * *Symmetric NAT*
 - *NAT Traversal*
 - * *Do-It-Yourself hole punching*
 - * *PySTUN*

Sources:

- Symmetric NAT and It's Problems
- Peer-to-Peer Communication Across Network Address Translators
- The hole trick - How Skype & Co. get round firewalls

33.1 Basic Concepts

33.1.1 IP connection

An IP connection is uniquely identified by a connection “quadruplet” consisting of two tuples:

- (Source IP address, source port number).
- (Destination IP address, destination port number).

Concept:

```
(SRC_IP, SRC_PORT) -> (DST_IP, DST_PORT)
```

Note:

- (SRC_IP, SRC_PORT) is the IP tuple of a local machine making a connection.
 - -> denotes the direction of the communication.
 - (DST_IP, DST_PORT) is the IP tuple of a remote machine receiving the connection.
-

33.1.2 Inbound NAT connection

A NAT establishes **inbound** rules which are used to convert an IP tuple existing in the external network (*WAN*) to an IP tuple existing in the internal network (*LAN*). Usually, the external network is the public internet while the internal network is the local network. These inbound rules have the form of a *hash table* (or *map*): for each given key, there is a resulting value:

- The key of the table is an IP quadruplet, formed by the WAN IP tuple and the destination IP tuple.
- The value of the table is the LAN IP tuple.

In other words, a NAT creates an equivalence between external combinations of IP addresses and ports, and internal IP addresses and ports.

Typically, these NAT rules were created automatically during an earlier outbound connection from the LAN to some external machine: it's at that moment when the NAT inserts a new entry into its table. Later, this entry in the NAT table is used to know which local machine needs to receive the response that the external machine might send. Rules created in this way are called “dynamic rules”; on the other hand, “static rules” can be created by the administrator in order to set a fixed table for a given local machine.

Also, it is worth noting that the port number used in the internal side of the network will also be kept by the NAT on the external side, and it won't change for each new connection that the local machine does to any external server. This is the crucial difference between Cone NAT and Symmetric NAT, as explained later.

Concept:

```
(LAN_IP, LAN_PORT) <= [ (WAN_IP, LAN_PORT) <- (REM_IP, REM_PORT) ]
```

Note:

- (REM_IP, REM_PORT) is the **source** IP tuple of a remote machine making a connection.
 - (WAN_IP, LAN_PORT) is the **destination** IP tuple on the *external side* of the NAT receiving the connection.
 - <= denotes the resolution of the NAT mapping.
-

- $(\text{LAN_IP}, \text{LAN_PORT})$ is the **destination** IP tuple on the *internal side* of the NAT for a local machine receiving the connection.
 - Note how the *same* port (LAN_PORT) is used in the internal and the external sides of the NAT. This is the most common case, only differing for Symmetric NAT.
-

33.2 Types of NAT

33.2.1 Full Cone NAT

This type of NAT allows inbound connections from *any source IP address* and *any source port*, as long as the destination tuple exists in any previously created rule. Typically, these rules are statically created beforehand by an administrator. These are the kind of rules that are used to configure *Port Forwarding* (aka. “*opening the ports*”) in most consumer routers.

Concept:

```
(\text{LAN\_IP}, \text{LAN\_PORT}) <= [ (\text{WAN\_IP}, \text{LAN\_PORT}) <- (\ast, \ast) ]
```

Note:

- \ast means that any value could be used here.
-

33.2.2 (Address-)Restricted Cone NAT

This type of NAT allows inbound connections from *any source port* of a *specific source IP address*. Typically, an inbound rule of this type was previously created dynamically, when the local machine initiated a connection to a remote one.

To connect with a local machine which is behind an Address-Restricted Cone NAT, it is first required that the local machine performs an outbound connection to the remote one. This way, a dynamic rule will be created for the destination IP tuple, allowing the remote machine to connect back.

