Kurento Documentation

Release 6.15.0

Kurento

Nov 11, 2020

User Documentation

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*

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

- *Release Notes*
- *Developer Documentation*

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.

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 are based on composable elements such as players, recorders, mixers, etc. that can be mix-and-matched, activated, or deactivated at any point in time, even when the media is already 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**: The Kurento Media Server has an extensible architecture based on plugins, which allows third parties to implement modules that can be added to their media pipelines. This allows integration of media processing algorithms to any WebRTC application, like integrating Computer Vision, Augmented Reality, video indexing, and speech analysis. All is needed is to create a new Kurento element, and use it in any already existing media pipeline.

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.

The code of Kurento Media Server is open source, released under the terms of Apache License Version 2.0 and available on GitHub.

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.
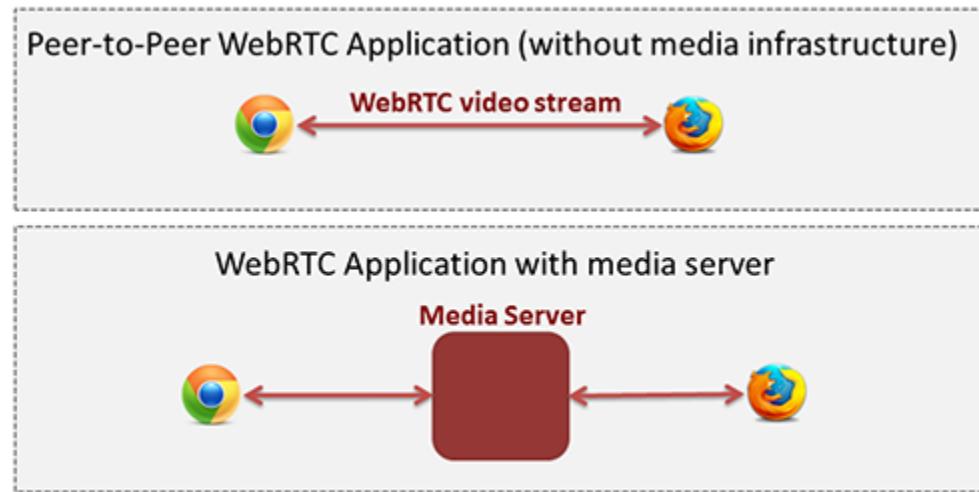
Fig. 1: *Peer-to-peer WebRTC approach vs. WebRTC through a media server*

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.

- 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.
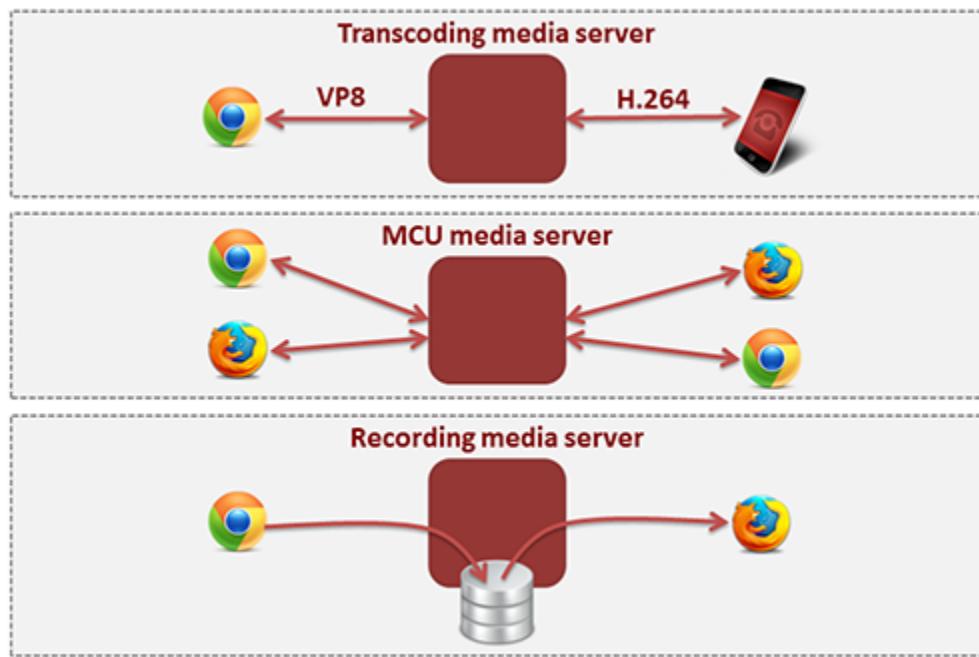
Fig. 2: *Typical WebRTC Media Server capabilities*

- 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.

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 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.
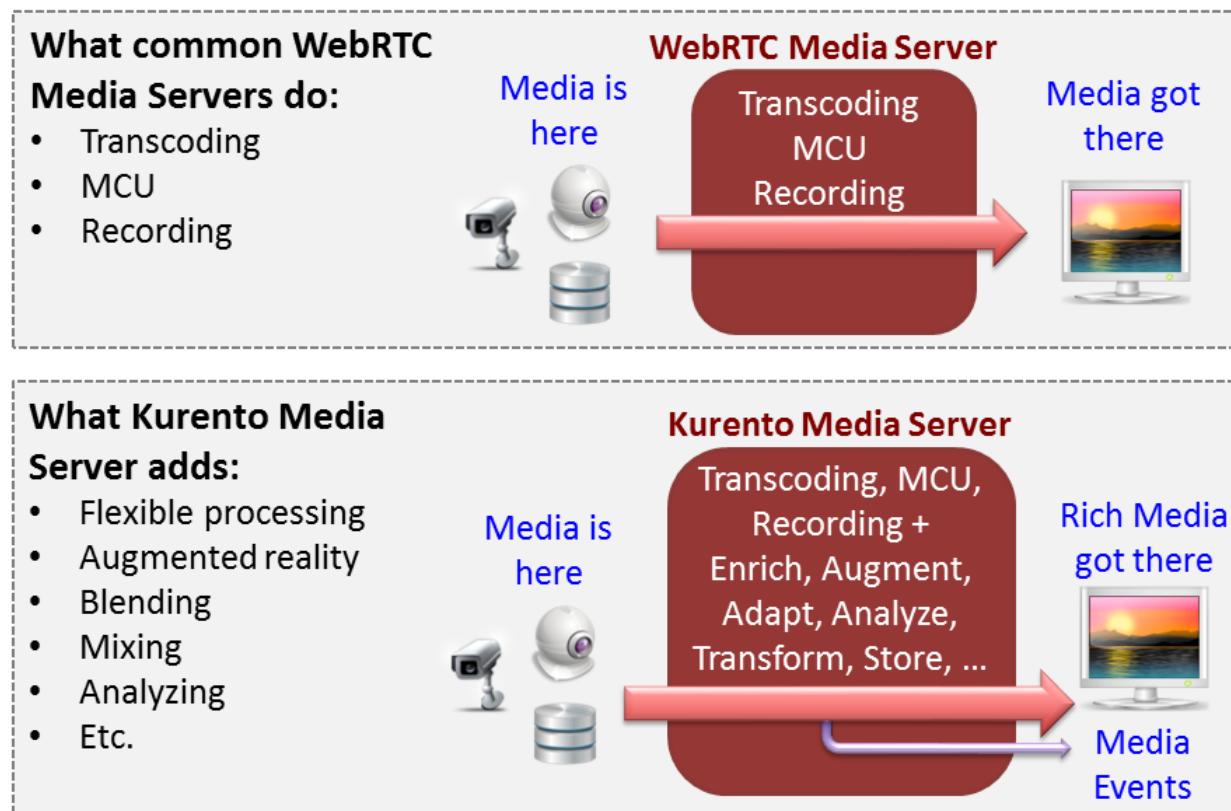
Fig. 3: *Kurento Media Server capabilities*

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.

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: https://openvidu.io/

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 Open-Vidu 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, get in contact with us to request *Commercial Support*.

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 :-)

Installation Guide

Kurento Media Server (KMS) can be installed in multiple ways

- *Using 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.

- *Using the Kurento Docker images*. Docker allows to run Kurento in any host machine, so for example it's possible to run KMS on top of a Fedora or CentOS system. In theory it could even be possible to run under Windows, but so far that possibility hasn't been explored by the Kurento team, so you would be at your own risk.

- *Setting up a local installation* with `apt-get install`. This method allows to have total control of the installation process.

Besides installing KMS, a common need is also *installing a STUN/TURN server*, especially if KMS or any of its clients are located behind a *NAT* router or firewall.

If you want to try nightly builds of KMS, then head to the section *Installing Nightly Builds*.

4.1 Amazon Web Services

Kurento provides a sample *AWS CloudFormation* template file for Amazon Web Services (AWS). 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-6.15.0.yaml
```

4. Follow through the steps of the configuration wizard:

   4.1. **Stack name**: A descriptive name for your Stack.

   4.2. **InstanceType**: Choose an appropriate size for your instance. Check the different ones.

   4.3. **KeyName**: You need to create an RSA key beforehand in order to access the instance. Check AWS documentation on how to create one.

   4.4. **SSHLocation**: For security reasons you may need to restrict SSH traffic to allow connections only from specific locations. For example, from your home or office.

   4.5. **TurnUser**: User name for the TURN relay.

   4.6. **TurnPassword**: Password required to use the TURN relay.

   > **Note:** The template file includes *Coturn* as a *STUN* server and *TURN* relay. 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 *Application Servers*.

Note: The Kurento CF template is written to deploy **on the default VPC** (see the Amazon Virtual Private Cloud docs). There is no VPC selector defined in this template, so you won't see a choice for it during the AWS CF wizard. If you need more flexibility than what this template offers, you have two options:

A. Manually create an EC2 instance, assigning all the resources as needed, and then using the other installation methods to set Kurento Media Server up on it: *Docker image*, *Local Installation*.

B. Download the current CF from the link above, and edit it to create your own custom version with everything you need from it.

4.2 Docker image

Kurento's Docker Hub contains images built from each KMS release. Just head to the kurento-media-server Docker Hub page, and follow the instructions you'll find there.

4.3 Local Installation

With this method, you will install Kurento Media Server from the native Ubuntu package repositories made available by the Kurento project. KMS has explicit support for two Long-Term Support (*LTS*) versions of Ubuntu: **Ubuntu 16.04 (Xenial)** and **Ubuntu 18.04 (Bionic)** (64-bits only).

Open a terminal and run these commands:

1. Make sure that GnuPG is installed.

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    gnupg
```

2. Add the Kurento repository to your system configuration.

   Run these commands:

```
# Import the Kurento repository signing key
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 5AFA7A83

# Get Ubuntu version definitions
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add the repository to Apt
sudo tee "/etc/apt/sources.list.d/kurento.list" >/dev/null <<EOF
# Kurento Media Server - Release packages
deb [arch=amd64] http://ubuntu.openvidu.io/6.15.0 $DISTRIB_CODENAME kms6
EOF
```

3. Install KMS:

Note: This step applies **only for a first time installation**. If you already have installed Kurento and want to upgrade it, follow instead the steps described here: *Local Upgrade*.

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    kurento-media-server
```

   This will install the release version of Kurento Media Server.

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
```

Log messages from KMS will be available in `/var/log/kurento-media-server/`. For more details about KMS logs, check *Debug Logging*.

4.4 Local Upgrade

To upgrade a local installation of Kurento Media Server, you have to write the new version number into the file `/etc/apt/sources.list.d/kurento.list`, which was created during *Local Installation*. After editing that file, you can choose between 2 options to actually apply the upgrade:

A. **Upgrade all system packages**.

   This is the standard procedure expected by Debian & Ubuntu maintainer methodology. Upgrading all system packages is a way to ensure that everything is set to the latest version, and all bug fixes & security updates are applied too, so this is the most recommended method:

```
sudo apt-get update && sudo apt-get dist-upgrade
```

   However, don't do this inside a Docker container. Running *apt-get upgrade* or *apt-get dist-upgrade* is frowned upon by the Docker best practices; instead, you should just move to a newer version of the Kurento Docker images.

B. **Uninstall the old Kurento version**, before installing the new one.

Note however that **apt-get is not good enough** to remove all of Kurento packages. We recommend that you use *aptitude* for this, which works much better than *apt-get*:

```
sudo aptitude remove '?installed?version(kurento)'

sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    kurento-media-server
```

Note: Be careful! If you fail to upgrade **all** Kurento packages, you will get wrong behaviors and **crashes**. Kurento is composed of several packages:

- *kurento-media-server*
- *kurento-module-creator*
- *kms-core*
- *kms-elements*
- *kms-filters*
- *libnice10*
- *libusrsctp*
- *openh264*
- *openwebrtc-gst-plugins*
- And more

To use a newer version **you have to upgrade all Kurento packages**, not only the first one.

4.5 STUN/TURN server install

Working with WebRTC *requires* developers to learn and have a good understanding about everything related to NAT, ICE, STUN, and TURN. If you don't know about these, you should start reading here: *About NAT, ICE, STUN, TURN*.

Kurento Media Server, just like any WebRTC endpoint, will work fine on its own, for *LAN* connections or for servers which have a public IP address assigned to them. However, sooner or later you will want to make your application work in a cloud environment with NAT firewalls, and allow KMS to connect with remote clients. At the same time, remote clients will probably want to connect from behind their own *NAT* router too, so your application needs to be prepared to perform *NAT Traversal* in both sides. This can be done by setting up a *STUN* server or a *TURN* relay, and configuring it **in both KMS and the client browser**.

These links contain the information needed to finish configuring your Kurento Media Server with a STUN/TURN server:

- *How to install Coturn?*
- *How to test my STUN/TURN server?*
- *How to configure STUN/TURN in Kurento?*

4.6 Check your installation

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

```
$ ps -fC kurento-media-server
UID        PID  PPID  C STIME TTY          TIME CMD
kurento   7688     1  0 13:36 ?        00:00:00 /usr/bin/kurento-media-server
```

Unless configured otherwise, KMS will listen on the port TCP 8888, to receive RPC Requests and send RPC Responses by means of the *Kurento Protocol*. Use this command to verify that this port is open and listening for incoming packets:

```
$ sudo netstat -tupln | grep -e kurento -e 8888
tcp6  0  0  :::8888  :::*  LISTEN  7688/kurento-media-
```

You can change these parameters in the file /etc/kurento/kurento.conf.json.

To check whether KMS is up and listening for connections, use the following command:

```
curl \
  --include \
  --header "Connection: Upgrade" \
  --header "Upgrade: websocket" \
  --header "Host: 127.0.0.1:8888" \
  --header "Origin: 127.0.0.1" \
  http://127.0.0.1:8888/kurento
```

You should get a response similar to this one:

```
HTTP/1.1 500 Internal Server Error
Server: WebSocket++/0.7.0
```

Ignore the "*Server Error*" message: this is expected, and it actually proves that KMS is up and listening for connections.

If you need to automate this, you could write a script similar to healthchecker.sh, the one we use in Kurento Docker images.

Installing Nightly 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:** Nightly 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, it's also possible (although unlikely) that these builds might include undocumented changes, regressions, bugs or deprecations. It's safer to be conservative and avoid using nightly builds in a production environment, unless you have a strong reason to do it.

> **Note:** If you are looking to build KMS from the source code, then you should check the section aimed at development of *KMS itself*: *Build from sources*.

5.1 Kurento Media Server

5.1.1 Docker image

Kurento's Docker Hub contains images built from each KMS nightly version. Just head to the kurento-media-server-dev Docker Hub page, and follow the instructions you'll find there. The nightly images work exactly the same as Kurento Docker release images, but just using nightly builds instead of release ones.

5.1.2 Local Installation

The steps to install a nightly version of Kurento Media Server are pretty much the same as those explained in *Local Installation* – with the only change of using *dev* instead of a version number, in the file /etc/apt/sources. list.d/kurento.list.

Open a terminal and run these commands:

1. Make sure that GnuPG is installed.

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    gnupg
```

2. Add the Kurento repository to your system configuration.

    Run these commands:

```
# Import the Kurento repository signing key
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 5AFA7A83

# Get Ubuntu version definitions
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add the repository to Apt
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 $DISTRIB_CODENAME kms6
EOF
```

3. Install KMS:

Note: This step applies **only for a first time installation**. If you already have installed Kurento and want to upgrade it, follow instead the steps described here: *Local Upgrade*.

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    kurento-media-server
```

This will install the nightly version of Kurento Media Server.

5.2 Kurento Java Client

The development builds of the Kurento Java Client are made available for Maven in https://maven.openvidu.io/ 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>
```

(continues on next page)

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

```xml
<dependency>
  <groupId>org.kurento</groupId>
  <artifactId>kurento-client</artifactId>
  <version>6.15.0-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:

```xml
<settings>
  ...
  <profiles>
    <profile>
      <id>kurento</id>
      <repositories>
```

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

Tutorials

This section contains tutorials showing how to use the Kurento framework to build different types of *WebRTC* and multimedia applications.

Note: These tutorials have been created with **learning purposes**. They don't have comprehensive error handling, or any kind of sophisticated session management. As such, *these tutorials should not be used in production environments*; they only show example code for you to study, in order to achieve what you want with your own code.

Use at your own risk!

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.

  To run the Java tutorials, you need to first install the Java JDK and Maven:

  ```
  sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
      git \
      default-jdk \
      maven
  ```

  Java tutorials are written on top of Spring Boot, so they already include most features expected from a full-fledged service, such as a web server or logging support.

  Spring Boot is also able to create a "fully executable jar", a standalone executable built out of the application package. This executable comes already with support for commands such as *start*, *stop*, or *restart*, so it can be used as a system service with either *init.d* (System V) and *systemd*. For more info, refer to the Spring Boot documentation and online resources such as this Stack Overflow answer.

- **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: 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 of 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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}: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:
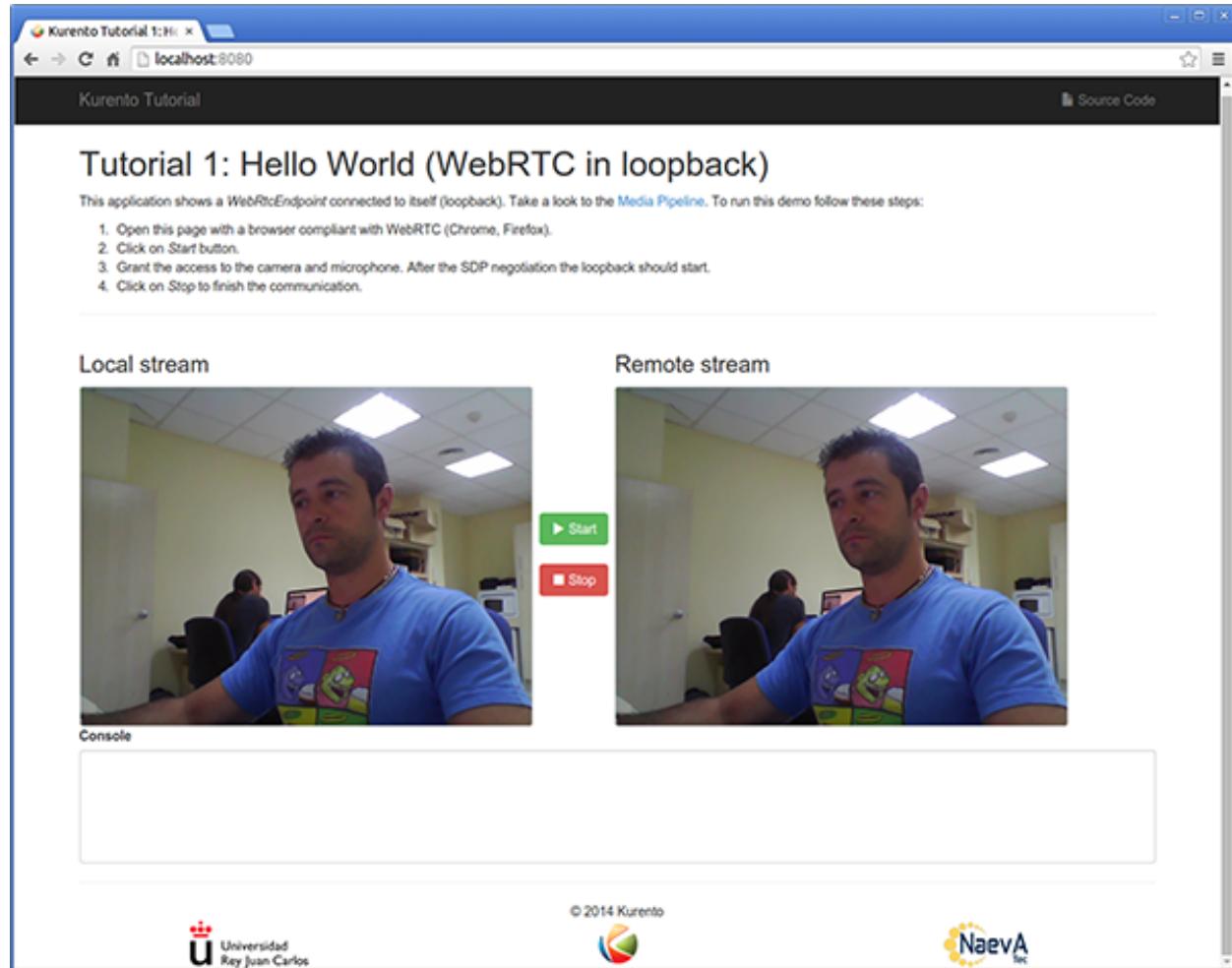
Fig. 1: *Kurento Hello World Screenshot: WebRTC in loopback*