Concept:

```
1. (\text{LAN\_IP}, \text{LAN\_PORT}) => [ (\text{WAN\_IP}, \text{LAN\_PORT}) -> (\text{REM\_IP}, \text{REM\_PORT}) ]
2. (\text{LAN\_IP}, \text{LAN\_PORT}) <= [ (\text{WAN\_IP}, \text{LAN\_PORT}) <- (\text{REM\_IP}, \ast) ]
```

Note:

- \Rightarrow denotes the creation of a new rule in the NAT table.
 - The **destination** IP address REM_IP in step 1 must be the same as the **source** IP address REM_IP in step 2.
-

33.2.3 Port-Restricted Cone NAT

This is the most restrictive type of NAT: it only allows inbound connections from a *specific source port* of a *specific source IP address*. Again, an inbound rule of this type was previously created dynamically, when the local machine initiated an outbound connection to a remote one.

To connect with a local machine which is behind a Port-Restricted Cone NAT, it is first required that the local machine performs an outbound connection to the remote one. This way, a dynamic rule will be created for the destination IP tuple, allowing the remote machine to connect back.

Concept:

```
1. (LAN_IP, LAN_PORT) => [(WAN_IP, LAN_PORT) -> (REM_IP, REM_PORT)]
2. (LAN_IP, LAN_PORT) <= [(WAN_IP, LAN_PORT) <- (REM_IP, REM_PORT)]
```

Note:

- The **destination** IP address REM_IP in step 1 must be the same as the **source** IP address REM_IP in step 2.
 - The **destination** port REM_PORT in step 1 must be the same as the **source** port REM_PORT in step 2.
-

33.2.4 Symmetric NAT

This type of NAT behaves in the same way of a Port-Restricted Cone NAT, with a crucial difference: for each outbound connection to a different remote machine, the NAT assigns a **new random source port** on the external side. This means that two consecutive connections to two different machines will have two different external source ports, even if the internal source IP tuple is the same for both of them.

This is also the only case where the ICE connectivity protocol will find Peer Reflexive candidates which differ from the Server Reflexive ones, due to the differing ports between the connection to the STUN server and the direct connection between peers.

Concept:

```
1. (LAN_IP, LAN_PORT) => [(WAN_IP, WAN_PORT) -> (REM_IP, REM_PORT)]
2. (LAN_IP, LAN_PORT) <= [(WAN_IP, WAN_PORT) <- (REM_IP, REM_PORT)]
```

Note:

- When the outbound connection is done in step 1, WAN_PORT gets defined as a new random port number, assigned for each new remote IP tuple (REM_IP, REM_PORT).
-

33.3 NAT Traversal

The NAT mechanism is implemented in a vast majority of home and corporate routers, and it completely prevents the possibility of running any kind of server software in a local machine which sits behind these kinds of devices. NAT make impossible for a remote client to be the active peer and send any kind of request to the server. NAT Traversal, also known as *Hole Punching*, is the procedure of opening an inbound port in the NAT tables of these routers.

To connect with a local machine which is behind an Address-Restricted Cone NAT, a Port-Restricted Cone NAT or a Symmetric NAT, it is first required that the local machine performs an outbound connection to the remote one. This way, a dynamic rule will be created for the destination IP tuple, allowing the remote machine to connect back.

In order to tell one machine when it has to perform an outbound connection to another one, and the destination IP tuple it must use, the typical solution is to use a signaling service such as STUN. This is usually managed by a third machine, a server sitting on a public internet address. It retrieves the external IP and port of each peer, and gives that information to the other peers that want to communicate.

To connect with a machine which is behind a Full Cone NAT, however, any direct connection to the external IP tuple will work.

STUN/TURN requirement:

- Symmetric to Symmetric: *TURN*.
- Symmetric to Port-Restricted Cone: *TURN*.
- Symmetric to Address-Restricted Cone: *STUN* (but probably not reliable).
- Symmetric to Full Cone: *STUN*.
- Everything else: *STUN*.

33.3.1 Do-It-Yourself hole punching

It is very easy to test the NAT capabilities in a local network. To do this, you need access to two machines:

1. One sitting behind a NAT. We'll call this the host **A**.
2. One directly connected to the internet, with no firewall. This is host **B**.

Set some helper variables: the *public* IP address of each host, and their listening ports:

```
A_IP="11.11.11.11" # Public IP address of the NAT which hides the host A
A_PORT="1111"       # Listening port on the host A
B_IP="22.22.22.22" # Public IP address of the host B
B_PORT="2222"       # Listening port of the host B
```

1. **A** starts listening for data. Leave this running in A:

```
nc -4nul "$A_PORT"
```

2. **B** tries to send data, but the NAT in front of **A** will discard the packets. Run in B:

```
echo "TEST" | nc -4nu -q 1 -p "$B_PORT" "$A_IP" "$A_PORT"
```

3. **A** performs a hole punch, forcing its NAT to create a new inbound rule. **B** awaits for the UDP packet, for verification purposes.

Run in B:

```
sudo tcpdump -n -i eth0 "src host $A_IP and udp dst port $B_PORT"
```

Run in A:

```
sudo hping3 --count 1 --udp --baseport "$A_PORT" --keep --destport "$B_PORT" "$B_IP"
```

4. **B** tries to send data again. Run in B:

```
echo "TEST" | nc -4nu -q 1 -p "$B_PORT" "$A_IP" "$A_PORT"
```

Note:

- The difference between a Cone NAT and a Symmetric NAT can be detected during step 3. If the `tcpdump` command on **B** shows a source port equal to `$A_PORT`, then the NAT is respecting the source port chosen by the application, which means that it is one of the Cone NAT types. However, if `tcpdump` shows that the source

port is different from \$A_PORT, then the NAT is changing the source port during outbound mapping, which means that it is a Symmetric NAT.

- In the case of a Cone NAT, the data sent from **B** should arrive correctly at **A** after step 4.
 - In the case of a Symmetric NAT, the data sent from **B** won't arrive at **A** after step 4, because \$A_PORT is the wrong destination port. If you write the correct port (as discovered in step 3) instead of \$A_PORT, then the data should arrive to **A**.
-

33.3.2 PySTUN

PySTUN is a tool that uses STUN servers in order to try and detect what is the type of the NAT, when ran from a machine behind it.

Currently it has been best updated in one of its forks, so we suggest using that instead of the version from the original creator. To install and run:

```
git clone https://github.com/konradkonrad/pystun.git pystun-konrad
cd pystun-konrad/
git checkout research
mv README.md README.rst
sudo python setup.py install
pystun
```

CHAPTER 34

Glossary

This is a glossary of terms that often appear in discussion about multimedia transmissions. Some of the terms are specific to [GStreamer](#) or [Kurento](#), and most of them are described and linked to their RFC, W3C or Wikipedia documents.

Agnostic media One of the big problems of media is that the number of variants of video and audio codecs, formats and variants quickly creates high complexity in heterogeneous applications. So Kurento developed the concept of an automatic converter of media formats that enables development of *agnostic* elements. Whenever a media element's source is connected to another media element's sink, the Kurento framework verifies if media adaption and transcoding is necessary and, if needed, it transparently incorporates the appropriate transformations making possible the chaining of the two elements into the resulting [Pipeline](#).

AVI Audio Video Interleaved, known by its initials AVI, is a multimedia container format introduced by Microsoft in November 1992 as part of its Video for Windows technology. AVI files can contain both audio and video data in a file container that allows synchronous audio-with-video playback. AVI is a derivative of the Resource Interchange File Format (RIFF).

See also:

Bower Bower is a package manager for the web. It offers a generic solution to the problem of front-end package management, while exposing the package dependency model via an API that can be consumed by a build stack.

Builder Pattern The builder pattern is an object creation software design pattern whose intention is to find a solution to the telescoping constructor anti-pattern. The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.

See also:

CORS Cross-origin resource sharing is a mechanism that allows JavaScript code on a web page to make XMLHttpRequests to different domains than the one the JavaScript originated from. It works by adding new HTTP headers that allow servers to serve resources to permitted origin domains. Browsers support these headers and enforce the restrictions they establish.

See also:

[enable-cors.org](#) Information on the relevance of CORS and how and when to enable it.

DOM Document Object Model is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents.

EOS End Of Stream is an event that occurs when playback of some media source has finished. In Kurento, some elements will raise an `EndOfStream` event.

GStreamer GStreamer is a pipeline-based multimedia framework written in the C programming language.

H.264 A Video Compression Format. The H.264 standard can be viewed as a “family of standards” composed of a number of profiles. Each specific decoder deals with at least one such profiles, but not necessarily all.