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 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 application server consuming the **Kurento**
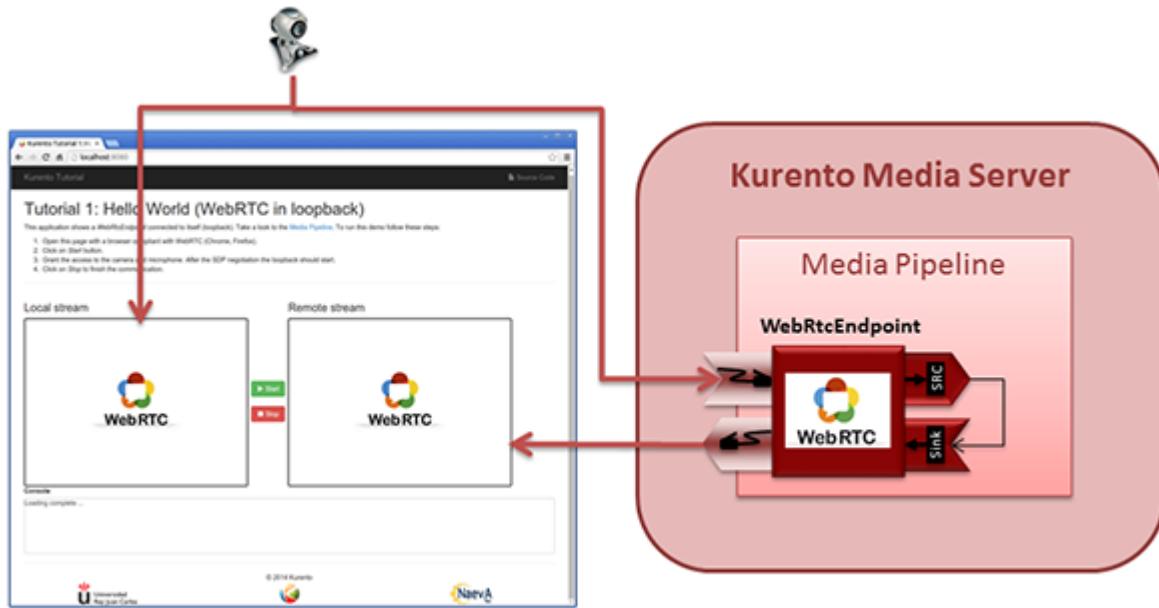
Fig. 2: *Kurento Hello World Media Pipeline in context*

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. 3: Complete sequence diagram of Kurento Hello World (WebRTC in loopback) demo
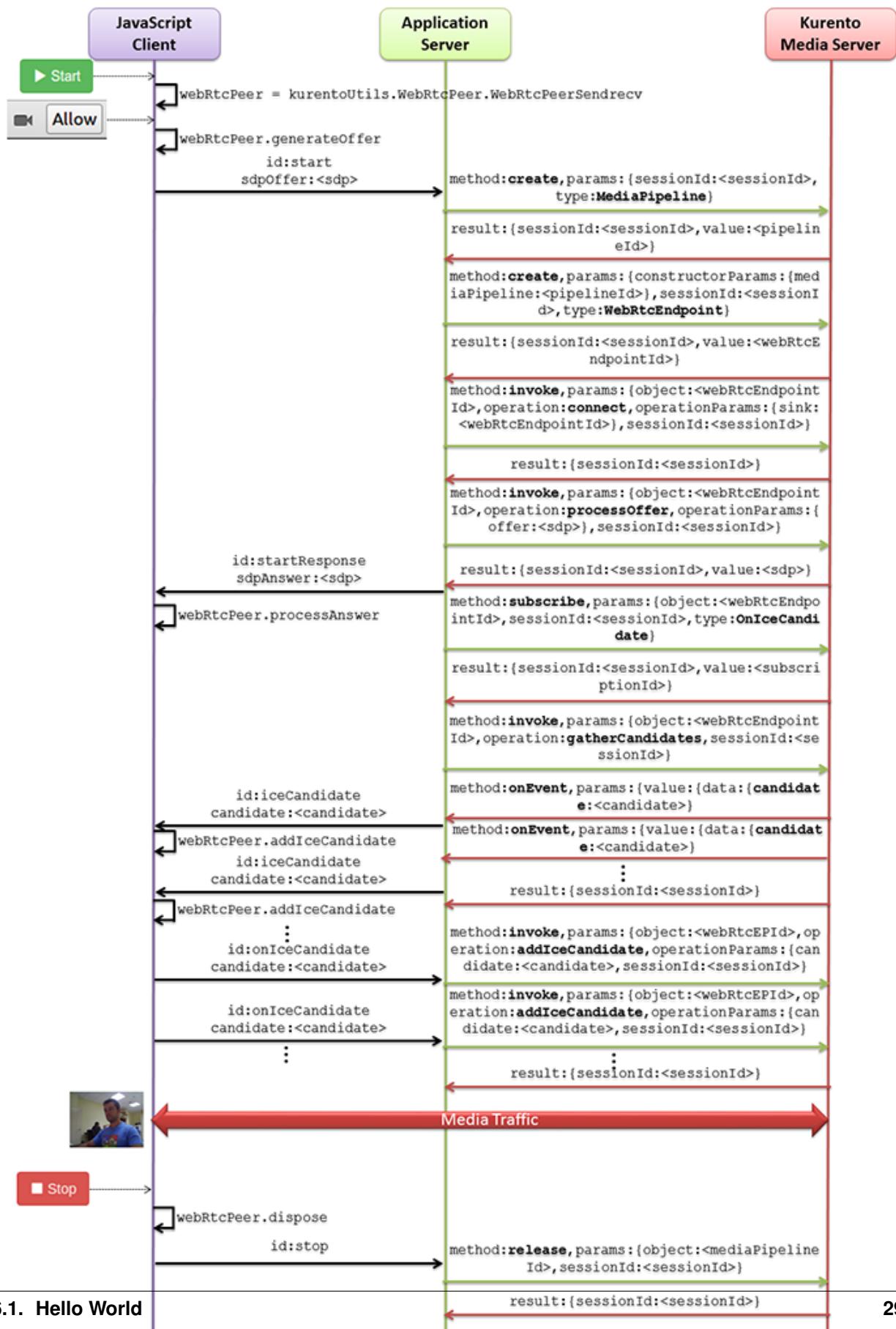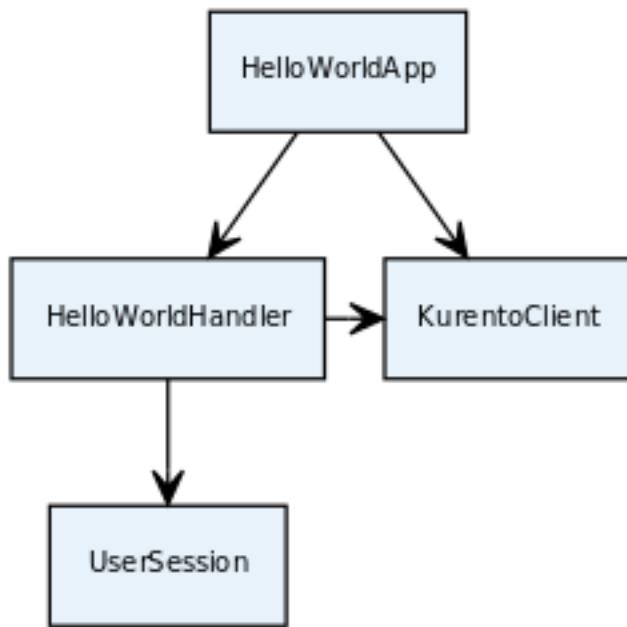
Fig. 4: *Server-side class diagram of the HelloWorld app*

`localhost`, listening in port TCP 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.

```java
@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.

```java
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.

```java
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*. Application Servers 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.

```java
// 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(
```

(continues on next page)

```java
    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.

  ```java
  // '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.

  ```java
  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.

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

ws.onmessage = function(message) {
   var parsedMessage = JSON.parse(message.data);
```

```javascript
   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);
```

```javascript
}

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('Sending 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 of 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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-hello-world/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

  ```
  https://localhost:8443/
  ```

- If KMS is running in a remote machine:

  ```
  https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
  ```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

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:

Fig. 5: *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 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 send 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. 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:
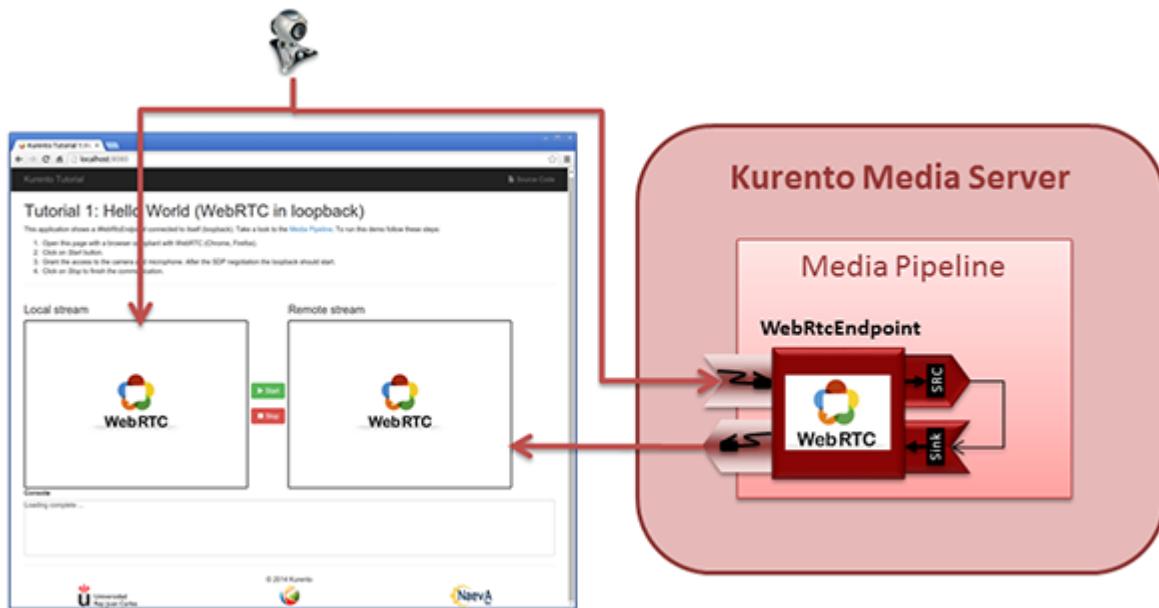
Fig. 6: *Kurento Hello World Media Pipeline in context*

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
  };
```

(continues on next page)

```
                                    JavaScript                    Kurento
                                      Client                   Media Server

        ▶ Start ---------------->
                                      webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv

        📹 Allow --------------->
                                      webRtcPeer.generateOffer

                                      method:create,params:{sessionId:<sessionId>,
                                                type:MediaPipeline}

                                      result:{sessionId:<sessionId>,value:<pipelin
                                                eId>}

                                      method:create,params:{constructorParams:{med
                                      iaPipeline:<pipelineId>},sessionId:<sessionI
                                                d>,type:WebRtcEndpoint}

                                      result:{sessionId:<sessionId>,value:<webRtcE
                                                ndpointId>}

                                      method:invoke,params:{object:<webRtcEndpoint
                                      Id>,operation:addIceCandidate,operationParam
                                                s:{candidate:{<candidate>}}
                                                ,sessionId:<sessionId>}

                                            result:{sessionId:<sessionId>}

                                      method:subscribe,params:{object:<webRtcEndpo
                                      intId>,sessionId:<sessionId>,type:OnIceCandi
                                                date}

                                      result:{sessionId:<sessionId>,value:<subscri
                                                ptionId>}

                                      method:invoke,params:{object:<webRtcEndpoint
                                      Id>,operation:processOffer,operationParams:{
                                            offer:<sdp>},sessionId:<sessionId>}

                                       result:{sessionId:<sessionId>,value:<sdp>}

                                      webRtcPeer.processAnswer

                                      webRtcPeer.gatherCandidates

                                      method:invoke,params:{object:<webRtcEndpoint
                                      Id>,operation:gatherCandidates,sessionId:<se
                                                ssionId>}

                                            result:{sessionId:<sessionId>}

                                      method:invoke,params:{object:<webRtcEndpoint
                                      Id>,operation:gatherCandidates,sessionId:<se
                                                ssionId>}

                                            result:{sessionId:<sessionId>}
                                                    ⋮

                                      webRtcPeer.connect

                                      method:invoke,params:{object:<webRtcEndpoint
                                      Id>,operation:connect,operationParams:{sink:
                                       <webRtcEndpointId>},sessionId:<sessionId>}

                                            result:{sessionId:<sessionId>}

                                      method:invoke,params:{object:<webRtcEndpoint
                                      Id>,operation:connect,operationParams:{sink:
                                       <webRtcEndpointId>},sessionId:<sessionId>}
```

```
    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 TCP 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": "6.15.0",
   "kurento-utils": "6.15.0"
}
```

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 of 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:** Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.
>
> For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 install *Bower* and *Node.js* **version 8.x** in your system. In an Ubuntu machine, you can install both as follows:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

Also, Node.js should already include NPM, the Node.js package manager.

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 6.15.0
npm install
cd static
bower install --allow-root
cd ..
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}:8888/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. 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:

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

Fig. 9: *Kurento Hello World Media Pipeline in context*

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'
});

/*
 * Management of WebSocket messages
 */
```

Fig. 10: *Complete sequence diagram of Kurento Hello World (WebRTC in loopback) demo*

```javascript
wss.on('connection', function(ws, req) {
    var sessionId = null;
    var request = req;
    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 TCP 8888.

```javascript
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:

```javascript
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) {
```

```javascript
                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.

```javascript
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.

```javascript
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
    }
```

```javascript
    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:

```json
"dependencies": {
    [...]
    "kurento-client" : "6.15.0"
}
```

At the client side, dependencies are managed using *Bower*. Take a look to the bower.json file and pay attention to the following section:

```json
"dependencies": {
    [...]
    "kurento-utils" : "6.15.0"
}
```

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 of 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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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 server-side, we use a Spring-Boot based application server 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

---
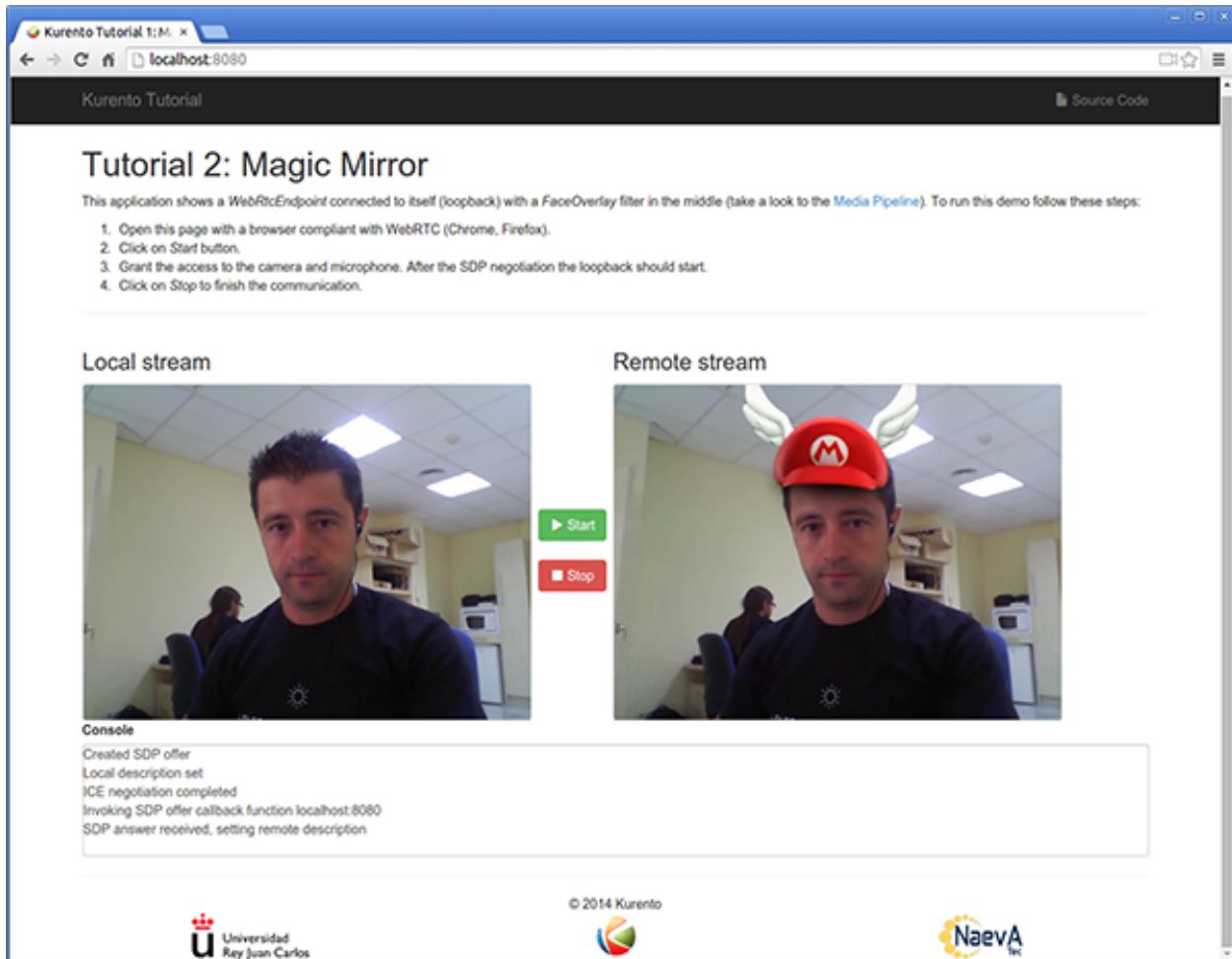
Fig. 11: *Kurento Magic Mirror Screenshot: WebRTC with filter in loopback*
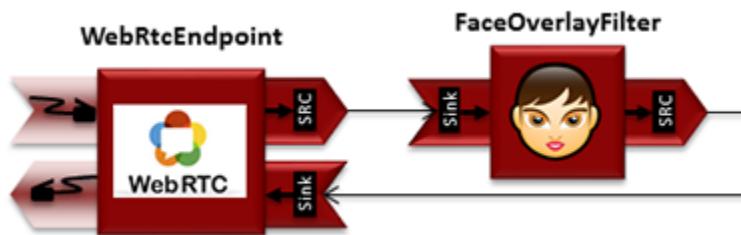
Fig. 12: *WebRTC with filter in loopback Media Pipeline*

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:
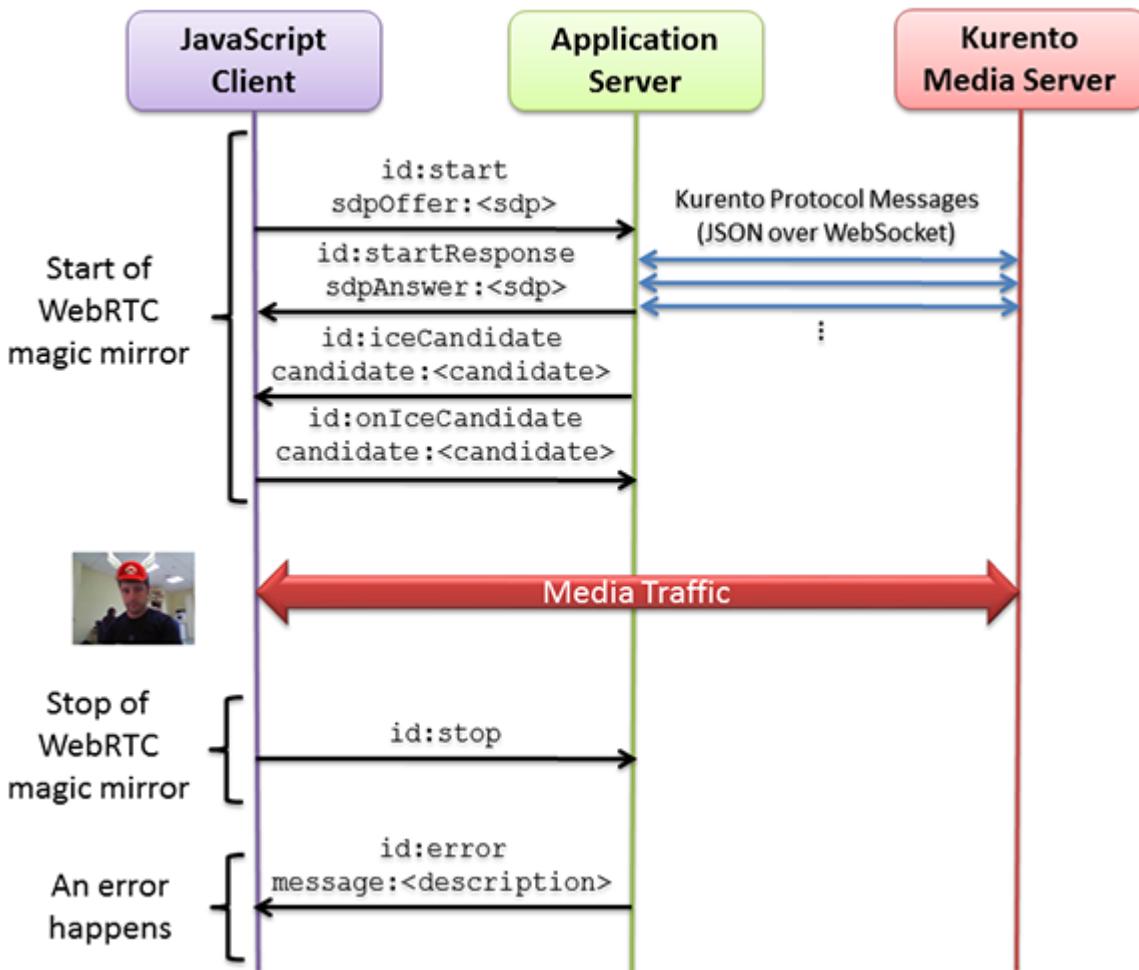
Fig. 13: *One to one video call signaling protocol*

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:
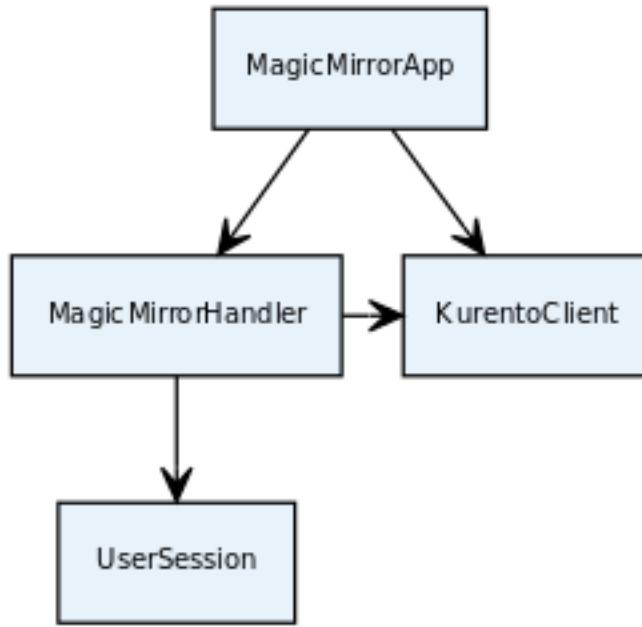
Fig. 14: *Server-side class diagram of the MagicMirror app*

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 TCP 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");
```

(continues on next page)

```
   }

   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());
```

```java
                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.