See also:

[RFC 6184](#) RTP Payload Format for H.264 Video (This RFC obsoletes [RFC 3984](#))

HTTP The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

See also:

[RFC 2616](#) Hypertext Transfer Protocol – HTTP/1.1

ICE Interactive Connectivity Establishment (ICE) is a technique used to achieve [NAT Traversal](#). ICE makes use of the [STUN](#) protocol and its extension, [TURN](#). ICE can be used by any application that makes use of the SDP Offer/Answer model..

See also:

[RFC 5245](#) Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols

IMS IP Multimedia Subsystem (IMS) is the 3GPP’s Mobile Architectural Framework for delivering IP Multimedia Services in 3G (and beyond) Mobile Networks.

See also:

[RFC 3574](#) Transition Scenarios for 3GPP Networks

jQuery jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

JSON JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is designed to be easy to understand and write for humans and easy to parse for machines.

JSON-RPC JSON-RPC is a simple remote procedure call protocol encoded in JSON. JSON-RPC allows for notifications and for multiple calls to be sent to the server which may be answered out of order.

Kurento Kurento is a platform for the development of multimedia-enabled applications. Kurento is the Esperanto term for the English word ‘stream’. We chose this name because we believe the Esperanto principles are inspiring for what the multimedia community needs: simplicity, openness and universality. Some components of Kurento are the [Kurento Media Server](#), the [Kurento API](#), the [Kurento Protocol](#), and the [Kurento Client](#).

Kurento API An object oriented API to create media pipelines to control media. It can be seen as an interface to Kurento Media Server. It can be used from the Kurento Protocol or from Kurento Clients.

Kurento Client A programming library (Java or JavaScript) used to control an instance of **Kurento Media Server** from an application. For example, with this library, any developer can create a web application that uses Kurento Media Server to receive audio and video from the user web browser, process it and send it back again over Internet. The Kurento Client libraries expose the [Kurento API](#) to application developers.

Kurento Protocol Communication between KMS and clients by means of [JSON-RPC](#) messages. It is based on [WebSocket](#) that uses [JSON-RPC](#) v2.0 messages for making requests and sending responses.

KMS

Kurento Media Server **Kurento Media Server** is the core element of Kurento since it responsible for media transmission, processing, loading and recording.

Maven [Maven](#) is a build automation tool used primarily for Java projects.

Media Element A **Media Element** is a module that encapsulates a specific media capability. For example **RecorderEndpoint**, **PlayerEndpoint**, etc.

Media Pipeline A Media Pipeline is a chain of media elements, where the output stream generated by one element (source) is fed into one or more other elements input streams (sinks). Hence, the pipeline represents a “machine” capable of performing a sequence of operations over a stream.

Media Plane In a traditional IP Multimedia Subsystem, the handling of media is conceptually splitted in two layers. The layer that handles the media itself -with functionalities such as media transport, encoding/decoding, and processing- is called Media Plane.

See also:

[Signaling Plane](#)

MP4 MPEG-4 Part 14 or MP4 is a digital multimedia format most commonly used to store video and audio, but can also be used to store other data such as subtitles and still images.

See also:

Multimedia Multimedia is concerned with the computer controlled integration of text, graphics, video, animation, audio, and any other media where information can be represented, stored, transmitted and processed digitally. There is a temporal relationship between many forms of media, for instance audio, video and animations. There 2 are forms of problems involved in

- Sequencing within the media, i.e. playing frames in correct order or time frame.
- Synchronization, i.e. inter-media scheduling. For example, keeping video and audio synchronized or displaying captions or subtitles in the required intervals.

See also:

Multimedia container format Container or wrapper formats are meta-file formats whose specification describes how different data elements and metadata coexist in a computer file. Simpler multimedia container formats can contain different types of audio formats, while more advanced container formats can support multiple audio and video streams, subtitles, chapter-information, and meta-data, along with the synchronization information needed to play back the various streams together. In most cases, the file header, most of the metadata and the synchro chunks are specified by the container format.

See also:

NAT

Network Address Translation Network address translation (NAT) is the technique of modifying network address information in Internet Protocol (IP) datagram packet headers while they are in transit across a traffic routing device for the purpose of remapping one IP address space into another.

See also:

NAT-T

NAT Traversal NAT traversal (sometimes abbreviated as NAT-T) is a general term for techniques that establish and maintain Internet protocol connections traversing network address translation (NAT) gateways, which break end-to-end connectivity. Intercepting and modifying traffic can only be performed transparently in the absence of secure encryption and authentication.