```java
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();
```

```java
        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 -U clean spring-boot:run -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.

```java
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 We-

bRTC 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 Web-Socket 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.

```javascript
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) {
```

```javascript
          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:

```xml
<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>
```

```xml
            <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:

```xml
<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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-magic-mirror/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

```
https://localhost:8443/
```

- If KMS is running in a remote machine:

```
https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

Note: By default, this tutorial assumes that Kurento Media Server can download the overlay image from a `localhost` web server. *It will fail if the web server is remote* (from the point of view of KMS). This includes the case of running KMS from Docker.

If you want to run this tutorial with a **remote Kurento Media Server** (including running KMS from Docker), then you have to provide it with the correct IP address of the application's web server:

- In *index.js*, change `hat_uri` to the correct one where KMS can reach the web server.

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:

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:

The complete source code of this demo can be found in GitHub.
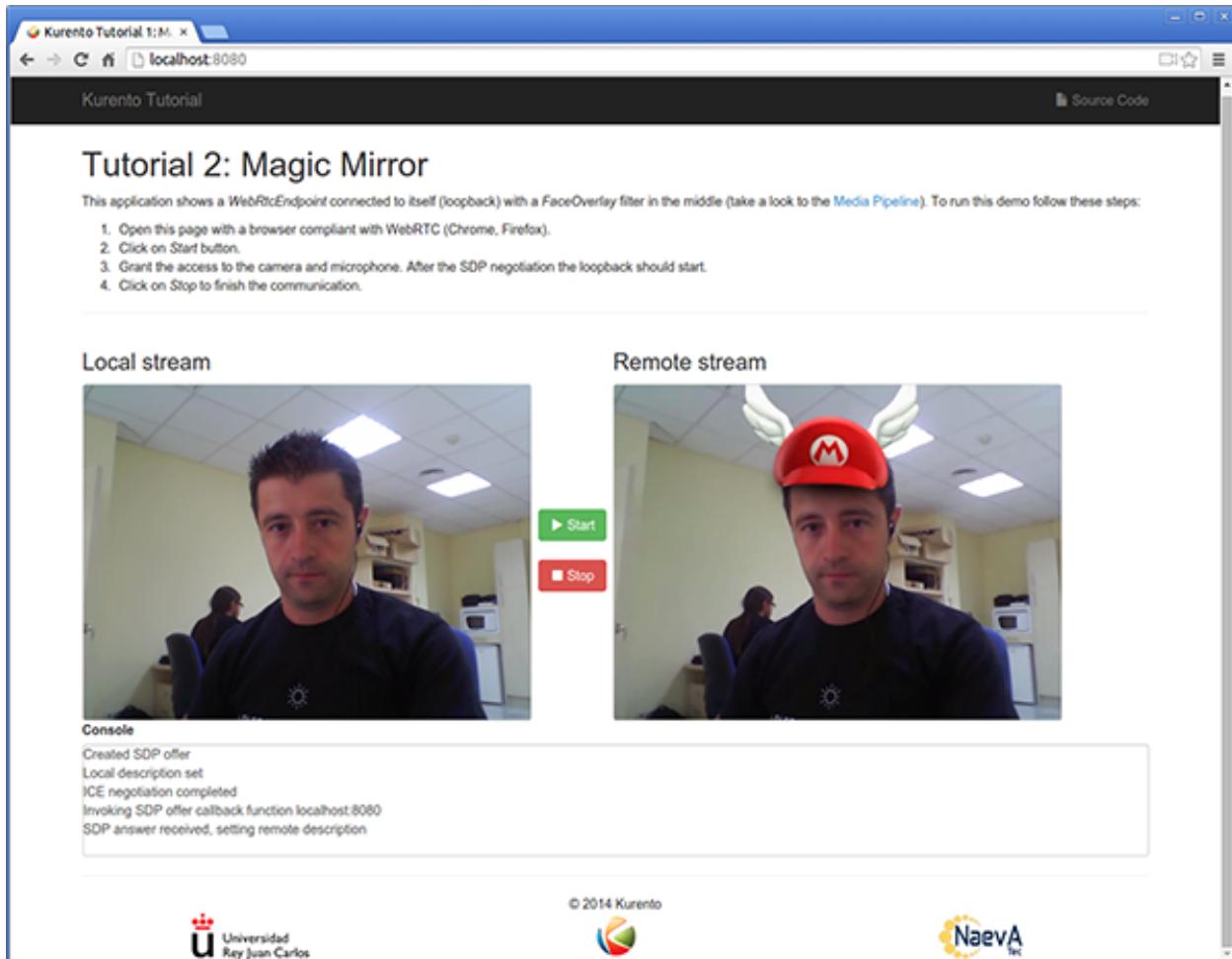
Fig. 15: *Kurento Magic Mirror Screenshot: WebRTC with filter in loopback*
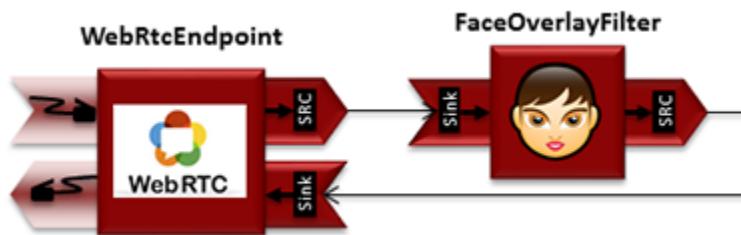
Fig. 16: *WebRTC with filter in loopback Media Pipeline*

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.

```javascript
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 TCP 8888 at the same host than the HTTP serving the application.

```javascript
[...]

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

<div align="right">(continues on next page)</div>

```
[...]

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.setOverlayedImage(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": "6.15.0",
    "kurento-utils": "6.15.0"
}
```

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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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:


Fig. 17: *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 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).

Fig. 18: *WebRTC with filter in loopback 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 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.

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:

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'
```

<div align="right">(continues on next page)</div>

Fig. 19: *One to one video call signaling protocol*

```javascript
});

/*
 * Management of WebSocket messages
 */
wss.on('connection', function(ws, req) {
    var sessionId = null;
    var request = req;
    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 TCP 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) {
```

```javascript
        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.
→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, 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.

```javascript
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.

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

[...]

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

```javascript
    // 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:

```json
"dependencies": {
   [...]
   "kurento-client" : "6.15.0"
}
```

At the client side, dependencies are managed using *Bower*. Take a look to the bower.json file and pay attention to the following section:

```json
"dependencies": {
   [...]
```

```
    "kurento-utils" : "6.15.0"
}
```

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 Application Server **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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Quick start

Follow these steps to run this demo application:

1. Install and run Kurento Media Server: *Installation Guide*.

2. Install Java JDK and Maven:

```
sudo apt-get update
sudo apt-get install default-jdk maven
```

3. Run these commands:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-rtp-receiver/
git checkout 6.15.0
mvn -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/kurento"
```

4. Open the demo page with a WebRTC-compliant browser (Chrome, Firefox): https://localhost:8443/

5. Click on *Start* to begin the demo.

6. Copy the KMS **IP** and **Port** values to the external streaming program.

7. 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.

8. 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 latter is used to show the 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 application server 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. A WebSocket is created between the Application Server and the browser client, to implement a custom signaling protocol.

2. Another WebSocket is used to perform the communication between the Application Server and the Kurento Media Server. For this, the Application Server uses the Kurento Java Client library. This communication takes place using the **Kurento Protocol** (see *Kurento Protocol*).

The complete source code for this tutorial 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 Application Server:

Fig. 20: Server-side class diagram of the Application Server

Client-Side Logic

We use a specific Kurento JavaScript library called **kurento-utils.js** to simplify the WebRTC interaction between browser and application server. 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 of a one-to-many video call using *WebRTC* technology. In other words, it is an implementation of a video broadcasting web application.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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:

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 a Spring-Boot based application server 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*.
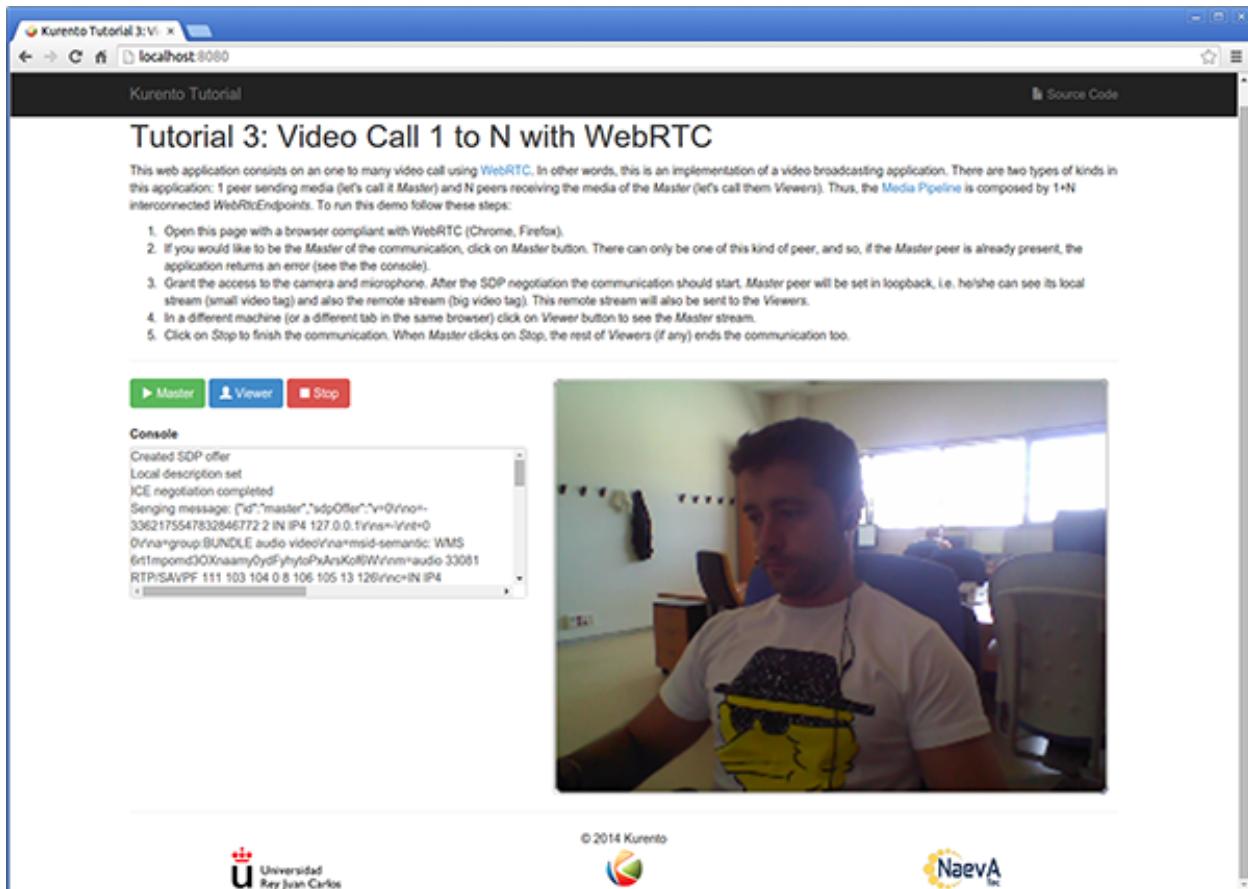
Fig. 21: *One to many video call screenshot*
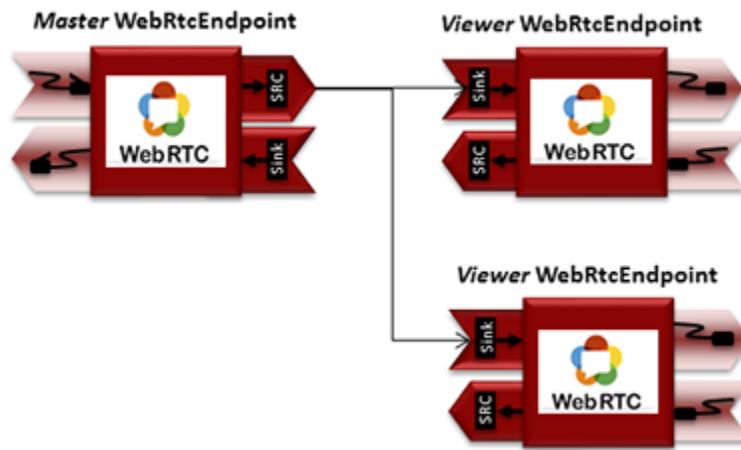
Fig. 22: *One to many video call Media Pipeline*

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 example, a pure Java EE application, SIP Servlets, Play, Vert.x, etc. 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 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 TCP 8888.

```java
@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);
    }

}
```
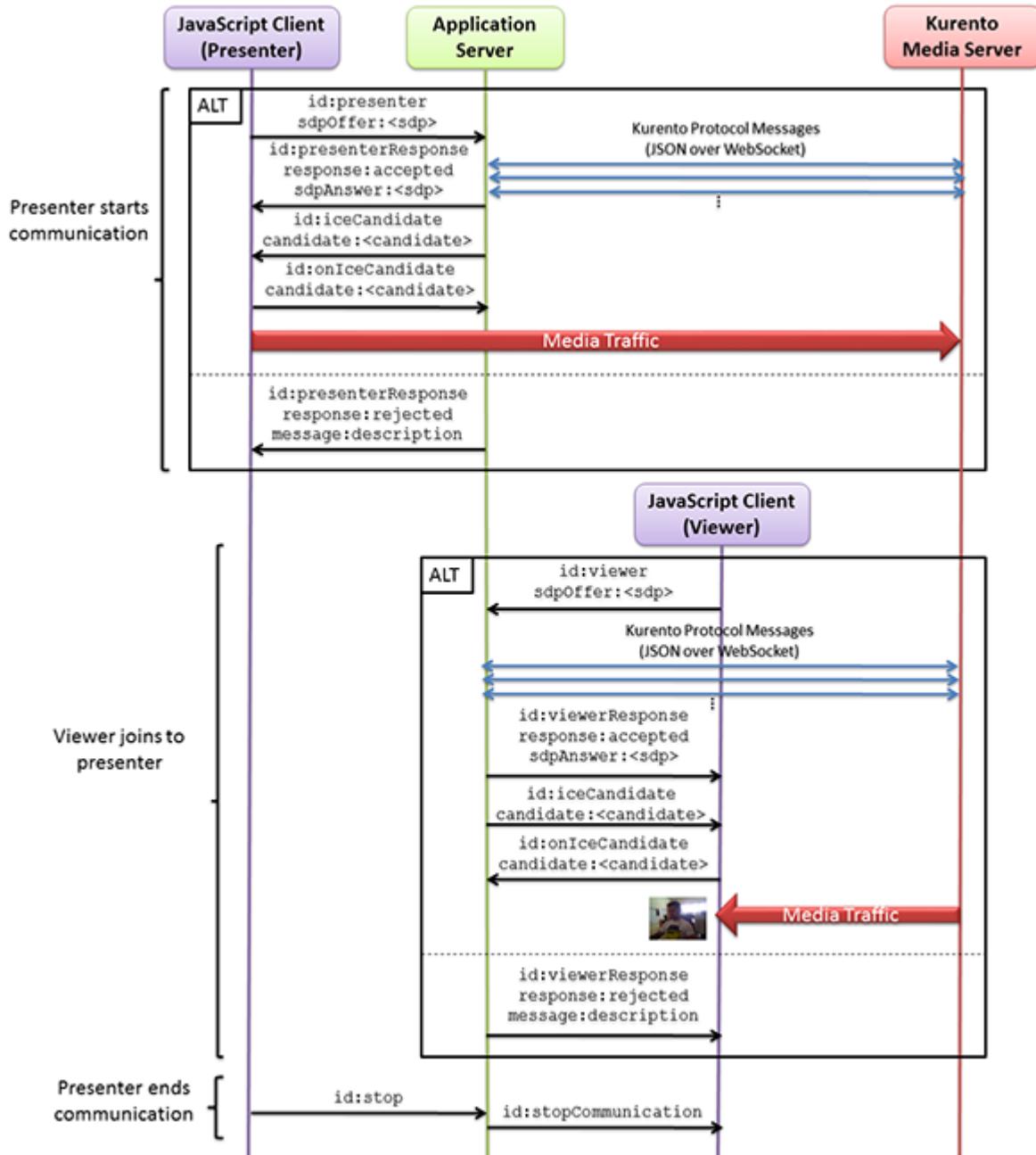
Fig. 23: *One to many video call signaling protocol*
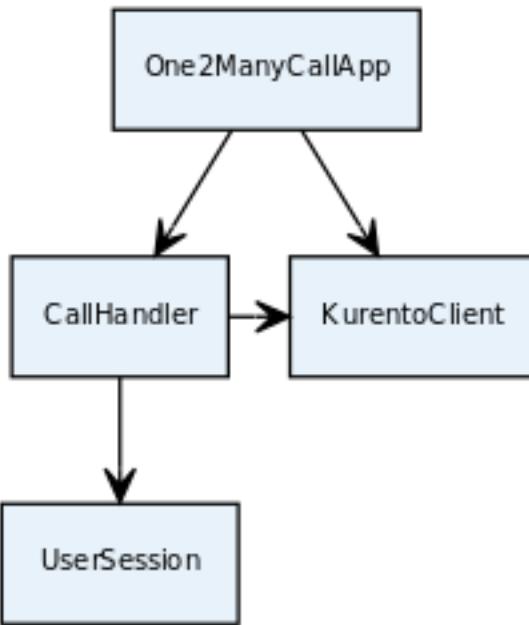
Fig. 24: *Server-side class diagram of the One2Many app*

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.

```java
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);
```

(continues on next page)

```java
      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 {
```

```java
      ...
   }

   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`:

```java
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();
```

```java
    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 not he *Presenter* WebRtcEndpoint is connected to each of the viewers WebRtcEndpoints, otherwise an error is sent back to the client.

```java
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();
```

```java
            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*:

```java
private synchronized void stop(WebSocketSession session) throws IOException {
  String sessionId = session.getId();
  if (presenterUserSession != null && presenterUserSession.getSession().getId().
→equals(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.

```javascript
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) {
      showSpinner(video);

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

(continues on next page)

```
      });
   }
}

function viewer() {
   if (!webRtcPeer) {
      showSpinner(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:

```xml
<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>
```

```xml
        <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:

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

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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*.

   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.

Fig. 25: *One to many video call screenshot*
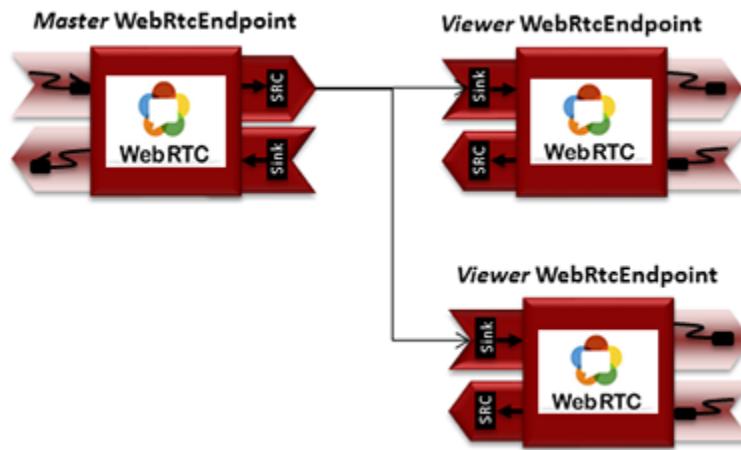
Fig. 26: *One to many video call Media Pipeline*
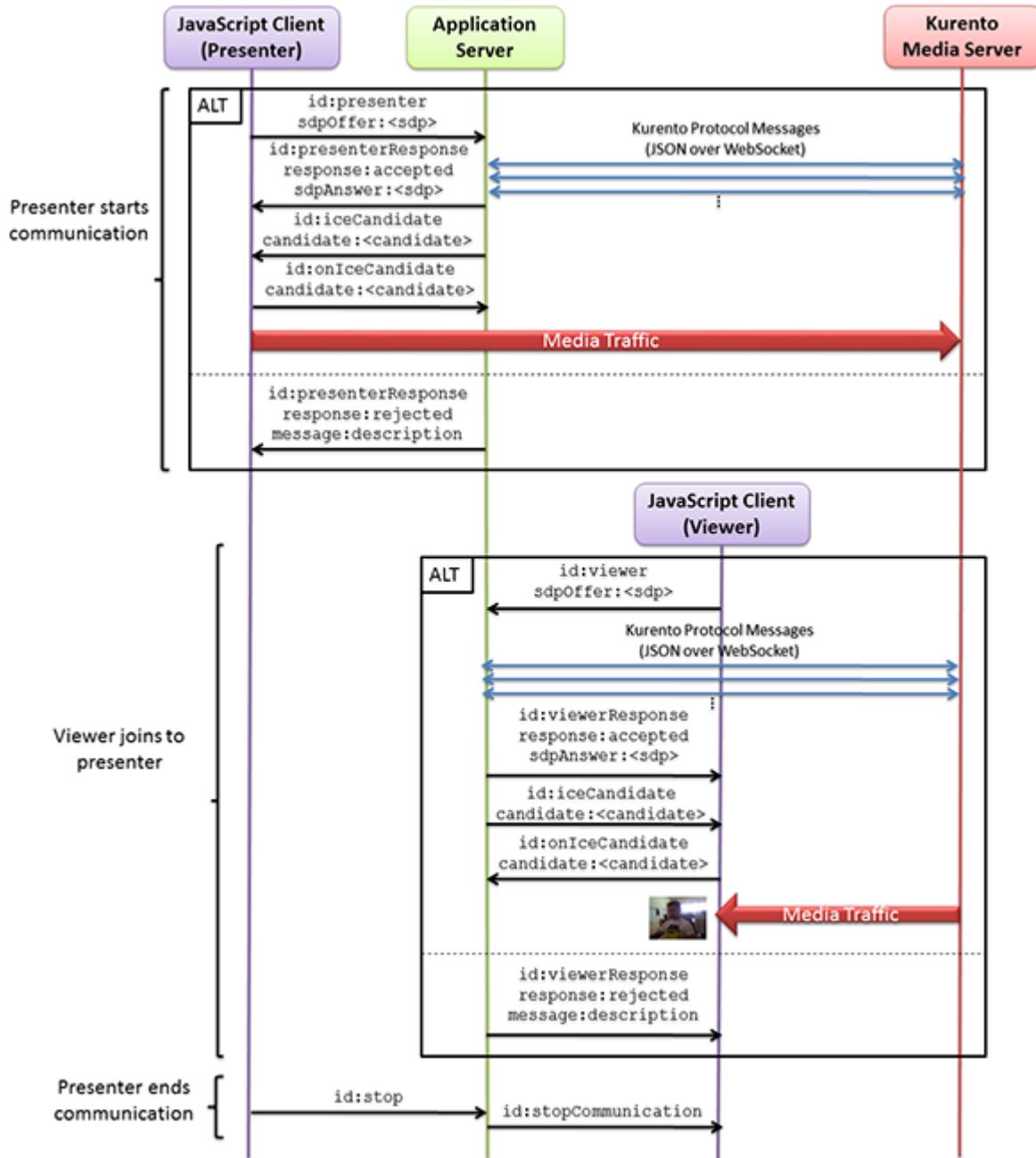
Fig. 27: *One to many video call signaling protocol*

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`).

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

```javascript
        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 TCP 8888.

```javascript
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
↪...");
    }

    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);
            }
```

```javascript
        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.

```javascript
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 : 'Unknow 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 : 'Unknow 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.

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

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

      webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendonly(options,
↪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,
↪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" : "6.15.0"
}
```

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" : "6.15.0"
}
```

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 of a one-to-one video call using *WebRTC* technology. In other words, this application provides a simple video softphone.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/kurento"
```

Understanding this example

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 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 sa *Media Pipeline* composed by two WebRtC endpoints connected in B2B. The implemented media pipeline is illustrated in the following picture:

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.

Fig. 28: *One to one video call screenshot*

Fig. 29: *One to one video call Media Pipeline*
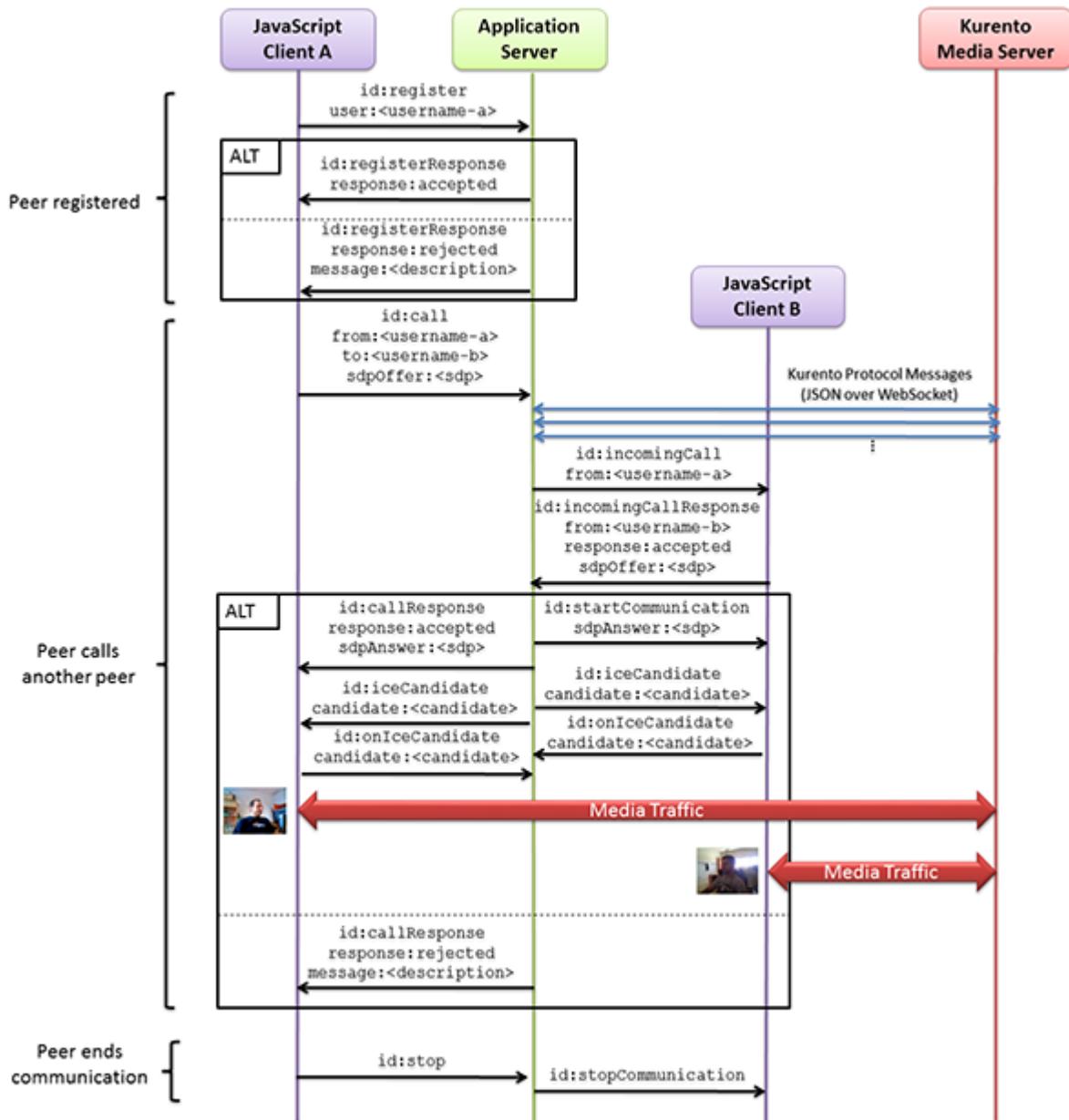
Fig. 30: *One to many one call signaling protocol*

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. We have choose Spring Boot for convenience.

In the following figure you can see a class diagram of the server side code:
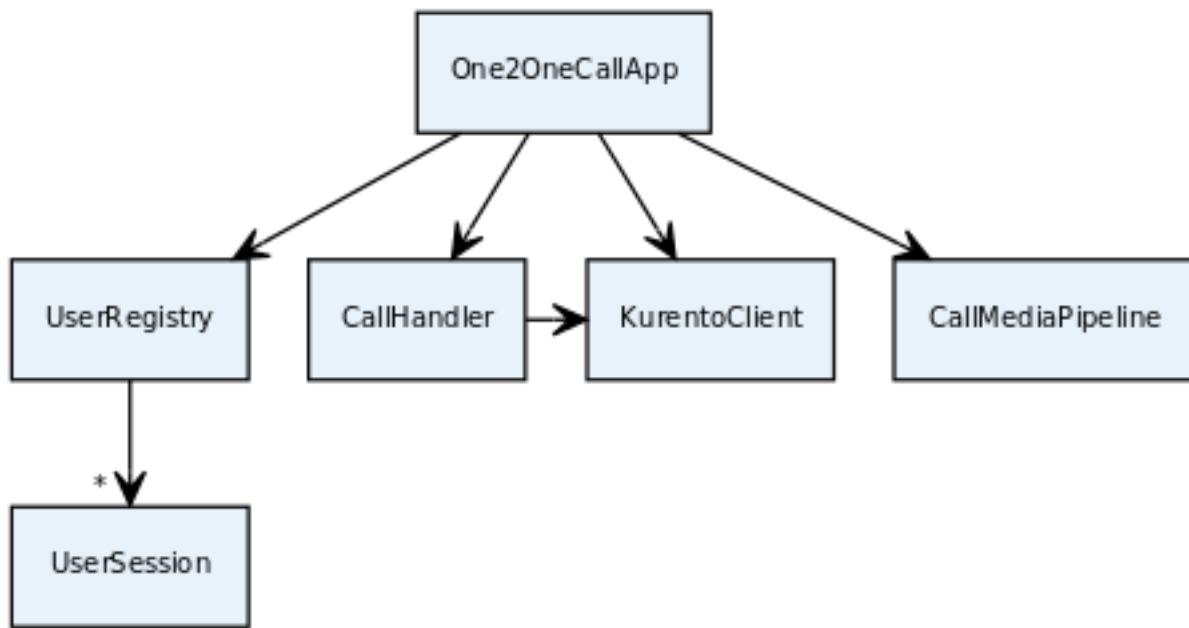
Fig. 31: *Server-side class diagram of the one to one video call app*

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 {

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

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

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

```java
    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.

```java
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");
```

```java
        }
        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", "resgisterResponse");
  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 {
```

```java
      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:

```java
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)`:

```java
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);
```

```java
    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.debug(e.getMessage());
                    }
                }
            });

            String calleeSdpAnswer = pipeline.generateSdpAnswerForCallee(calleeSdpOffer);
            String callerSdpOffer = registry.getByName(from).getSdpOffer();
            calleer.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 (calleer.getSession()) {
                            calleer.getSession().sendMessage(new TextMessage(response.
→toString()));
                        }
                    } catch (IOException e) {
                        log.debug(e.getMessage());
                    }
                }
            });

            String callerSdpAnswer = pipeline.generateSdpAnswerForCaller(callerSdpOffer);
```

```java
            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 (calleer) {
                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.

```java
public class CallMediaPipeline {

    private MediaPipeline pipeline;
```

```java
   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();
      }
   }

   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: `resgisterResponse`, `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.

```javascript
 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 '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(PROCESSING_CALL);
   if (confirm('User ' + message.from
         + ' is calling you. Do you accept the call?')) {
      showSpinner(videoInput, videoOutput);
```

(continues on next page)

```javascript
        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:

```xml
<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:

```xml
<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 of a one-to-one video call using *WebRTC* technology. In other words, this application provides a simple video softphone.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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:

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:
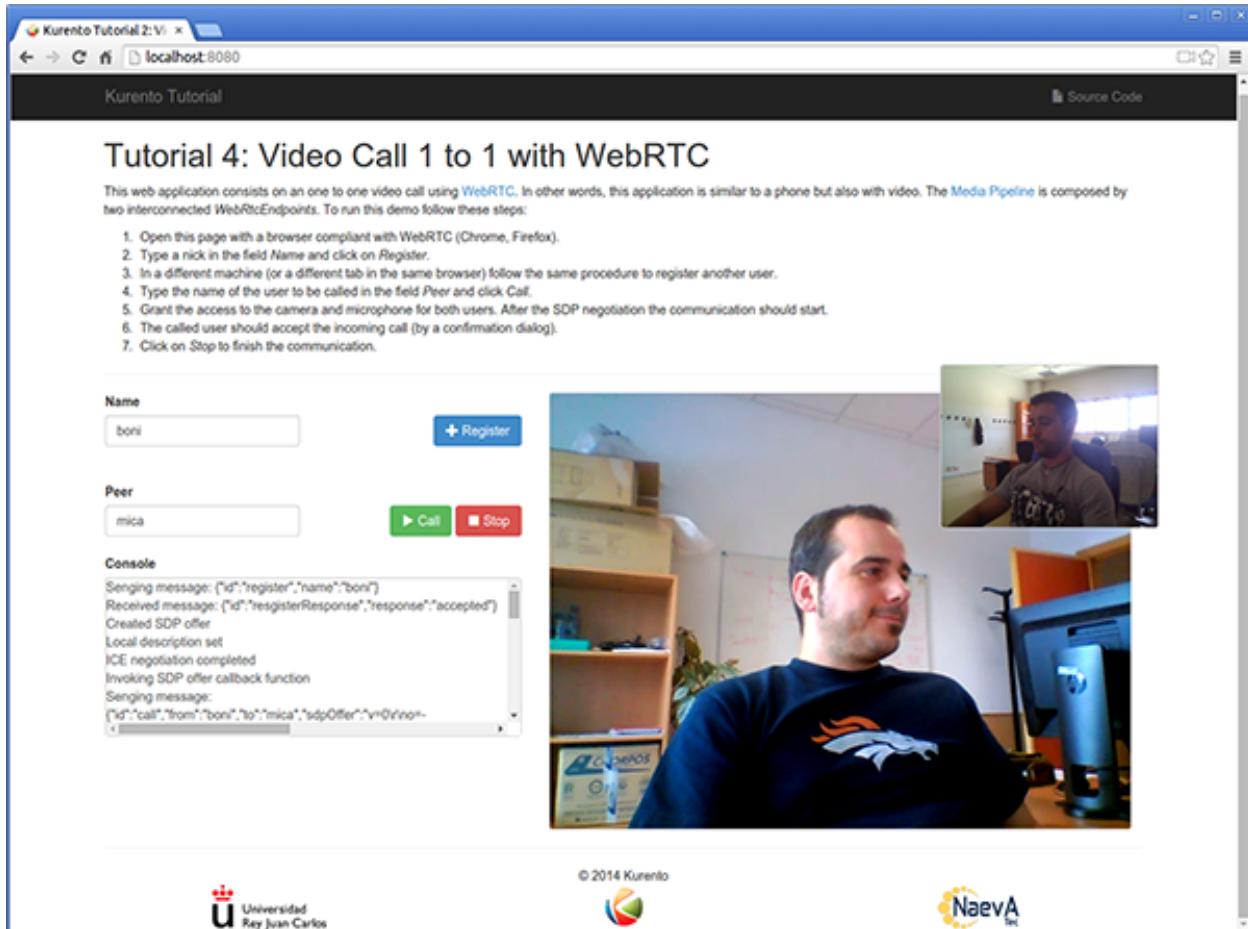

Fig. 32: *One to one video call screenshot*

Fig. 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`).

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

[...]

var wss = new ws.Server({
    server : server,
    path : '/one2one'
});

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);
        userRegistry.unregister(sessionId);
    });

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

(continues on next page)
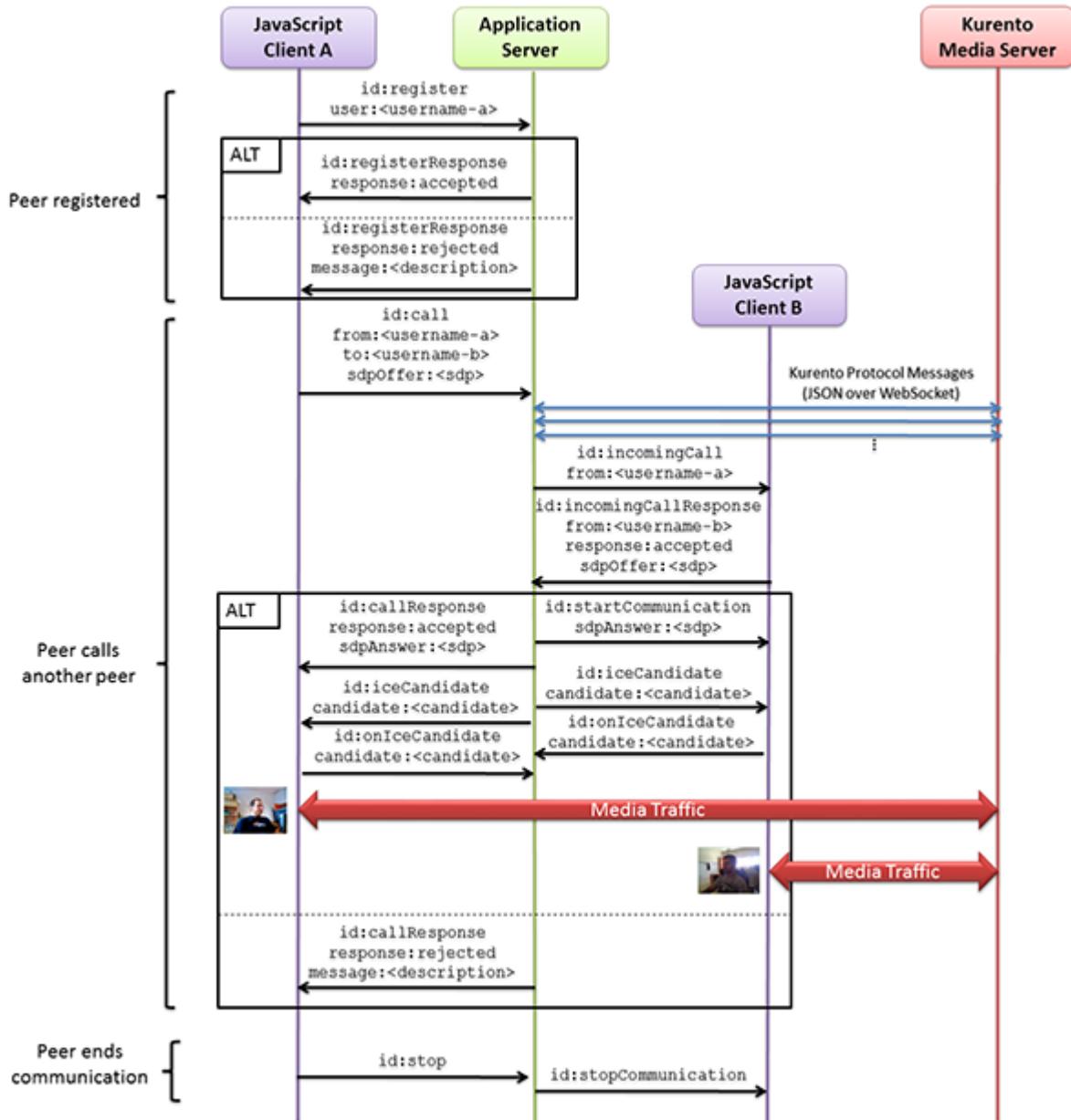
Fig. 34: *One to many one call signaling protocol*

```javascript
        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:

```javascript
// 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];
```

```javascript
}

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 in port TCP 8888.

```javascript
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 `/one2one`. 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 + '/one2one');
var webRtcPeer;

[...]

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

   switch (parsedMessage.id) {
   case 'registerResponse':
      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 '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) {
```

```javascript
        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(PROCESSING_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);
                    }
```

```javascript
                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:

```json
"dependencies": {
    [...]
    "kurento-client" : "6.15.0"
}
```

At the client side, dependencies are managed using *Bower*. Take a look to the bower.json file and pay attention to the following section:

```json
"dependencies": {
    [...]
    "kurento-utils" : "6.15.0"
}
```

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 of an advanced one to one video call using *WebRTC* technology. It is an improved version of the *one 2 one call tutorial*).

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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:
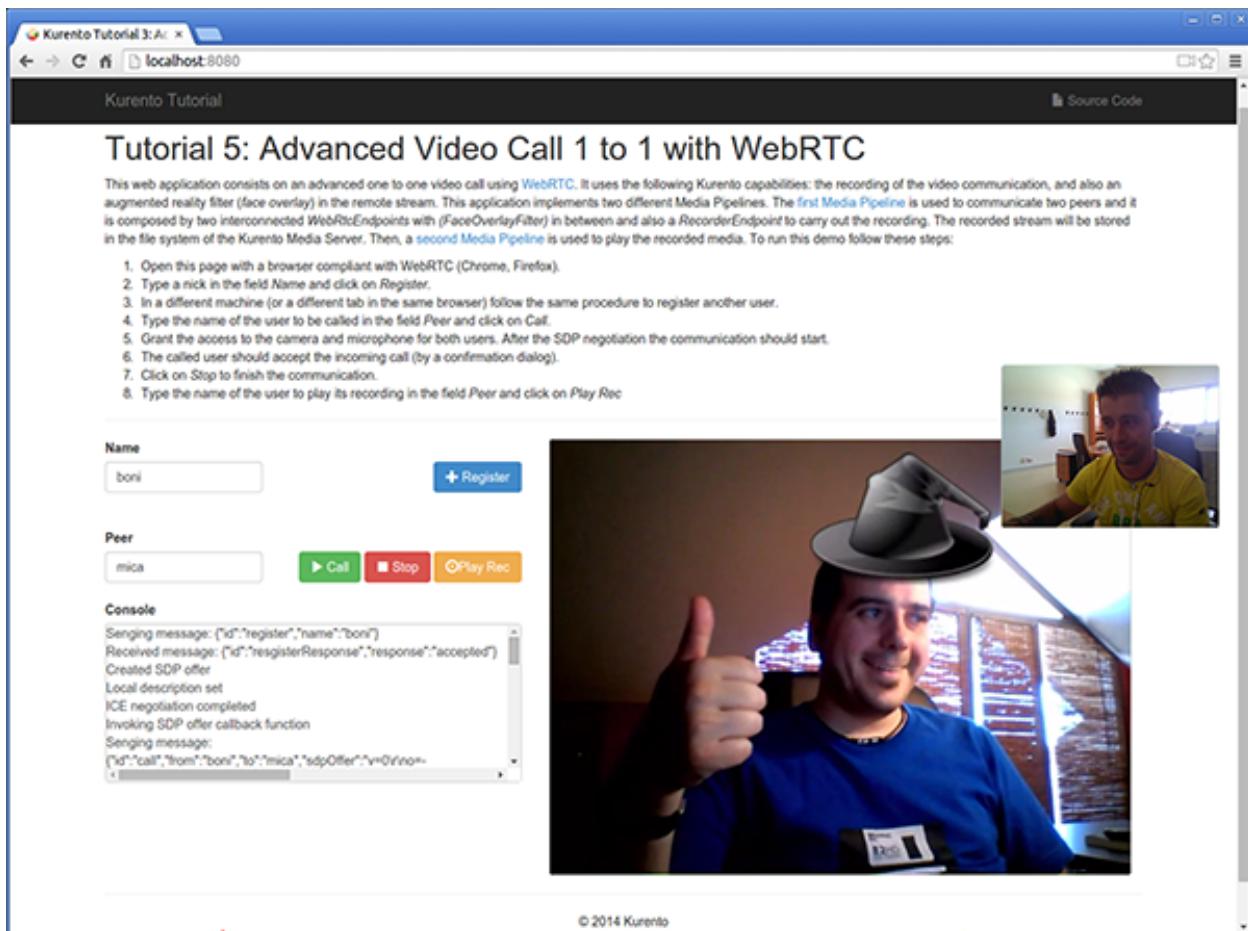
Fig. 35: *Advanced one to one video call screenshot*
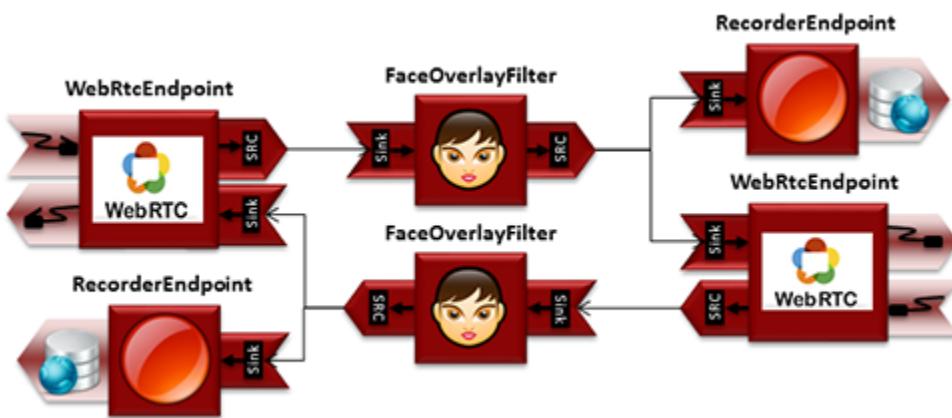
Fig. 36: *Advanced one to one video call media pipeline (1)*
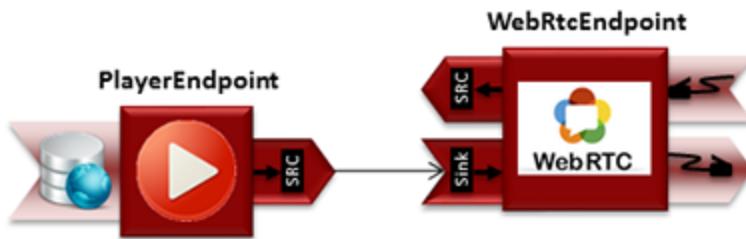
Fig. 37: *Advanced one to one video call media pipeline (2)*

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, Vert.x, etc. We have choose Spring Boot for convenience.