See also:

[NAT Types and NAT Traversal](#) Entry in our Knowledge Base.

NAT Traversal White Paper White paper on NAT-T and solutions for end-to-end connectivity in its presence

Node.js [Node.js](#) is a cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows and Linux with no changes.

npm [npm](#) is the official package manager for [Node.js](#).

OpenCL [OpenCL](#) is the standard framework for cross-platform, parallel programming of heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors.

OpenCV OpenCV (Open Source Computer Vision Library) is a BSD-licensed open source computer vision and machine learning software library. OpenCV aims to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception.

Pad, Media A Media Pad is an element's interface with the outside world. Data streams from the MediaSource pad to another element's MediaSink pad.

See also:

[GStreamer Pad](#) Definition of the Pad structure in GStreamer

PubNub [PubNub](#) is a publish/subscribe cloud service for sending and routing data. It streams data to global audiences on any device using persistent socket connections. PubNub has been designed to deliver data with low latencies to end-user devices. These devices can be behind firewalls, NAT environments, and other hard-to-reach network environments. PubNub provides message caching for retransmission of lost signals over unreliable network environments. This is accomplished by maintaining an always open socket connection to every device.

QR QR code (Quick Response Code) is a type of two-dimensional barcode. that became popular in the mobile phone industry due to its fast readability and greater storage capacity compared to standard UPC barcodes.

See also:

REMB **Receiver Estimated Maximum Bitrate** (REMB) is a type of RTCP feedback message that a RTP receiver can use to inform the sender about what is the estimated reception bandwidth currently available for the stream itself. Upon reception of this message, the RTP sender will be able to adjust its own video bitrate to the conditions of the network. This message is a crucial part of the *Google Congestion Control* (GCC) algorithm, which provides any RTP session with the ability to adapt in cases of network congestion.

The *GCC* algorithm is one of several proposed algorithms that have been proposed by an IETF Working Group named *RTP Media Congestion Avoidance Techniques* (RMCAT).

See also:

[What is RMCAT congestion control, and how will it affect WebRTC?](#)

[draft-alvestrand-rmcat-remb](#) RTCP message for Receiver Estimated Maximum Bitrate

[draft-ietf-rmcat-gcc](#) A Google Congestion Control Algorithm for Real-Time Communication

REST Representational state transfer (REST) is an architectural style consisting of a coordinated set of constraints applied to components, connectors, and data elements, within a distributed hypermedia system. The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his [doctoral dissertation](#).

See also:

RTCP The RTP Control Protocol (RTCP) is a sister protocol of the [RTP](#), that provides out-of-band statistics and control information for an RTP flow.

See also:

[RFC 3605](#) Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP)

RTP Real-time Transport Protocol (RTP) is a standard packet format designed for transmitting audio and video streams on IP networks. It is used in conjunction with the [RTP Control Protocol](#). Transmissions using the RTP audio/video profile (RTP/AVP) typically use [SDP](#) to describe the technical parameters of the media streams.

See also:

[RFC 3550](#) RTP: A Transport Protocol for Real-Time Applications

Same-origin policy The “same-origin policy” is a web application security model. The policy permits scripts running on pages originating from the same domain to access each other’s [DOM](#) with no specific restrictions, but prevents access to [DOM](#) on different domains.

See also:

[SDP](#)

Session Description Protocol

SDP Offer/Answer The **Session Description Protocol** (SDP) is a text document that describes the parameters of a streaming media session. It is commonly used to describe the characteristics of RTP streams (and related protocols such as RTSP).

The **SDP Offer/Answer** model is a negotiation between two peers of a unicast stream, by which the sender and the receiver share the set of media streams and codecs they wish to use, along with the IP addresses and ports they would like to use to receive the media.