In the following figure you can see a class diagram of the server side code:
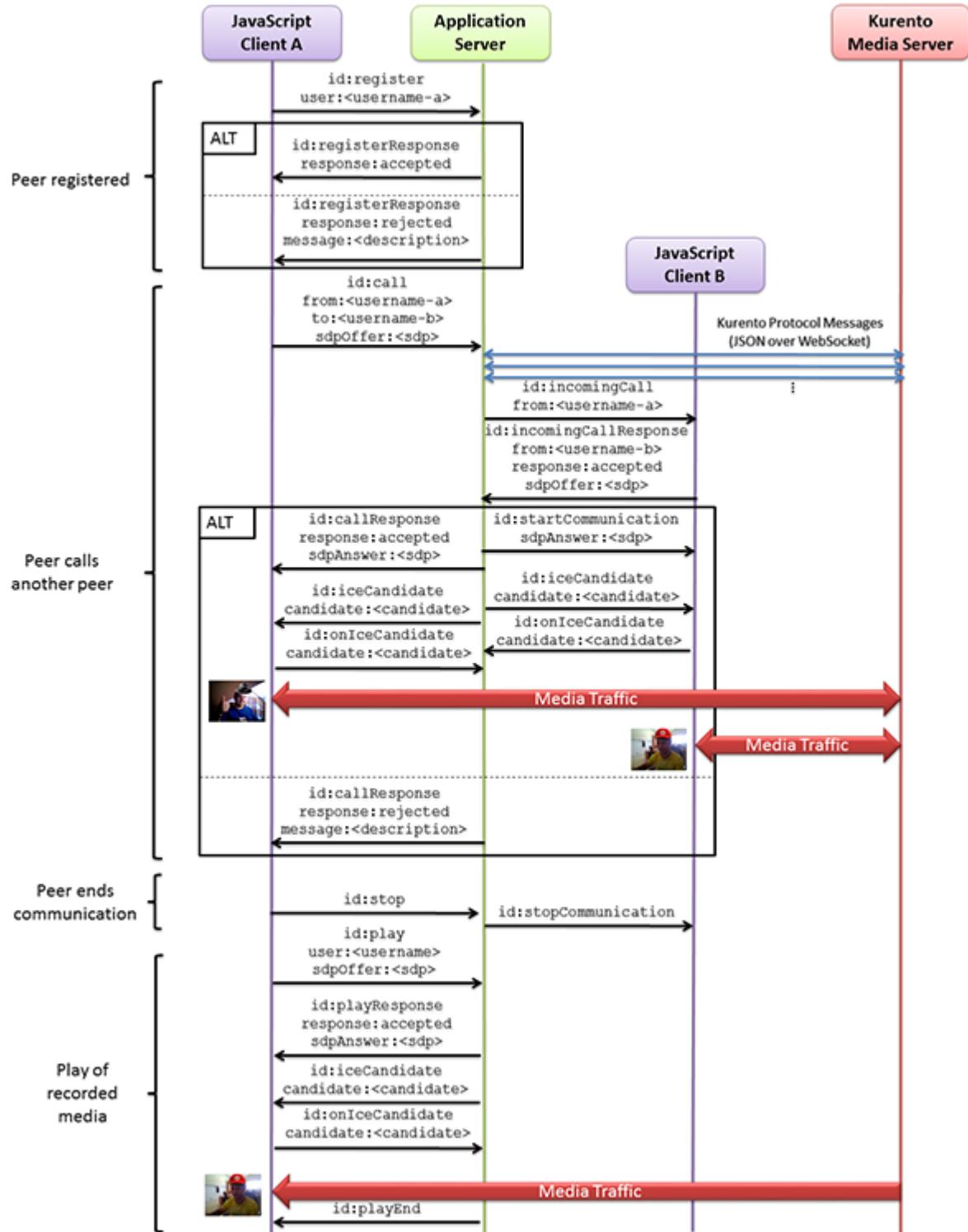
Fig. 38: *Advanced one to one video call signaling protocol*
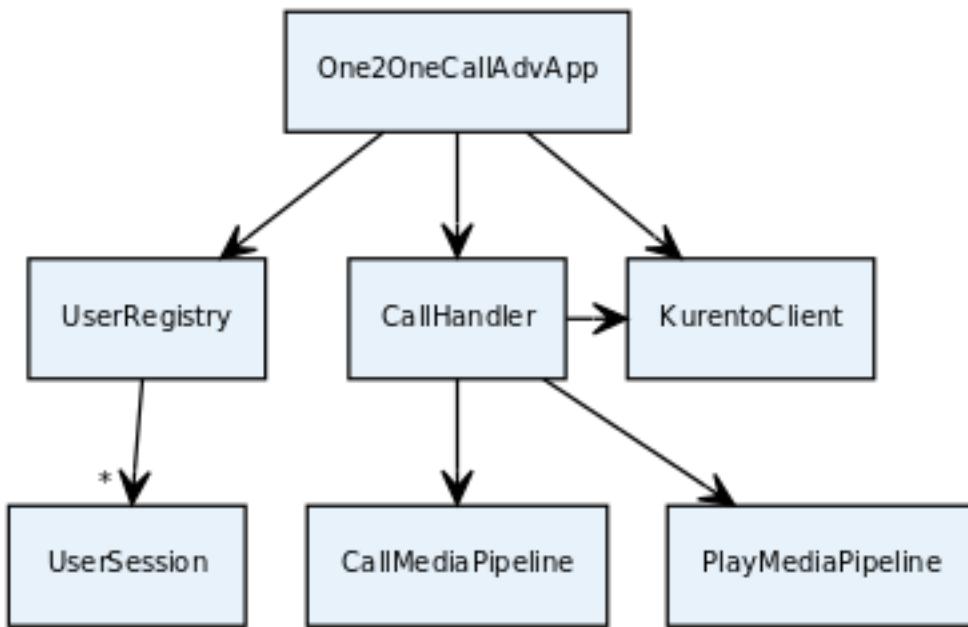
Fig. 39: *Server-side class diagram of the advanced one to one video call app*

The main class of this demo is named One2OneCallAdvApp. As you can see, the KurentoClient is instantiated in this class as a Spring Bean.

```java
@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.

```java
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;
```

```java
        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);
            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.

```java
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", "resgisterResponse");
    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.

```java
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)`:

```java
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);
```

```java
        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.

```java
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 of 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();
```

```java
        faceOverlayFilterCaller.setOverlayedImage(appServerUrl
                + "/img/mario-wings.png", -0.35F, -1.2F, 1.6F, 1.6F);

        FaceOverlayFilter faceOverlayFilterCallee = new FaceOverlayFilter.Builder(
                pipeline).build();
        faceOverlayFilterCallee.setOverlayedImage(
                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 -U clean spring-boot:run -Dapp.server.url=https://app_server_host:app_server_port
```

The second media pipeline consists of 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.

```java
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;
  }
```

```java
    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.

```javascript
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':
```

```javascript
          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;
```

```javascript
    }
    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␣
↪(field 'Peer')");
        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:

```xml
<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 wih the concept of rooms, allowing to connect several clients between them using *WebRTC* technology, creating a multiconference.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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*n 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 application server 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 GroupCalldApp. 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 TCP 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.

```java
@EnableWebSocket
@SpringBootApplication
public class GroupCallApp implements WebSocketConfigurer {

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

  @Bean
  public RoomManager roomManager() {
    return new RoomManager();
  }

  @Bean
```

(continues on next page)

```java
  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.

```java
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);
```

```java
    }

    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.

```java
@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.

```java
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.

```java
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.

```javascript
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;
```

(continues on next page)

```javascript
   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);
```

```javascript
      });
   }
}

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('Sending 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:

```xml
<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>
```

```xml
        <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:

```xml
<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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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 application server 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 TCP 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.

```java
@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.

```java
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<>
  ↪();

  @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);
```

(continues on next page)

```java
    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.

```java
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>() {
```

```java
      @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.

```java
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.

```javascript
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,
```

(continues on next page)

```javascript
      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('Sending 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:

```xml
<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>
```

6.9 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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/kurento"
```

Understanding this example

To implement this behavior we have to create a *Media Pipeline* composed by one **PlayerEndpoint** and one **WebRtcEndpoint**. The **PlayerEnpdoint** 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 application server 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 TCP 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.

```java
@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");
  }
```

(continues on next page)

```java
  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.

```java
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;
```

```java
        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.

```java
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();
```

```java
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(session.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.

```java
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.

```java
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.

```java
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.

```javascript
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␣
↪offer ...');

   // Video and audio by default
```

(continues on next page)

```javascript
    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 ...');
```

```javascript
    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'
    }
```

```javascript
    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('Sending 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:

```xml
<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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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 **PlayerEnpdoint** 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 application server 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 TCP 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.

```java
@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 {
    new SpringApplication(SendDataChannelApp.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 `/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.

```java
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);
```

(continues on next page)

```java
    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.

```java
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.openvidu.io/video/filter/barcodes.webm").build();
```

```java
      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.

```java
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.

```javascript
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);
}
```

```javascript
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('Sending 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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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 application server 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 TCP 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.

```java
@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.

```java
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").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) {
    ...
  }
}
```

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.

```java
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.

```java
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.

```javascript
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);
```

```javascript
      }
      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) {
```

```javascript
            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();
```

```javascript
      webRtcPeer.dispose();
      webRtcPeer = null;

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

function sendMessage(message) {
   var jsonMessage = JSON.stringify(message);
   console.log('Sending 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:

```xml
<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>
```

```xml
        <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:

```xml
<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.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-hello-world-data-channel/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

```
https://localhost:8443/
```

- If KMS is running in a remote machine:

```
https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

Note: This demo uses the **kms-datachannelexample** module, which must be installed in the media server. That module is available in the Kurento Apt repositories, so it is possible to install it with this command:

```
sudo apt-get update && sudo apt-get install --yes 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.

```javascript
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 TCP 8433 at the same host than the HTTP serving the application.

```javascript
[...]

var args = getopts(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:

```javascript
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": "6.15.0",
   "kurento-utils": "6.15.0"
}
```

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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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 **RecorderEnpoint**.

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 application server 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 TCP 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.

```java
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;
```

```java
      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.

```java
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);
```

```java
      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.

```java
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);
```

```java
    // 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 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 `/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':
```

```javascript
      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
   };
```

```javascript
      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 = {
```

```javascript
        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('Sending 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:

```xml
<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.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-recorder/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

  ```
  https://localhost:8443/
  ```

- If KMS is running in a remote machine:

  ```
  https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
  ```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

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 **RecorderEnpoint**.

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.

```javascript
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 TCP 8433 at the same host than the HTTP serving the application.

```javascript
[...]

var args = getopts(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": "6.15.0",
   "kurento-utils": "6.15.0"
}
```

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: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="\
        -Dkms.url=ws://{KMS_HOST}:8888/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 also be 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 application server 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 TCP 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();
```

(continues on next page)

```
  }

  @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;
```

```java
@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);
```

```java
      }
    } 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
```

```javascript
   };
   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('Sending 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:

```xml
<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>
```

```xml
    </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:

```xml
<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.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-loopback-stats/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

```
https://localhost:8443/
```

- If KMS is running in a remote machine:

```
https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

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.

```javascript
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 TCP 8433 at the same host than the HTTP serving the application.

```javascript
[...]

var args = getopts(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:

```javascript
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();
      }
    });
  });
});
```

Note: The *TURN* and *STUN* servers to be used can be configured simply 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": "6.15.0",
   "kurento-utils": "6.15.0"
}
```

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

CHAPTER 7

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 theses 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*.
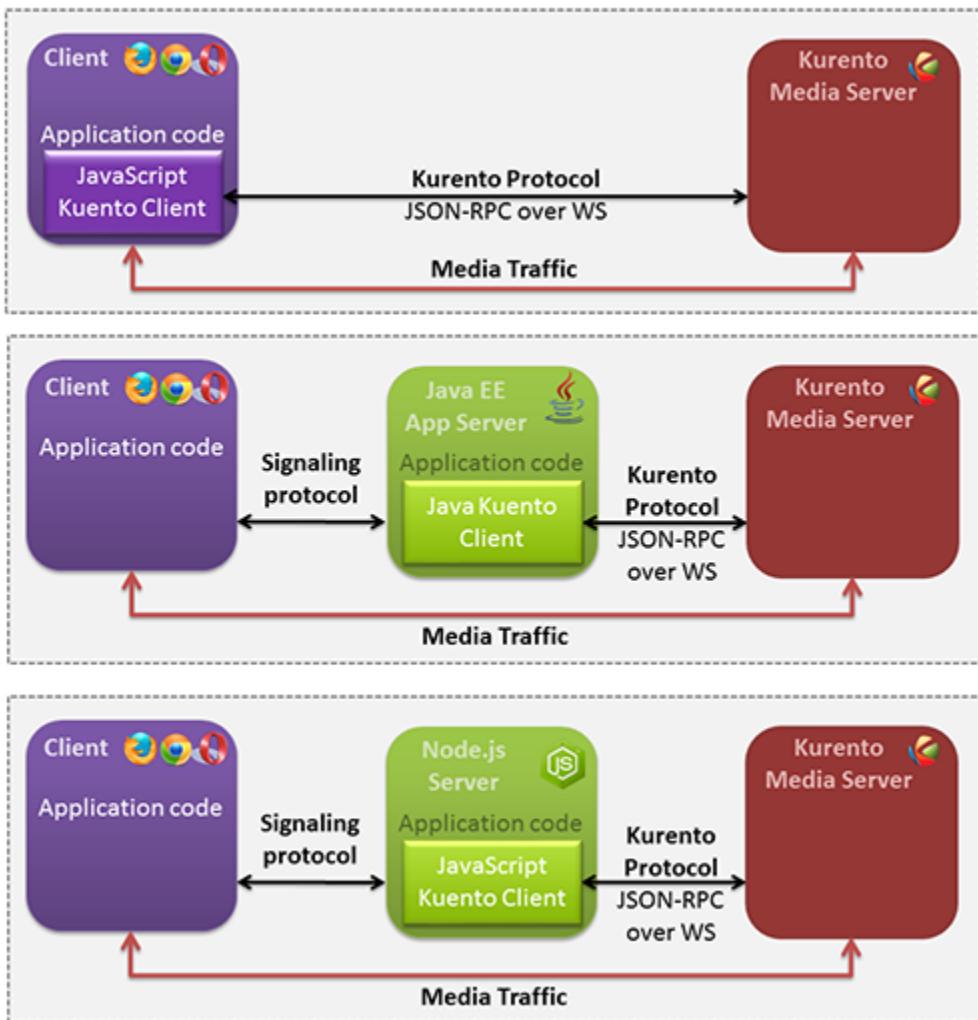
Fig. 1: *Connection of Kurento Clients (Java and JavaScript) to Kuento Media Server*
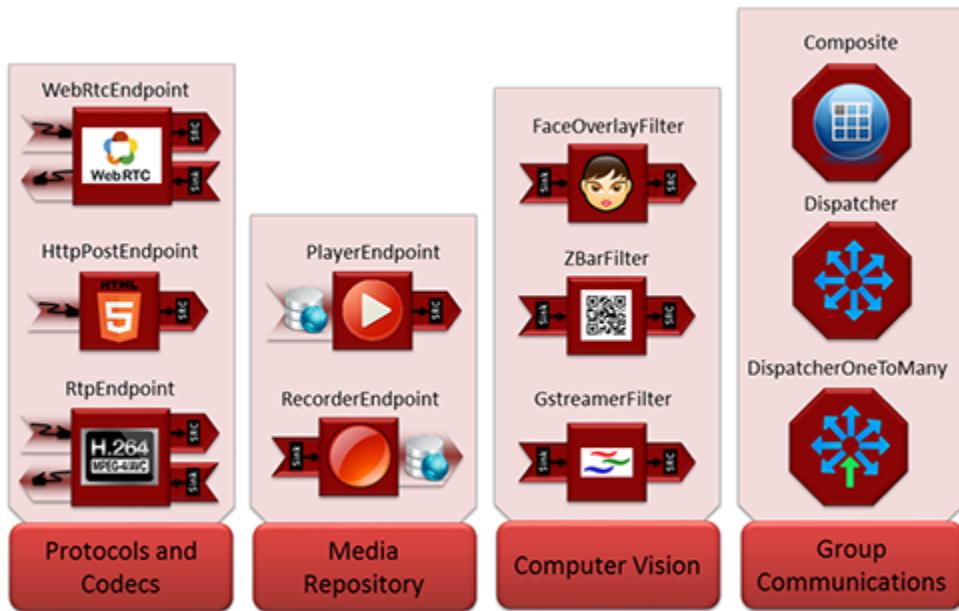
Fig. 2: *Some Media Elements provided out of the box by Kurento*

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 by passing the media-level attribute *goog-remb* in the SDP Offer. For example:
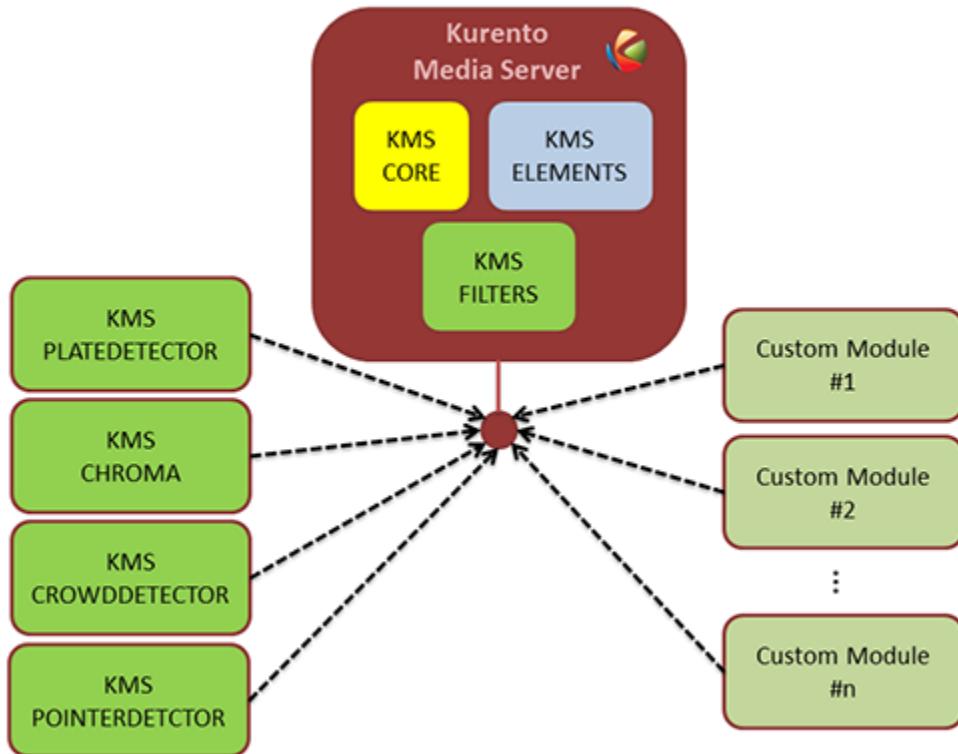
Fig. 3: *Kurento modules architecture. Kurento Media Server can be extended with built-in modules (crowddetector, pointerdetector, chroma, platedetector) and also with other custom modules.*

```
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 the *RTCP Feedback* capability attribute, as defined in **RFC 4585**.

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)*.

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 *Core 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`.

8.7 Service Init

The package *kurento-media-server* provides a service file that integrates with the Ubuntu init system. This service file loads its user configuration from `/etc/default/kurento-media-server`, where the user is able to configure several features as needed.

Writing Kurento Applications

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 with JavaScript code.

- **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:
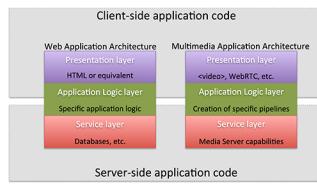
Fig. 1: *Layered architecture of web and multimedia applications. Applications created using Kurento (right) can be similar to standard Web applications (left). Both types of applications may choose to place the application logic at the client or at the server code.*

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:
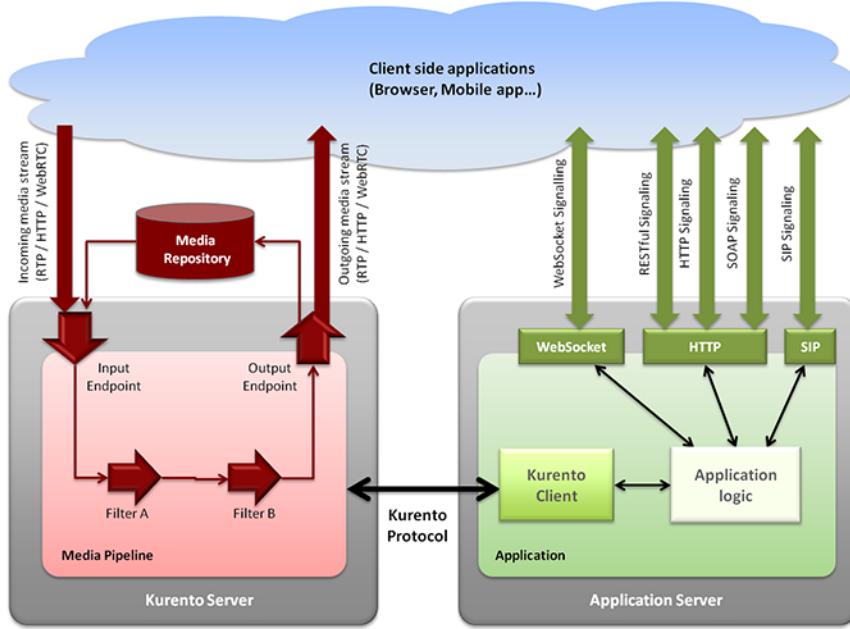
Fig. 2: *Kurento Architecture. Kurento architecture follows the traditional separation between signaling and media planes.*

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 build 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 below, 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 to 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:
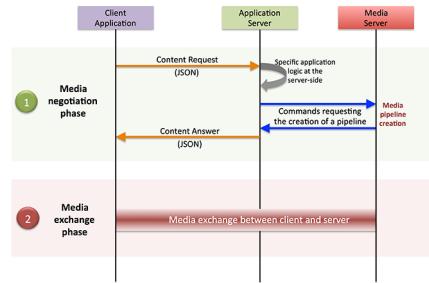
Fig. 3: *Main interactions between architectural modules. These occur in two phases: negotiation and media exchange. Remark that the color of the different arrows and boxes is aligned with the architectural figures presented above. For example, orange arrows show exchanges belonging to the signaling 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:
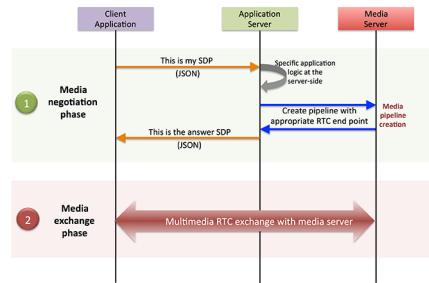
Fig. 4: *Interactions in a WebRTC session. During the negotiation phase, an SDP Offer is sent to KMS, requesting the capabilities of the client. As a result, Kurento Media Server generates an SDP Answer that can be used by the client for establishing the media exchange.*

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.
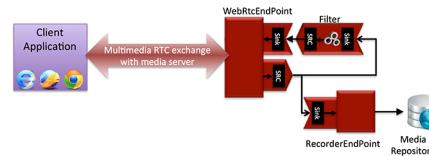
Fig. 5: *Example pipeline for a WebRTC session. A WebRtcEndpoint is connected to a RecorderEndpoint storing the received media stream and to an Augmented Reality filter, which feeds its output media stream back to the client. As a result, the end user will receive its own image filtered (e.g. with a 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).
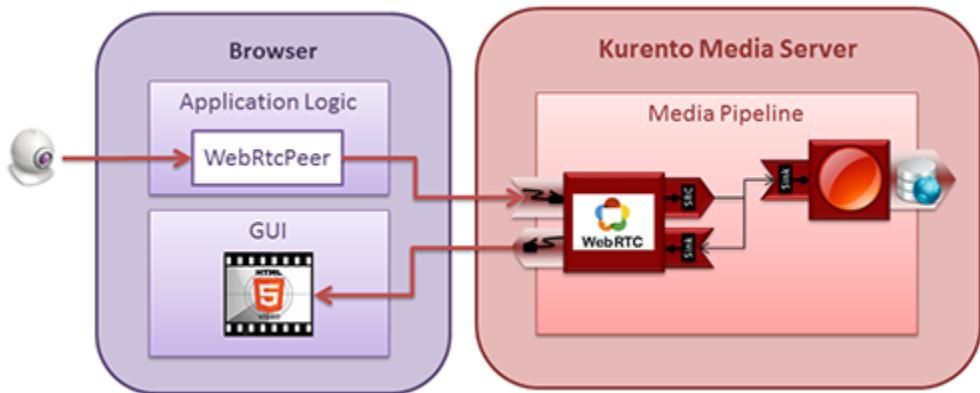
Fig. 6: *Simple Example of a Media Pipeline*

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 latter:

```
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*.

Writing Kurento Modules

Table of Contents

10.1 Scaffolding and development

You can develop your own modules to expand the features of Kurento Media Server. There are two main flavors of Kurento modules:

- Modules based on *OpenCV*. These are recommended if you would like to add features such as **Computer Vision** or **Augmented Reality**.

- Modules based on *GStreamer*. This kind of modules provide a generic entry point for media processing within the GStreamer framework. Such modules are more powerful, but also they are more difficult to develop. It is necessary to have good knowledge of GStreamer development.

The starting point to develop a filter is to create a basic structure for the source code, what we'll call the *scaffolding*. This is done with the *kurento-module-scaffold* tool, which comes included with the *kurento-media-server-dev* package. To install it, run this command:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    kurento-media-server-dev
```

Now use the scaffold tool to generate code for your new module. For example:

- For an OpenCV module:

```
kurento-module-scaffold MyCvModule cv-module-dir true
```

- For a GStreamer module:

```
kurento-module-scaffold MyGstModule gst-module-dir
```

The scaffolding tool generates a complete folder tree, with all the needed *CMakeLists.txt* files to build with CMake. You'll also find empty Kurento Module Descriptor files (*.kmd*), which must contain a complete description of the module: constructor, methods, properties, events, and the complex types defined by the developer.

Once your *.kmd* files have been filled with a complete description of the module, it is time to generate the corresponding server stub code with *kurento-module-creator*. Run this from the root directory of your module:

```
mkdir build/ && cd build/
cmake ..
make
```

The following sections detail how to create your module, depending on the filter type you chose (OpenCV or GStreamer).

10.1.1 OpenCV module

There are several files in `src/server/implementation/objects/`:

```
<ModuleName>Impl.cpp
<ModuleName>Impl.hpp
<ModuleName>OpenCVImpl.cpp
<ModuleName>OpenCVImpl.hpp
```

The first two files contain the server-side implementation of the JSON-RPC API, and should not be modified. The last two files will contain the logic of your module.

The file *<ModuleName>OpenCVImpl.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.

10.1.2 GStreamer module

In this case, these are the files that you'll find under `src/`:

- `src/gst-plugins/` contains the implementation of your GStreamer Element:

```
gst<modulename>.cpp
gst<modulename>.h
<modulename>.c
```

- `src/server/implementation/objects/` contains the server-side implementation of the JSON-RPC API:

```
<ModuleName>Impl.cpp
<ModuleName>Impl.hpp
```

In the file `<ModuleName>Impl.cpp` you have to invoke the methods of your GStreamer element. The actual module logic should be implemented in the GStreamer Element.

10.1.3 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 `src/server/CMakeLists.txt` file.

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_HEADERS*, *INTER-FACE_LIB_EXTRA_INCLUDE_DIRS*, *INTERFACE_LIB_EXTRA_LIBRARIES*

  These parameters allow to add additional source code to the static library. Files included in *INTER-FACE_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_HEADERS*, *SERVER_IMPL_LIB_EXTRA_INCLUDE_DIRS*, *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[<VersionRange>]
```

  where *<VersionRange>* can use these symbols: `AND`, `OR`, `<`, `<=`, `>`, `>=`, `^`, and `~`.

Note:

  - `^` indicates a "compatible" version, under the definition of *Semantic Versioning*.

  - `~` indicates a "similar" version, again according to the definition of SemVer.

10.2 Installation and usage

Before being able to use your new module, its binary files must be installed to the host where Kurento Media Server is running. Using a module with Kurento comprises two sides of the same coin:

1. Install the module. This allows KMS to know about the module, and allows the media server to use it when clients attempt to instantiate a new object that is exported by the module code.

2. Use the module from client applications. Technically this step is optional, but unless your application directly implements the *Kurento Protocol*, you will want to use the client-side module API. This is done by using client code that gets automatically generated from the Kurento Module Descriptor files (*.kmd*).

10.2.1 Installing locally

The recommended way to distribute a module is to build it into a Debian package file (*.deb*). This is the easiest and most convenient method for end users of the module, as they will just have to perform a simple package installation on any system where KMS is already running. Besides, this doesn't require the user to know anything about plugin paths or how the module files must be laid out on disk.

To build a Debian package file, you can either use the **kurento-buildpackage** tool as described in *Create Deb packages*, or do it manually by installing and running the appropriate tools:

```
# Install dpkg-buildpackage, the Debian package builder
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    dpkg-dev

# Run dpkg-buildpackage to build Debian packages
dpkg-buildpackage -us -uc

# Copy the generated packages to their final destination
cp ../*.*deb /path/to/destination/
```

The Debian builder tool ends up generating one or more *.deb* package files **in the parent directory** from where it was called, together with some additional files that can be ignored. For example:

```
$ ls -1 ../*.*deb
../my-gst-module-dev_0.0.1~rc1_amd64.deb
../my-gst-module_0.0.1~rc1_amd64.deb
```

Depending on the contents of the module project, the Debian package builder can generate multiple *.deb* files:

- The file without any suffix contains the shared library code that has been compiled from our source code. This is the file that end users of the module will need to install in their systems.

- *-dev* packages contain header files and are used by *other developers* to build their software upon the module's code. This is not needed by end users.

- *-doc* packages usually contain *manpages* and other documentation, if the module contained any.

- *-dbg* and *-dbgsym* packages contain the debug symbols that have been extracted from the compilation process. It can be used by other developers to troubleshoot crashes and provide bug reports.

Now copy and install the package(s) into any Debian- or Ubuntu-based system where KMS is already installed:

```
sudo dpkg -i my-gst-module_0.0.1~rc1_amd64.deb
```

For more information about the process of creating Debian packages, check these resources:

- Debian Building Tutorial

- Debian Policy Manual

Alternatively, it is also possible to just build the module and manually copy its binary files to the destination system. You can then define the following environment variables in the file `/etc/default/kurento`, to instruct KMS about where the plugin files have been copied:

```
KURENTO_MODULES_PATH+=" /path/to/module"
GST_PLUGIN_PATH+=" /path/to/module"
```

KMS will then add these paths to the path lookup it performs at startup, when looking for all available plugins.

When ready, you should **verify the module installation**. Run KMS twice, with the `--version` and `--list` arguments. The former shows a list of all installed modules and their versions, while the latter prints a list of all the actual *MediaObject* Factories that clients can invoke with the JSON-RPC API. Your own module should show up in both lists:

```
$ /usr/bin/kurento-media-server --version
Kurento Media Server version: 6.12.0
Found modules:
    'core' version 6.12.0
    'elements' version 6.12.0
    'filters' version 6.12.0
    'mygstmodule' version 0.0.1~0.gd61e201

$ /usr/bin/kurento-media-server --list
Available factories:
    [...]
    MyGstModule
    mygstmodule.MyGstModule
```

10.2.2 Installing in Docker

It is perfectly possible to install and use additional Kurento modules with Docker-based deployments of Kurento Media Server. To do so, first follow any of the installation methods described above, but then instead of copying files to a host server you would add them into a Docker image or container.

Our recommendation is to leverage the FROM feature of *Dockerfiles*, to derive directly from a Kurento Docker image, and create your own fully customized image.

A *Dockerfile* such as this one would be a good enough starting point:

```
FROM kurento/kurento-media-server:latest
COPY my-gst-module_0.0.1~rc1_amd64.deb /
RUN dpkg -i /my-gst-module_0.0.1~rc1_amd64.deb
```

Now build the new image:

```
$ docker build --tag kms-with-my-gst-module:latest .
Step 1/3 : FROM kurento/kurento-media-server:latest
Step 2/3 : COPY my-gst-module_0.0.1~rc1_amd64.deb /
Step 3/3 : RUN dpkg -i /my-gst-module_0.0.1~rc1_amd64.deb
Successfully built d10d3b4a8202
Successfully tagged kms-with-my-gst-module:latest
```

And verify your module is correctly loaded by KMS:

```
$ docker run --rm kms-with-my-gst-module:latest --version
Kurento Media Server version: 6.12.0
Found modules:
    'core' version 6.12.0
    'elements' version 6.12.0
    'filters' version 6.12.0
    'mygstmodule' version 0.0.1~0.gd61e201

$ docker run --rm kms-with-my-gst-module:latest --list
Available factories:
    [...]
    MyGstModule
    mygstmodule.MyGstModule
```

10.2.3 Java client code

Run this from the root directory of your module:

```
mkdir build/ && cd build/
cmake .. -DGENERATE_JAVA_CLIENT_PROJECT=TRUE
```

This generates a `build/java/` directory, containing all the client code. You can now run `make java` (equivalent to *mvn package*) to build the Java code and package it, or `make java_install` (equivalent to *mvn install*) to build the package *and* install it into the local Maven repository (typically located at *$HOME/.m2/*). To use the module in your Maven project, you have to add the dependency to the *pom.xml* file:

```xml
<dependency>
  <groupId>org.kurento.module</groupId>
  <artifactId>modulename</artifactId>
  <version>0.0.1</version>
</dependency>
```

10.2.4 JavaScript client code

Run this from the root directory of your module:

```
mkdir build/ && cd build/
cmake .. -DGENERATE_JS_CLIENT_PROJECT=TRUE
```

This generates a `build/js/` directory, containing all the client code. You can now manually copy this code to your application. Alternatively, you can use *Bower* (for *Browser JavaScript*) or *NPM* (for *Node*). To do that, you should add your JavaScript module as a dependency in your *bower.json* or *package.json* file, respectively:

```
"dependencies": {
  "modulename": "0.0.1"
}
```

10.3 Examples

Simple examples for both kinds 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.

Besides that, all of the Kurento main modules are developed using this methodology, so you can also have a look in these:

- kms-core.

- kms-elements.

- kms-filters.

Frequently Asked Questions

11.1 About NAT, ICE, STUN, TURN

These are very important concepts that developers must understand well to work with WebRTC. Here is a collection of all Kurento material talking about these acronyms:

- Glossary:

    - *What is NAT*?

    - *What is NAT Traversal*?

    - *What is ICE*?

    - *What is STUN*?

    - *What is TURN*?

    - *How does TURN work?*

- Installing and configuring a STUN/TURN server:

    - *How to install Coturn?*

    - *How to test my STUN/TURN server?*

    - *How to configure STUN/TURN in Kurento?*

- Troubleshooting *WebRTC failures*

- Advanced knowledge: *NAT Types and NAT Traversal*

11.1.1 When are STUN and TURN needed?

STUN (and possibly *TURN*) is needed **for every WebRTC participant behind a NAT**. All peers that try to connect from behind a *NAT* will need to auto-discover their own external IP address, and also open up ports for RTP data transmission, a process that is known as *NAT Traversal*. This is achieved by using a STUN server which must be deployed **outside of the NAT**.

The STUN server uses a single port for client connections (3478 by default), so this port should be opened up for the public in the server's network configuration or *Security Group*. If using TURN relay, then the whole range of TURN ports (49152 to 65535 by default) should be opened up too, besides the client port. Depending on the features of the STUN/TURN server, these might be only UDP or both UDP and TCP ports. For example, *Coturn* uses both UDP and TCP in its default configuration.

If you are installing Kurento in a NAT environment (e.g. if your media server is behind a NAT firewall), you also need to configure an external STUN server, in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini` (check *How to configure STUN/TURN in Kurento?* for more details). Similarly, all browser clients that are behind a NAT need to use the STUN server through the *iceServers* field of the RTCPeerConnection constructor.

Example:

Kurento Media Server and its Application Server are running in a cloud machine **without any NAT** or port restriction on incoming connections, while a browser client runs from a possibly restricted *NAT* network that forbids incoming connections on any port that hasn't been "opened" in advance

The browser client may communicate with the Application Server for signaling purposes, but at the end of the day the bulk of the audio/video RTP transmission is done between the WebRTC engines of the browser and KMS.
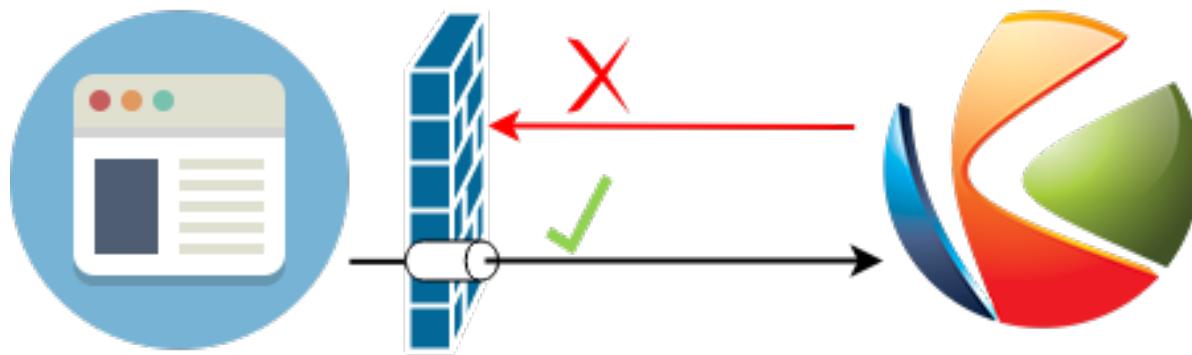
In scenarios like this, the client is able to send data to KMS because its NAT will allow outgoing packets. However, KMS will *not* be able to send data to the client, because the client's NAT is closed for incoming packets. This is solved by configuring the client to use a STUN server; this server will be used by the client's browser to open the appropriate ports in its own NAT. After this operation, the client is now able to receive audio/video streams from KMS:

This procedure is done by the *ICE* implementation of the client's browser.

Note that you *can* also deploy KMS behind a NAT firewall, as long as KMS itself is also configured to use a STUN server.

Further reading:

- WebRTC - How many STUN/TURN servers do I need to specify?.
- What are STUN, TURN, and ICE? (archive).

11.1.2 How does TURN work?

This is a *very* simplified explanation of TURN; for the complete details on how it works, read the **RFC 8656** (*Traversal Using Relays around NAT (TURN)*).

TURN separates two network segments that cannot connect directly (otherwise, STUN and direct connections would be used). In order to allow for maximum probabilities of successful connections, TURN servers such as Coturn will enable both UDP and TCP protocols by default.
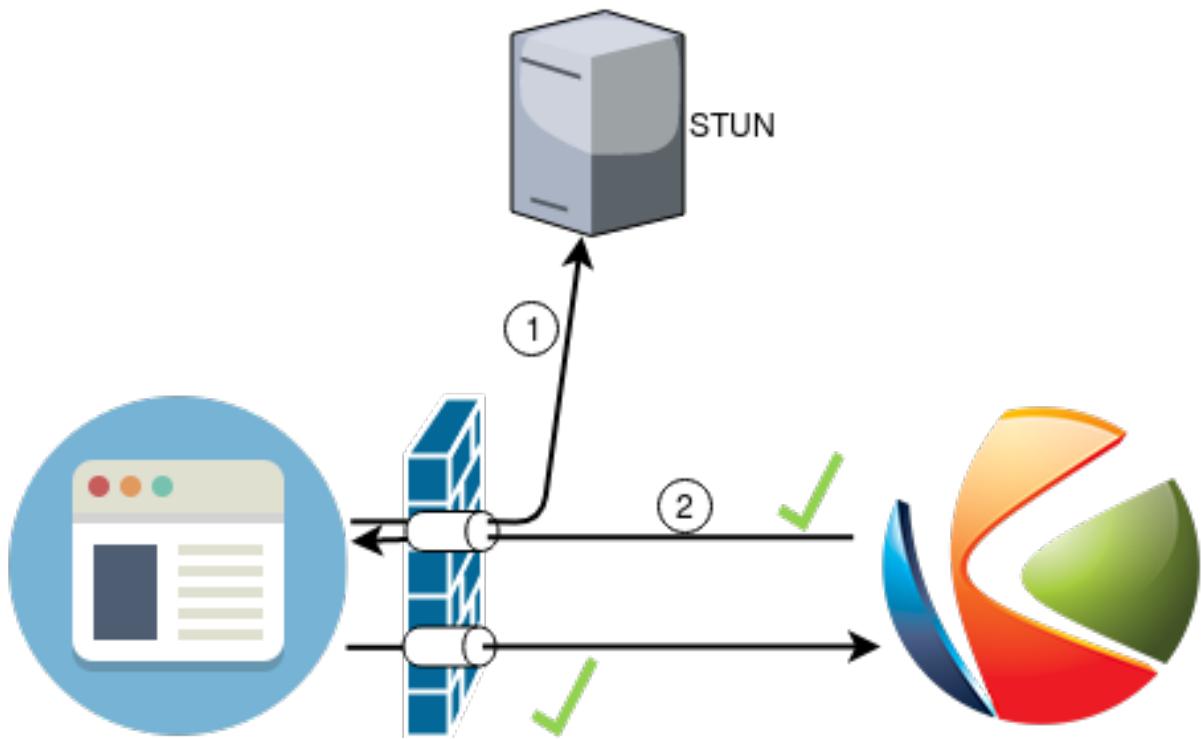
- When a WebRTC participant is behind a strict NAT or firewall that requires relay, it becomes a **TURN client**, contacting the TURN server on its client listening port (3478 by default, either UDP or TCP), and requesting a **TURN relay transport**.