This is an example SDP Offer/Answer negotiation. First, there must be a peer that wishes to initiate the negotiation; we’ll call it the *offerer*. It composes the following SDP Offer and sends it to the other peer -which we’ll call the *answerer*-:

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=Example sender
c=IN IP4 127.0.0.1
t=0 0
m=audio 5006 RTP/AVP 96
a=rtpmap:96 opus/48000/2
a=sendonly
m=video 5004 RTP/AVP 103
a=rtpmap:103 H264/90000
a=sendonly
```

Upon receiving that Offer, the *answerer* studies the parameters requested by the *offerer*, decides if they can be satisfied, and composes an appropriate SDP Answer that is sent back to the *offerer*:

```
v=0
o=- 3696336115 3696336115 IN IP4 192.168.56.1
s=Example receiver
c=IN IP4 192.168.56.1
t=0 0
m=audio 0 RTP/AVP 96
a=rtpmap:96 opus/48000/2
a=recvonly
m=video 31278 RTP/AVP 103
a=rtpmap:103 H264/90000
a=recvonly
```

The SDP Answer is the final step of the SDP Offer/Answer Model. With it, the *answerer* agrees to some of the parameter requested by the *offerer*, but not all.

In this example, `audio 0` means that the *answerer* rejects the audio stream that the *offerer* intended to send; also, it accepts the video stream, and the `a=recvonly` acknowledges that the *answerer* will exclusively act as a receiver, and won't send any stream back to the other peer.

See also:

[Anatomy of a WebRTC SDP](#)

[RFC 4566](#) SDP: Session Description Protocol

[RFC 4568](#) Session Description Protocol (SDP) Security Descriptions for Media Streams

Semantic Versioning [Semantic Versioning](#) is a formal convention for specifying compatibility using a three-part version number: major version; minor version; and patch.

Signaling Plane It is the layer of a media system in charge of the information exchanges concerning the establishment and control of the different media circuits and the management of the network, in contrast to the transfer of media, done by the Signaling Plane. Functions such as media negotiation, QoS parametrization, call establishment, user registration, user presence, etc. as managed in this plane.

See also:

[Media Plane](#)

Sink, Media A Media Sink is a MediaPad that outputs a Media Stream. Data streams from a MediaSource pad to another element's MediaSink pad.

SIP Session Initiation Protocol (SIP) is a [signaling plane](#) protocol widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP) networks. SIP works in conjunction with several other application layer protocols:

- [SDP](#) for media identification and negotiation.
- [RTP](#), [SRTP](#) or [WebRTC](#) for the transmission of media streams.
- A [TLS](#) layer may be used for secure transmission of SIP messages.

See also:

Source, Media A Media Source is a Media Pad that generates a Media Stream.

SPA

Single-Page Application A single-page application is a web application that fits on a single web page with the goal of providing a more fluid user experience akin to a desktop application.

Sphinx [Sphinx](#) is a documentation generation system. Text is first written using [reStructuredText](#) markup language, which then is transformed by Sphinx into different formats such as PDF or HTML. This is the documentation tool of choice for the Kurento project.

See also:

[Easy and beautiful documentation with Sphinx](#)

Spring Boot [Spring Boot](#) is Spring's convention-over-configuration solution for creating stand-alone, production-grade Spring based applications that can you can "just run". It embeds Tomcat or Jetty directly and so there is no need to deploy WAR files in order to run web applications.

SRTCP SRTCP provides the same security-related features to RTCP, as the ones provided by SRTP to RTP. Encryption, message authentication and integrity, and replay protection are the features added by SRTCP to [RTCP](#).

See also:

[SRTP](#)

SRTP Secure RTP is a profile of RTP (*Real-time Transport Protocol*), intended to provide encryption, message authentication and integrity, and replay protection to the RTP data in both unicast and multicast applications. Similarly to how RTP has a sister RTCP protocol, SRTP also has a sister protocol, called Secure RTCP (or *SRTCP*).

See also:

[RFC 3711](#) The Secure Real-time Transport Protocol (SRTP)

SSL Secure Socket Layer. See [TLS](#).

STUN STUN stands for **Session Traversal Utilities for NAT**. It is a standard protocol ([IETF RFC 5389](#)) used by *NAT* traversal algorithms to assist hosts in the discovery of their public network information. If the routers between peers use full cone, address-restricted, or port-restricted NAT, then a direct link can be discovered with STUN alone. If either one of the routers use symmetric NAT, then a link can be discovered with STUN packets only if the other router does not use symmetric or port-restricted NAT. In this later case, the only alternative left is to discover a relayed path through the use of *TURN*.

Trickle ICE Extension to the *ICE* protocol that allows ICE agents to send and receive candidates incrementally rather than exchanging complete lists. With such incremental provisioning, ICE agents can begin connectivity checks while they are still gathering candidates and considerably shorten the time necessary for ICE processing to complete.

See also:

[draft-ietf-ice-trickle](#) Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol

TLS Transport Layer Security (TLS) and its predecessor Secure Socket Layer (SSL).

See also:

[RFC 5246](#) The Transport Layer Security (TLS) Protocol Version 1.2

TURN TURN stands for **Traversal Using Relays around NAT**. Like *STUN*, it is a network protocol ([IETF RFC 5766](#)) used to assist in the discovery of paths between peers on the Internet. It differs from STUN in that it uses a public intermediary relay to act as a proxy for packets between peers. It is used when no other option is available since it consumes server resources and has an increased latency. The only time when TURN is necessary is when one of the peers is behind a symmetric NAT and the other peer is behind either a symmetric NAT or a port-restricted NAT.

VP8 VP8 is a video compression format created by On2 Technologies as a successor to VP7. Its patents rights are owned by Google, who made an irrevocable patent promise on its patents for implementing it and released a specification under the [Creative Commons Attribution 3.0 license](#).

See also:

[RFC 6386](#) VP8 Data Format and Decoding Guide

WebM WebM is an open media file format designed for the web. WebM files consist of video streams compressed with the VP8 video codec and audio streams compressed with the Vorbis audio codec. The WebM file structure is based on the Matroska media container.

WebRTC WebRTC is a set of protocols, mechanisms and APIs that provide browsers and mobile applications with Real-Time Communications (RTC) capabilities over peer-to-peer connections.

See also:

[WebRTC Working Draft](#)

WebSocket WebSocket specification (developed as part of the HTML5 initiative) defines a full-duplex single socket connection over which messages can be sent between client and server.

CHAPTER 35

Indices and tables

- genindex
- search

Index

A

Agnostic media, **407**
AVI, **407**

B

Bower, **407**
Builder Pattern, **407**

C

CORS, **407**

D

DOM, **408**

E

EOS, **408**

G

GStreamer, **408**

H

H.264, **408**
HTTP, **408**

I

ICE, **408**
IMS, **408**

J

jQuery, **408**
JSON, **408**
JSON-RPC, **408**

K

KMS, **408**
Kurento, **408**
Kurento API, **408**
Kurento Client, **408**
Kurento Media Server, **409**

Kurento Protocol, **408**

M

Maven, **409**
Media
 Pad, **410**
 Pipeline, **409**
 Sink, **412**
 Source, **412**
Media Element, **409**
Media Pipeline, **409**
Media Plane, **409**
MP4, **409**
Multimedia, **409**
Multimedia container format, **409**

N

NAT, **409**
NAT Traversal, **409**
NAT-T, **409**
Network Address Translation, **409**
Node.js, **410**
npm, **410**

O

OpenCL, **410**
OpenCV, **410**

P

Pad, Media, **410**
Plane
 Media, **409**
 Signaling, **412**
PubNub, **410**

Q

QR, **410**

R

REMB, **410**

REST, [410](#)

RFC

RFC 2616, [408](#)
RFC 3550, [411](#)
RFC 3574, [408](#)
RFC 3605, [410](#)
RFC 3711, [413](#)
RFC 3984, [408](#)
RFC 4566, [412](#)
RFC 4568, [412](#)
RFC 4585, [219](#)
RFC 5245, [408](#)
RFC 5246, [413](#)
RFC 5766, [236](#)
RFC 6184, [408](#)
RFC 6386, [413](#)

RTCP, [410](#)

RTP, [411](#)

S

Same-origin policy, [411](#)

SDP, [411](#)

SDP Offer/Answer, [411](#)

Semantic Versioning, [412](#)

Session Description Protocol, [411](#)

Signaling Plane, [412](#)

Single-Page Application, [412](#)

Sink, Media, [412](#)

SIP, [412](#)

Source, Media, [412](#)

SPA, [412](#)

Sphinx, [412](#)

Spring Boot, [412](#)

SRTCP, [412](#)

SRTP, [413](#)

SSL, [413](#)

STUN, [413](#)

T

TLS, [413](#)

Trickle ICE, [413](#)

TURN, [413](#)

V

VP8, [413](#)

W

WebM, [413](#)

WebRTC, [413](#)

WebSocket, [413](#)