    - The TURN server listens for client requests on both UDP and TCP ports, to maximize the chances that the client's firewall will allow the connection.

    - The *TURN relay transport*, mentioned above, is a random port selected on the **TURN port range** of the TURN server. This range, again, can be either UDP or TCP, to maximize the chances that remote peers are also able to send RTP data to the server.

- When a remote WebRTC peer wants to send RTP data to the *TURN client*, it doesn't send to it directly, instead it sends data towards the corresponding *TURN relay transport* of the TURN server. Then the server will relay this data through its client port (3478) towards the actual *TURN client*.

11.1.3 How to install Coturn?

Coturn is a *STUN* server and *TURN* relay, supporting all features required for the *ICE* protocol and allowing to establish WebRTC connections from behind a *NAT*.

Coturn can be installed directly from the Ubuntu package repositories:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    coturn
```

To configure it for WebRTC, follow these steps:

1. Edit `/etc/turnserver.conf`.

    This example configuration is a good baseline; it will work for using Coturn with Kurento Media Server for WebRTC streams. However, you may want to change it according to your needs:

```
# The external IP address of this server, if Coturn is behind a NAT.
# It must be an IP address, not a domain name.
external-ip=<CoturnIp>

# STUN listener port for UDP and TCP.
# Default: 3478.
#listening-port=<CoturnPort>

# TURN lower and upper bounds of the UDP relay ports.
# Default: 49152, 65535.
#min-port=49152
#max-port=65535

# Uncomment to run server in 'normal' 'moderate' verbose mode.
# Default: verbose mode OFF.
#verbose

# TURN fingerprints in messages.
fingerprint

# TURN long-term credential mechanism.
lt-cred-mech

# TURN realm used for the long-term credential mechanism.
realm=kurento.org

# TURN static user account for long-term credential mechanism.
user=<TurnUser>:<TurnPassword>

# Set the log file name.
# The log file can be reset sending a SIGHUP signal to the turnserver process.
log-file=/var/log/turnserver/turnserver.log

# Disable log file rollover and use log file name as-is.
simple-log
```

Notes:

- The *external-ip* is necessary in cloud providers which use internal NATs, such as **Amazon EC2** (AWS). Write your server's **public** IP address, like *198.51.100.1*, in the *<CoturnIp>* parameter. **It must be an IP address, not a domain name**.

- Comment out all the TURN parameters if you only want Coturn acting as a STUN server.

- The *user* parameter is the most basic form of authorization to use the TURN relay capabilities. Write your desired user name and password in the fields *<TurnUser>* and *<TurnPassword>*.

- Other parameters can be tuned as needed. For more information, check the Coturn help pages:

  - Main project page: https://github.com/coturn/coturn/wiki/turnserver

  - Fully commented configuration file: https://github.com/coturn/coturn/blob/master/examples/etc/turnserver.conf

  - Additional docs on configuration: https://github.com/coturn/coturn/wiki/CoturnConfig

2. Edit the file /etc/default/coturn and set

```
TURNSERVER_ENABLED=1
```

so the server starts automatically as a system service daemon.

---

3. Follow with the next sections to test that Coturn is working, and then set it up as your STUN/TURN server in both Kurento Media Server and the WebRTC clients.

11.1.4 How to test my STUN/TURN server?

To test if your *STUN*/*TURN* server is functioning properly, open the Trickle ICE test page. In that page, follow these steps:

1. Remove any server that might be filled in already by default.

2. Fill in your STUN/TURN server details.

   - To only test STUN (TURN relay will not be tested):

     ```
     stun:<StunServerIp>:<StunServerPort>
     ```

   - To test both STUN and TURN:

     ```
     turn:<TurnServerIp>:<TurnServerPort>
     ```

     . . . and also fill in the *TURN username* and *TURN password*.

3. Click on *Add Server*. You should have only **one entry** in the list, with your server details.

4. Click on *Gather candidates*. **Verify** that you get candidates of type *srflx* if you are testing STUN. Likewise, you should get candidates of type *srflx and* type *relay* if you are testing TURN.

   If you are missing any of the expected candidate types, *your STUN/TURN server is not working well* and WebRTC will fail. Check your server configuration, and your cloud provider's network settings.

11.1.5 How to configure STUN/TURN in Kurento?

To configure a *STUN* server or *TURN* relay with Kurento Media Server, you may use either of two methods:

A. Write the parameters into the file `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`. Do this if your settings are static and you know them beforehand.

   To only use STUN server (TURN relay will not be used):

   ```
   stunServerAddress=<StunServerIp>
   stunServerPort=<StunServerPort>
   ```

   <StunServerIp> should be the public IP address of the STUN server. **It must be an IP address, not a domain name**. For example:

   ```
   stunServerAddress=198.51.100.1
   stunServerPort=3478
   ```

   To use both STUN server and TURN relay:

   ```
   turnURL=<TurnUser>:<TurnPassword>@<TurnServerIp>:<TurnServerPort>
   ```

   <TurnServerIp> should be the public IP address of the TURN relay. **It must be an IP address, not a domain name**. For example:

   ```
   turnURL=myuser:mypassword@198.51.100.1:3478
   ```

B. Use the API methods to set the parameters dynamically. Do this if your STUN server details are not known beforehand, or if your TURN credentials are generated on runtime:

To only use STUN server (TURN relay will not be used):

```
setStunServerAddress("<StunServerIp>");
setStunServerPort(<StunServerPort>);
```

Kurento Client API docs: Java, JavaScript.

To use both STUN server and TURN relay:

```
setTurnUrl("<TurnUser>:<TurnPassword>@<TurnServerIp>:<TurnServerPort>");
```

Kurento Client API docs: Java, JavaScript.

Note: **You don't need to configure both STUN and TURN**, because TURN already includes STUN functionality.

The following ports should be open in the firewall or your cloud provider *Security Group*:

- **<CoturnPort>** (Default: 3478) UDP & TCP, unless you disable either UDP or TCP in Coturn (for example, with `no-tcp`).
- **49152 to 65535** UDP & TCP: As per **RFC 8656**, this port range will be used by a TURN relay to exchange media by default. These ports can be changed using Coturn's `min-port` and `max-port` parameters. Again, you can disable using either TCP or UDP for the relay port range (for example, with `no-tcp-relay`).

Note: **Port ranges do NOT need to match between Coturn and Kurento Media Server**.

If you happen to deploy both Coturn and KMS in the same machine, we recommend that their port ranges do not overlap.

When you are done, (re)start both Coturn and Kurento servers:

```
sudo service coturn restart
sudo service kurento-media-server restart
```

11.2 How many Media Pipelines do I need for my Application?

A Pipeline is a top-level container that handles every resource that should be able to achieve any kind of interaction with each other. Media Elements can only communicate when they are part of the same Pipeline. Different Pipelines in the server are independent, so they do not share audio, video, data or events.

99% times, this translates to using 1 Pipeline object for each "room"-like videoconference. It doesn't matter if there is 1 single presenter and N viewers ("one-to-many"), or if there are N participants Skype-style ("many-to-many"), all of them are managed by the same Pipeline. So, most actual real-world applications would only ever create 1 Pipeline, because that's good enough for most needs.

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 exchanged with a participant.

11.3 How many Endpoints do I need?

Your application will need to create at least one Endpoint for each media stream flowing to (or from) each participant. You might actually need more, if the streams are to be recorded or if streams are being duplicated for other purposes.

11.4 Which participant corresponds to which Endpoint?

The Kurento API offers no way to get application-level semantic attributes stored in a Media Element. However, the application developer can maintain a HashMap or equivalent data structure, storing the Endpoint identifiers (which are plain strings) to whatever application information is desired, such as the names of the participants.

Troubleshooting Issues

If you are facing an issue with Kurento Media Server, follow this basic check list:

- **Step 1**: Test with the **latest version** of Kurento Media Server: **6.15.0**. Follow the installation instructions here: *Installation Guide*.

- **Step 2**: Test with the latest (unreleased) changes by installing a nightly version: *Installing Nightly Builds*.

- **Step 3**: Search for your issue in our GitHub bugtracker and the Kurento Public Mailing List.

- **Step 4**: If you want full attention from the Kurento team, get in contact with us to request *Commercial Support*.

For more information about how to request support, and how to submit bug reports and commercial enquiries, have a look at the *Support* page.

My Kurento Media Server doesn't work, what should I do?

This document outlines several bits of knowledge that can prove very useful when studying a failure or error in KMS:

12.1 Media Server Crashes

We want Kurento to be as stable as possible! When you notice a server crash, it's a good time to report a bug so we can know about the issue. But before that, you need to check a couple things:

1. Make sure that you are running the **latest version** of Kurento Media Server: **6.15.0**.

2. Have *debug symbols installed*. Otherwise, *your bug report won't be useful*.

Then, please provide us with information about the crash:

- Kurento tries to write an **execution stack trace** in the file `/var/log/kurento-media-server/errors.log`. Open the *errors.log* file and look for a line similar to this one:

```
2019-09-19T13:44:48+02:00 -- New execution
```

  Then, see if you can find the stack trace that matches with the time when the crash occurred. Attach that stack trace to your bug report.

- If you installed Kurento with *apt-get install*, *and* Apport is installed, then Ubuntu generates a **crash report** that

you will find in `/var/crash/_usr_bin_kurento-media-server.<PID>.crash`. This contains information that can be used to inspect KMS with a debugger, so it can be very useful to find the cause of the crash. Upload it somewhere, or attach it to your bug report.

Note: The *.crash* report file **is already compressed**, so you can go ahead and upload it to some file transfer service to share it with us.

Note: The *.crash* report file **must be deleted** afterwards. *If an old crash report exists, new ones will not be generated.* So if you are experiencing crashes, make sure that the crash report file is always deleted after having shared it with us, so future crashes will also generate new report files.

- Otherwise, you can manually enable the generation of a **core dump** whenever KMS crashes. For this, edit the file `/etc/default/kurento-media-server` and uncomment the setting *DAEMON_CORE_PATTERN*, which by default will tell the Linux Kernel to generate core dumps in `/tmp/`.

Note: The *core dump* file **is NOT compressed**, so before uploading you should compress it, for a typically huge file size reduction, before uploading it to some file transfer service and sharing it with us.

- As a last resort, if no crash report can be obtained by any means, you may need to run KMS with a debugger. To do so, please follow the instructions here *Run and debug with GDB*, to get a **backtrace** when the crash happens.

- Finally, if a developer suspects that the crash might be due to a memory corruption error, we could ask you to run with a special build of Kurento that comes bundled with support for AddressSanitizer, a memory access error detector.

  To do this, you'll need to run a Kurento Docker image with AddressSanitizer. If we ask for it, you would have to provide the Docker logs from running this image.

  For this reason (and also for better test repeatability), it's a very good idea that you have your services planned in a way that it's possible to **run Kurento Media Server from Docker**, at any time, regardless of what is your normal / usual method of deploying Kurento.

12.2 Other Media Server issues

12.2.1 Reached limit / Resource temporarily unavailable

If the server is malfunctioning or crashing and you can see a message similar to this one in the logs:

```
Reached KMS files limit: 819 (system max: 1024)
```

or

```
Error creating thread: Resource temporarily unavailable
```

then KMS is hitting resource limits imposed by the Kernel. The 2 most common reasons for this issue are:

1. You might have a custom service or Kurento plugin that is acquiring resources without releasing them afterwards. You should profile and debug your code to make sure that it is not leaking resources (such as open file descriptors, threads, sockets, etc) and exhausting the limits enforced by the Linux Kernel.

2. Congratulations! Your service is growing, time to deal with resource & concurrency issues.

   Check the running stats of your operating system, to see if the KMS process is being limited to the default 1024 file/device handles (*ulimit*), and increase that number.

   For local installations (*with apt-get install*), you can edit the file `/etc/default/kurento-media-server` to raise either of the *DAEMON_MAX_FILES* and/or *DAEMON_MAX_THREADS* and see if this solves the issue. For other installation methods such as Docker, you will need to use tool-specific mechanisms to change resource limits, like running with `docker run --ulimit`.

   If **systemd** is configured, it might also be applying its own limit on process resources; for example you can check how many threads are being used by Kurento and what is the maximum, with these commands:

   ```
   systemctl status kurento-media-server | grep Tasks
   systemctl show kurento-media-server | grep TasksMax
   ```

   In *systemd* it is possible to configure limits with parameters such as *DefaultTasksMax* in either `/etc/systemd/system.conf` (for the default system instance) or `/etc/systemd/user.conf` (for the custom user instance, if you have configured one in your machine). The current effective value of *DefaultTasksMax* can be queried with this command:

   ```
   systemctl show --property DefaultTasksMax
   ```

   If left to its default setting, *DefaultTasksMax* will be 15% of the absolute maximum limit from the Kernel, that you can get or set through the file `/proc/sys/kernel/pid_max`. If you change that, don't forget to run `systemctl daemon-reexec` to have *systemd* load the new value.

   Note: You need to use `systemctl daemon-reexec` because `systemctl daemon-reload` has a bug in *systemd* <= v242 (so this affects Ubuntu versions earlier than 20.04 "Focal").

12.2.2 `GStreamer-CRITICAL` messages in the log

GLib and GStreamer are libraries that use a lot of internal `assert()` functions to check for valid conditions whenever a function is called. Normally, these are meant to catch programming bugs in their own source code; when (if) any of these checks fail, a warning message is printed to the logs. The media server won't be brought down in this situation, but a bug in any of Kurento's underlying 3rd-party libraries will have an adverse effect on Kurento itself sooner or later. So, it's in our best interest to watch out for these. Report them to us if you see any! ;-)

Here are a couple examples of such messages:

```
(kurento-media-server:4619): GStreamer-CRITICAL **: gst_element_query: assertion 'GST_
→IS_ELEMENT (element)' failed
```

```
(kurento-media-server:15636): GLib-CRITICAL **: g_error_free: assertion 'error != NULL
→' failed
```

The problem of these messages is that they don't really provide much information about *how* the error happens, of *where*. To find out, we'll need you to run KMS under a debug session. Please, follow the instructions here *Run and debug with GDB*, to get a **backtrace** from the *GStreamer-CRITICAL* error.

12.2.3 CPU usage grows too high

Kurento Media Pipelines can get pretty complex if your use case requires so, which would mean more processing power is required to run them; however, even for the simplest cases it's possible that you find out unexpected spikes

in CPU usage, which in extreme cases could end up crashing the server due to resource exhaustion in the machine.

Check these points in an attempt to find possible causes for the high CPU usage:

- Kurento Media Server is known to work well with videos of up to **720p** resolution (1280x720) at **30fps** and around **2Mbps**. Using values beyond those might work fine, but the Kurento team hasn't done any factual analysis to prove it. With heavier data loads there is a chance that KMS will be unable to process all incoming data on time, and this will cause that buffers fill up and frames get dropped. Try reducing the resolution of your input videos if you see video stuttering.

- Source and destination video codecs must be compatible. This has always been a source of performance problems in WebRTC communications.

  - For example, if some participants are using Firefox and talking in a room, they will probably negotiate **VP8** codec with Kurento; then later someone enters with Safari, CPU usage explodes due to transcoding is now suddenly required, because Safari only supports **H.264** (VP8 support was added only since Desktop Safari v68).

  - Another example is you have some VP8 streams running nicely but then stream recording is enabled with the **MP4** recording profile, which uses H.264. Same story: video needs to be converted, and that uses a lot of CPU.

- Also check if other processes are running in the same machine and using the CPU. For example, if Coturn is running and using a lot of resources because too many users end up connecting via Relay (TURN).

Of these, video transcoding is the main user of CPU cycles, because encoding video is a computationally expensive operation. As mentioned earlier, keep an eye on the *TRANSCODING* events sent from Kurento to your Application Server, or alternatively look for *TRANSCODING ACTIVE* messages in the media server logs.

If you see that transcoding is active at some point, you may get a bit more information about why, by enabling this line:

```
export GST_DEBUG="${GST_DEBUG:-3},Kurento*:5,agnosticbin*:5"
```

in your daemon settings file, `/etc/default/kurento-media-server`.

Then look for these messages in the media server log output:

- `Upstream provided caps:  (caps)`
- `Downstream wanted caps:  (caps)`
- `Find TreeBin with wanted caps:  (caps)`

Which will end up with either of these sets of messages:

- If source codec is compatible with destination:

  - `TreeBin found! Use it for (audio|video)`
  - `TRANSCODING INACTIVE for (audio|video)`

- If source codec is **not** compatible with destination:

  - `TreeBin not found! Transcoding required for (audio|video)`
  - `TRANSCODING ACTIVE for (audio|video)`

These messages can help understand what codec settings are being received by Kurento ("*Upstream provided caps*") and what is being expected at the other side by the stream receiver ("*Downstream wanted caps*").

12.2.4 Memory usage grows too high

If you are trying to establish whether Kurento Media Server has a memory leak, then neither *top* nor *ps* are the right tool for the job; **Valgrind** is.

If you are using *top* or *ps* to evaluate memory usage, keep in mind that these tools show memory usage *as seen by the Operating System*, not by the process of the media server. Even after freeing memory, there is no guarantee that the memory will get returned to the Operating System. Typically, it won't! Memory allocator implementations do not return *free*'d memory : it is available for use by the same program, but not by others. So *top* or *ps* won't be able to "see" the free'd memory.

See: free() in C doesn't reduce memory usage

To run Kurento Media Server with Valgrind and find memory leaks, the process is just a matter of following the steps outlined in *Build from sources*, but instead of

```
./bin/kms-build-run.sh
```

you'll want to do

```
./bin/kms-build-run.sh --valgrind-memcheck
```

Also, please have a look at the information shown in *Media Server Crashes* about our special Docker image based on **AddressSanitizer**. Running Kurento with this image might help finding memory-related issues.

12.2.5 Service init doesn't work

The package *kurento-media-server* provides a service file that integrates with the Ubuntu init system. This service file loads its user configuration from `/etc/default/kurento-media-server`, where the user is able to configure several features as needed.

In Ubuntu, log messages from init scripts are managed by *systemd*, and can be checked in to ways:

- `/var/log/syslog` contains a copy of all init service messages. You can open it to see past messages, or follow it in real time with this command:

```
tail -f /var/log/syslog
```

- You can query the status of the *kurento-media-server* service with this command:

```
systemctl status kurento-media-server.service
```

12.2.6 OpenH264 not found

Problem: Installing and running KMS on a clean Ubuntu installation shows this message:

```
(gst-plugin-scanner:15): GStreamer-WARNING **: Failed to load plugin
'/usr/lib/x86_64-linux-gnu/gstreamer-1.5/libgstopenh264.so': libopenh264.so.0:
cannot open shared object file: No such file or directory
```

Also these conditions apply:

- Packages *openh264-gst-plugins-bad-1.5* and *openh264* are already installed.

- The file `/usr/lib/x86_64-linux-gnu/libopenh264.so` is a broken link to the non-existing file `/usr/lib/x86_64-linux-gnu/libopenh264.so.0`.

Reason: The package *openh264* didn't install correctly. This package is just a wrapper that needs Internet connectivity during its installation stage, to download a binary blob file from this URL: http://ciscobinary.openh264.org/libopenh264-1.4.0-linux64.so.bz2

If the machine is disconnected during the actual installation of this package, the download will fail silently with some error messages printed on the standard output, but the installation will succeed.

Solution: Ensure that the machine has access to the required URL, and try reinstalling the package:

```
sudo apt-get update && sudo apt-get install --reinstall openh264
```

12.2.7 Missing audio or video streams

If the Kurento Tutorials are showing an spinner, or your application is missing media streams, that's a strong indication that the network topology requires using either a *STUN* server or a *TURN* relay, to traverse through the *NAT* of intermediate routers. Check the section about *installing a STUN/TURN server*.

12.2.8 Low video quality

You have several ways to override the default settings for variable bitrate:

- Methods in org.kurento.client.BaseRtpEndpoint:

    - *setMinVideoRecvBandwidth() / setMaxVideoRecvBandwidth()*

    - *setMinVideoSendBandwidth() / setMaxVideoSendBandwidth()*

- Methods in org.kurento.client.MediaElement:

    - *setMinOutputBitrate() / setMaxOutputBitrate()*

        This setting is also configurable in `/etc/kurento/modules/kurento/MediaElement.conf.ini`.

12.2.9 Video has green artifacts

This is typically caused by missing information in the video decoder, most probably due to a high packet loss rate in the network.

The *H.264* and VP8 video codecs use a color encoding system called YCbCr (sometimes also written as *YCrCb*), which the decoder has to convert into the well known RGB ("*Red-Green-Blue*") model that is used by computer screens. When there is data loss, the decoder will assume that all missing values are *0* (zero). It just turns out that a YCbCr value of *(0,0,0)* is equivalent to the **green** color in RGB.

When this problem happens, Kurento sends retransmission requests to the source of the RTP stream. However, in cases of heavy packet loss, there isn't much else that can be done and enough losses will build up until the video decoding gets negatively affected. In situations like this, the most effective change you can do is to reduce the video resolution and/or quality at the sender.

Cisco has too a nice paragraph covering this in their Knowledge Base: Pink and green patches in a video stream (archive)

> **Why do I see pink or green patches in my video stream [. . .]?**
>
> *Pink and green patches or lines seen in decoded video are often the result of packet loss or incorrect data in the video stream. Many video codecs (including H.261, H.263 and H.264) use the Y'CbCr system to represent color space, where Y' is the 'luma' (brightness) component and Cb and Cr are the blue and red chroma components respectively. For many Y'CbCr values there is no equivalent RGB value and the*

colour seen on the display depends on the details of the algorithm used. A Y'CbCr value of (0,0,0) is often converted into the green color while a Y'CbCr value of (255,255,255) leads to a pink color.

If you encounter the symptoms described above, follow normal packet loss and network troubleshooting procedures.

12.3 Application Server

These are some common errors found to affect Kurento Application Servers:

12.3.1 KMS is not running

Usually, the Kurento Client library is directed to connect with an instance of KMS that the developer expects will be running in some remote server. If there is no instance of KMS running at the provided URL, the Kurento Client library will raise an exception which **the Application Server should catch** and handle accordingly.

This is a sample of what the console output will look like, with the logging level set to DEBUG:

```
$ mvn -U clean spring-boot:run \
      -Dspring-boot.run.jvmArguments="-Dkms.url=ws://localhost:8888/kurento"
INFO org.kurento.tutorial.player.Application  : Starting Application on TEST with PID␣
↪16448
DEBUG o.kurento.client.internal.KmsUrlLoader  : Executing getKmsUrlLoad(b843d6f6-02dd-
↪49b4-96b6-f2fd2e8b1c8d) in KmsUrlLoader
DEBUG o.kurento.client.internal.KmsUrlLoader  : Obtaining kmsUrl=ws://localhost:8888/
↪kurento from config file or system property
DEBUG org.kurento.client.KurentoClient        : Connecting to kms in ws://
↪localhost:8888/kurento
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket      : Creating JsonRPC NETTY Websocket␣
↪client
DEBUG o.kurento.jsonrpc.client.JsonRpcClient  : Enabling heartbeat with an interval␣
↪of 240000 ms
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket  : [KurentoClient]  Connecting webSocket␣
↪client to server ws://localhost:8888/kurento
WARN o.kurento.jsonrpc.client.JsonRpcClient   : [KurentoClient]  Error sending␣
↪heartbeat to server. Exception: [KurentoClient]  Exception connecting to WebSocket␣
↪server ws://localhost:8888/kurento
WARN o.kurento.jsonrpc.client.JsonRpcClient   : [KurentoClient]  Stopping heartbeat␣
↪and closing client: failure during heartbeat mechanism
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket  : [KurentoClient]  Connecting webSocket␣
↪client to server ws://localhost:8888/kurento
DEBUG o.k.jsonrpc.internal.ws.PendingRequests : Sending error to all pending requests
WARN o.k.j.c.JsonRpcClientNettyWebSocket       : [KurentoClient]  Trying to close a␣
↪JsonRpcClientNettyWebSocket with channel == null
WARN ationConfigEmbeddedWebApplicationContext : Exception encountered during context␣
↪initialization - cancelling refresh attempt: Factory method 'kurentoClient' threw␣
↪exception; nested exception is org.kurento.commons.exception.KurentoException:␣
↪Exception connecting to KMS
ERROR o.s.boot.SpringApplication               : Application startup failed
```

As opposed to that, the console output for when a connection is successfully done with an instance of KMS should look similar to this sample:

```
$ mvn -U clean spring-boot:run \
      -Dspring-boot.run.jvmArguments="-Dkms.url=ws://localhost:8888/kurento"
```

```
INFO org.kurento.tutorial.player.Application : Starting Application on TEST with PID␣
↪21617
DEBUG o.kurento.client.internal.KmsUrlLoader : Executing getKmsUrlLoad(af479feb-dc49-
↪4a45-8b1c-eedf8325c482) in KmsUrlLoader
DEBUG o.kurento.client.internal.KmsUrlLoader : Obtaining kmsUrl=ws://localhost:8888/
↪kurento from config file or system property
DEBUG org.kurento.client.KurentoClient       : Connecting to kms in ws://
↪localhost:8888/kurento
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket    : Creating JsonRPC NETTY Websocket client
DEBUG o.kurento.jsonrpc.client.JsonRpcClient : Enabling heartbeat with an interval of␣
↪240000 ms
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  Connecting webSocket␣
↪client to server ws://localhost:8888/kurento
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Connecting native␣
↪client
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Creating new␣
↪NioEventLoopGroup
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Initiating new Netty␣
↪channel. Will create new handler too!
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket    : [KurentoClient]  channel active
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket    : [KurentoClient]  WebSocket Client␣
↪connected!
INFO org.kurento.tutorial.player.Application : Started Application in 1.841 seconds␣
↪(JVM running for 4.547)
```

12.3.2 KMS became unresponsive (due to network error or crash)

The Kurento Client library is programmed to start a retry-connect process whenever the other side of the RPC channel
-ie. the KMS instance- becomes unresponsive. An error exception will raise, which again **the Application Server
should handle**, and then the library will automatically start trying to reconnect with KMS.

This is how this process would look like. In this example, KMS was restarted so the Kurento Client library lost
connectivity with KMS for a moment, but then it was able con reconnect and continue working normally:

```
INFO org.kurento.tutorial.player.Application  : Started Application in 1.841 seconds␣
↪(JVM running for 4.547)

(... Application is running normally at this point)
(... Now, KMS becomes unresponsive)

INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  channel closed
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  JsonRpcWsClient␣
↪disconnected from ws://localhost:8888/kurento because Channel closed.
DEBUG o.kurento.jsonrpc.client.JsonRpcClient : Disabling heartbeat. Interrupt if␣
↪running is false
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  JsonRpcWsClient␣
↪reconnecting to ws://localhost:8888/kurento.
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  Connecting webSocket␣
↪client to server ws://localhost:8888/kurento
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Connecting native␣
↪client
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Closing previously␣
↪existing channel when connecting native client
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket    : [KurentoClient]  Closing client
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Initiating new Netty␣
↪channel. Will create new handler too!
```

```
WARN o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Trying to close a␣
→JsonRpcClientNettyWebSocket with channel == null
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : tryReconnectingForever = true
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : tryReconnectingMaxTime = 0
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : maxTimeReconnecting =␣
→9223372036854775807
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : currentTime = 1510773733903
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : Stop connection retries: false
WARN o.k.j.c.AbstractJsonRpcClientWebSocket  : [KurentoClient]  Exception trying to␣
→reconnect to server ws://localhost:8888/kurento. Retrying in 5000 ms

org.kurento.jsonrpc.JsonRpcException: [KurentoClient]  Exception connecting to␣
→WebSocket server ws://localhost:8888/kurento
   at (...)
Caused by: io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection␣
→refused: localhost/127.0.0.1:8888
   at (...)

(... Now, KMS becomes responsive again)

DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  JsonRpcWsClient␣
→reconnecting to ws://localhost:8888/kurento.
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  Connecting webSocket␣
→client to server ws://localhost:8888/kurento
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Connecting native␣
→client
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Creating new␣
→NioEventLoopGroup
INFO o.k.j.c.JsonRpcClientNettyWebSocket     : [KurentoClient]  Initiating new Netty␣
→channel. Will create new handler too!
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket    : [KurentoClient]  channel active
DEBUG o.k.j.c.JsonRpcClientNettyWebSocket    : [KurentoClient]  WebSocket Client␣
→connected!
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient] Req-> {"id":2,"method
→":"connect","jsonrpc":"2.0"}
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  <-Res {"id":2,"result
→":{"serverId":"1a3b4912-9f2e-45da-87d3-430fef44720f","sessionId":"f2fd16b7-07f6-
→44bd-960b-dd1eb84d9952"},"jsonrpc":"2.0"}
DEBUG o.k.j.c.AbstractJsonRpcClientWebSocket : [KurentoClient]  Reconnected to the␣
→same session in server ws://localhost:8888/kurento

(... At this point, the Kurento Client is connected again to KMS)
```

12.3.3 Node / NPM failures

Kurento Client does not currently support Node v10 (LTS), you will have to use Node v8 or below.

12.3.4 "Expects at least 4 fields"

This message can manifest in multiple variations of what is essentially the same error:

```
DOMException: Failed to parse SessionDescription: m=video 0 UDP/TLS/RTP/SAVPF Expects␣
→at least 4 fields
```

```
OperationError (DOM Exception 34): Expects at least 4 fields
```

The reason for this is that Kurento hasn't enabled support for the video codec H.264, but it needs to communicate with another peer which only supports H.264, such as the Safari browser. Thus, the SDP Offer/Answer negotiation rejects usage of the corresponding media stream, which is what is meant by `m=video 0`.

The solution is to ensure that both peers are able to find a match in their supported codecs. To enable H.264 support in Kurento, check these points:

- The package *openh264-gst-plugins-bad-1.5* must be installed in the system.

- The package *openh264* must be **correctly** installed. Specifically, the post-install script of this package requires Internet connectivity, because it downloads a codec binary blob from the Cisco servers. See *OpenH264 not found*.

- The H.264 codec must be enabled in the corresponding Kurento settings file: `/etc/kurento/modules/kurento/SdpEndpoint.conf.json`. Ensure that the entry corresponding to this codec does exist and is not commented out. For example:

```
"videoCodecs": [
  { "name": "VP8/90000" },
  { "name": "H264/90000" }
]
```

12.3.5 "Error: 'operationParams' is required"

This issue is commonly caused by setting an invalid ID to any of the client method calls. The usual solution is to provide a null identifier, forcing the server to generate a new one for the object.

For example, a Node application wanting to use the *ImageOverlayFilter* (Java API, JavaScript API) might mistakenly try to provide an invalid ID in the addImage() call:

```
const filter = await pipeline.create("ImageOverlayFilter");
await filter.addImage("IMAGE_ID", "https://IMAGE_URL", 0.5, 0.5, 0.5, 0.5, true,
→true);
await webRtcEndpoint.connect(filter);
await filter.connect(webRtcEndpoint);
```

This will fail, causing a *MARSHALL_ERROR* in the media server, and showing the following stack trace in the client side:

```
Trace: { Error: 'operationParams' is required
    at node_modules/kurento-client/lib/KurentoClient.js:373:24
    at Object.dispatchCallback [as callback] (node_modules/kurento-jsonrpc/lib/index.
→js:546:9)
    at processResponse (node_modules/kurento-jsonrpc/lib/index.js:667:15)
    [...]
    at WebsocketStream.onMessage (node_modules/websocket-stream/index.js:45:15) code:
→40001, data: { type: 'MARSHALL_ERROR' } }
```

The solution is to simply use *null* for the first argument of the method:

```
await filter.addImage(null, "https://IMAGE_URL", 0.5, 0.5, 0.5, 0.5, true, true);
```

12.4 WebRTC failures

There is a multitude of possible reasons for a failed WebRTC connection, so you can start by following this checklist:

- Deploy a *STUN*/*TURN* server (such as Coturn), to make remote WebRTC connections possible: *How to install Coturn?*.

- Test if your *STUN*/*TURN* server is working correctly: *How to test my STUN/TURN server?*.

- Configure your *STUN*/*TURN* server in Kurento Media Server: *How to configure STUN/TURN in Kurento?*.

- Check the debug logs of your *STUN*/*TURN* server. Maybe the server is failing and some useful error messages are being printed in there.

- Check the debug logs of Kurento Media Server. Look for messages that confirm a correct configuration:

```
INFO  Using STUN reflexive server IP: <IpAddress>
INFO  Using STUN reflexive server Port: <Port>

INFO  Using TURN relay server: <user:password>@<IpAddress>:<Port>
INFO  TURN server info set: <user:password>@<IpAddress>:<Port>
```

- Check that any SDP mangling you (or any of your third-party libraries) might be doing in your Application Server is being done correctly.

  This is one of the most hard to catch examples we've seen in our mailing list:

  > The problem was that our Socket.IO client did not correctly *URL-Encode* its JSON payload when *xhr-polling*, which resulted in all "plus" signs ('+') being changed into spaces (' ') on the server. This meant that the *ufrag* in the client's SDP was invalid if it contained a plus sign! Only some of the connections failed because not all *ufrag* contain plus signs.

- If WebRTC seems to disconnect exactly after some amount of time, every single time, **watch out for proxy timeouts**. Sometimes you have to extend the timeout for the site that is being hit with the problem.

- Have a look at these articles about troubleshooting WebRTC:

  - Troubleshooting WebRTC Connection Issues (archive).

  - Common (beginner) mistakes in WebRTC (archive).

12.4.1 ICE connection problems

If your application receives an *IceComponentStateChange* event with state *FAILED* from Kurento Media Server, it means that the WebRTC ICE connectivity has been abruptly interrupted. In general terms, this implies that **there is some network connectivity issue** between KMS and the remote peer (typically, a web browser), but the exact reason can fall into a myriad possible causes. You will need to investigate what happened on the user's and the server's network when the failure happened.

Here are some tips to keep in mind:

- Check that you have correctly configured a *STUN* server or *TURN* relay, both in Kurento Media Server (file *WebRtcEndpoint.conf.ini*), and in the client browsers (through the RTCPeerConnection's iceServers setting).

- Check that the *TURN* credentials are correct, by using the Trickle ICE test page to test your STUN/TURN server, as explained here: *How to test my STUN/TURN server?*.

- It is always a good idea to work out the **correlation between ICE failures on KMS with ICE failures on the client browser**. The combined logs of both sides might shed some light into what caused the disconnection.

- Analyze all *NewCandidatePairSelected* events emitted by Kurento. A lot of ICE candidates are tested for connectivity during the WebRTC session establishment, but only the actual working ones are reported with the *NewCandidatePairSelected* event. A **careful examination of all selected local and remote candidates** might reveal useful information about the kind of connectivity issues that clients might be having.

  For example, maybe you see that most or all of the selected local or remote candidates are of `typ relay`, i.e. using a *TURN* relay as a proxy for the audio/video streams. This would mean two things:

  1. That the *TURN* relay will be under high server load, possibly saturating the machine's resources.

  2. That **direct peer-to-peer WebRTC connections are not being established**, giving you a good starting point to investigate why this is happening. Usually, when you see usage of the *TURN* relay, this is caused by overzealous hardware or software firewalls, or the presence of Symmetric *NAT* modem/routers somewhere in the network path.

- If you see messages about ICE connection tests failing due to **timeout on trying pairs**, make sure that all required UDP ports for media content are open on the sever; otherwise, not only the ICE process will fail, but also the video or audio streams themselves won't be able to reach each WebRTC peer.

12.4.2 mDNS ICE candidate fails: Name or service not known

Problem:

When the browser conceals the local IP address behind an mDNS candidate, these errors appear in Kurento logs:

```
kmsicecandidate  [...] Error code 0: 'Error resolving '2da1b2bb-a601-44e8-b672-
→dc70e3493bc4.local': Name or service not known'
kmsiceniceagent  [...] Cannot parse remote candidate: 'candidate:2382557538 1 udp␣
→2113937151 2da1b2bb-a601-44e8-b672-dc70e3493bc4.local 50635 typ host generation 0␣
→ufrag /Og/ network-cost 999'
kmswebrtcsession [...] Adding remote candidate to ICE Agent: Agent failed, stream_id:
→'1'
```

Solution:

mDNS name resolution must be enabled in the system. Check out the contents of `/etc/nsswitch.conf`, you should see something similar to this:

```
hosts: files mdns4_minimal [NOTFOUND=return] dns
```

If not, try fully reinstalling the package *libnss-mdns*:

```
sudo apt-get purge --yes libnss-mdns
sudo apt-get update && sudo apt-get install --yes libnss-mdns
```

Installing this package does automatically edit the config file in an appropriate way. Now the *mdns4_minimal* module should appear listed in the hosts line.

Caveat: **mDNS does not work from within Docker**

See mDNS and Crossbar.io Fabric (Docker) #21:

> Docker does not play well with mDNS/zeroconf/Bonjour: resolving *.local* hostnames from inside containers does not work (easily). [...] The reasons run deep into how Docker configures DNS *inside* a container.

So if you are running a Docker image, *.local* names won't be correctly resolved even if you install the required packages. This happens with Kurento or whatever other software; it seems to be a Docker configuration problem / bug.

Disabling mDNS in Chrome

Chrome allows disabling mDNS, which is something that could be useful during development. However when development is finished, don't forget to test your application with default settings, including with this option enabled!

To disable mDNS, open this URL: `chrome://flags/#enable-webrtc-hide-local-ips-with-mdns` and change the setting to "Disabled".

12.5 Docker issues

12.5.1 Publishing Docker ports eats memory

Docker will consume a lot of memory when publishing big enough port ranges. As of this writing, there is no quick and easy solution to this issue.

You should not expose a large port range in your Docker containers; instead, prefer using Host Networking (`--network host`). To elaborate a bit more, as mentioned here:

> the problem is that - given the current state of Docker - it seems you should NOT even be trying to expose large numbers of ports. You are advised to use the host network anyway, due to the overhead involved with large port ranges. (it adds both latency, as well as consumes significant resources - e.g. see https://www.percona.com/blog/2016/02/05/measuring-docker-cpu-network-overhead/)

> If you are looking for a more official source, there is still (for years) an open issue in Docker about this: moby/moby#11185 (comment)

12.5.2 Multicast fails in Docker

Problem:

- Your Kurento Media Server is running in a Docker container.

- MULTICAST streams playback fail with an error such as this one:

```
DEBUG rtspsrc gstrtspsrc.c:7553:gst_rtspsrc_handle_message:<source> timeout on
→UDP port
```

> Note that in this example, to see this message you would need to enable *DEBUG* log level for the *rtspsrc* category; see *Logging levels and components*.

Solution:

For Multicast streaming to work properly, you need to disable Docker network isolation and use `--network host`. Note that this gives the container direct access to the host interfaces, and you'll need to connect through published ports to access others containers.

This is a limitation of Docker; you can follow the current status with this issue: #23659 Cannot receive external multicast inside container.

If using Docker Compose, use `network_mode:  host` such as this:

```
version: "3.7"
services:
  kms:
    image: kurento/kurento-media-server:6.9.0
    container_name: kms
    restart: always
```

(continues on next page)

```
    network_mode: host
    environment:
      - GST_DEBUG=2,Kurento*:5
```

References:

- https://github.com/Kurento/bugtracker/issues/349

- https://stackoverflow.com/questions/51737969/how-to-support-multicast-network-in-docker

12.6 Element-specific info

12.6.1 PlayerEndpoint

RTSP broken video

Some users have reported huge macro-blocks or straight out broken video frames when using a PlayerEndpoint to receive an RTSP stream containing H.264 video. A possible solution to fix this issue is to fine-tune the PlayerEndpoint's **networkCache** parameter. It basically sets the buffer size (in milliseconds) that the underlying GStreamer decoding element will use to cache the stream.

There's no science for that parameter, though. The perfect value depends on your network topology and efficiency, so you should proceed in a trial-and-error approach. For some situations, values lower than **100ms** have worked fine; some users have reported that 10ms was required to make their specific camera work, others have seen good results with setting this parameter to **0ms**.

RTSP Video stuttering

The GStreamer element in charge of RTSP reception is rtspsrc, and this element contains an rtpjitterbuffer.

This jitter buffer gets full when network packets arrive faster than what Kurento is able to process. If this happens, then PlayerEndpoint will start dropping packets, which will show up as video stuttering on the output streams, while triggering a warning in Kurento logs:

```
WARNING  kmsutils  discont_detection_probe() <kmsagnosticbin0:sink>  Stream␣
↪discontinuity detected on non-keyframe
```

You can check if this problem is affecting you by running with DEBUG *logging level* enabled for the *rtpjitterbuffer* component, and searching for a specific message:

```
export GST_DEBUG="${GST_DEBUG:-3},rtpjitterbuffer:5"
/usr/bin/kurento-media-server 2>&1 | grep -P 'rtpjitterbuffer.*(Received packet|Queue␣
↪full)'
```

With this command, a new line will get printed for each single *Received packet*, plus an extra line will appear informing about *Queue full* whenever a packet is dropped.

There is not much you can fine tune in KMS to solve this problem; the most practical solution is to reduce the amount of data, mostly by decreasing either video resolution or video bitrate.

Kurento Media Server is known to work well receiving videos of up to **720p** resolution (1280x720) at **30fps** and around **2Mbps**. If you are using values beyond those, there is a chance that KMS will be unable to process all incoming data on time, and this will cause that buffers fill up and frames get dropped. Try reducing the resolution of your input videos to see if this helps solving the issue.

12.6.2 RecorderEndpoint

Zero-size video files

If you are trying to generate a video recording, keep in mind that **the endpoint will wait until all tracks (audio, video) start arriving**.

Quoting from the Client documentation:

Follow this checklist to see if any of these problems is preventing the RecorderEndpoint from working correctly:

- The RecorderEndpoint is configured for both audio and video, but only video (or only audio) is being provided by the application.
- Availability of audio/video devices at recorder client initialization, and just before starting the recording.
- User is disconnecting existing hardware, or maybe connecting new hardware (usb webcams, mic, etc).
- User is clicking "*Deny*" when asked to allow access to microphone/camera by the browser.
- User is sleeping/hibernating the computer, and then possibly waking it up, while recording.
- Check the browser information about the required media tracks, e.g. *track.readyState*.
- Track user agents, ICE candidates, etc.

12.7 Browser

12.7.1 Safari doesn't work

Apple Safari is a browser that follows some policies that are much more restrictive than those of other common browsers such as Google Chrome or Mozilla Firefox.

For some tips about how to ensure the best compatibility with Safari, check *Apple Safari*.

Support

If you are facing an issue with Kurento Media Server, follow this basic check list:

- Step 1. Test with the **latest version** of Kurento Media Server: **6.15.0**. Follow the installation instructions here: *Installation Guide*.

- Step 2: If the problem still happens in the latest version, and the Kurento developers are already tracking progress for a solution in a bug report or a support contract, you may test the latest (unreleased) changes by installing a nightly version of KMS: *Installing Nightly Builds*.

- Step 3: When your issue exists in both the latest and nightly versions, try resorting to the *Open-Source Community*. Kurento users might be having the same issue and maybe you can find help in there.

- Step 4: If you want full attention from the Kurento team, get in contact with us to request *Commercial Support*.

13.1 Community Support

Kurento is a project mainly supported by its Open-Source Community. **All people answering your questions are doing it with their own time, so please be kind and provide as much information as possible.**

The Kurento Public Mailing List is *the best place to ask* if you have questions about configuration, infrastructure, or general usage of Kurento Media Server.

Another option is Stack Overflow; tag your questions with kurento so other folks can find them easily.

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?

13.1.1 Bugs & Support Issues

You can file bug reports on our Issue Tracker, and they will be addressed as soon as possible.

Support is a volunteered effort, and there is no guaranteed response time. If you need answers quickly, you can get in contact with us to request *Commercial Support*.

13.1.2 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 they make it easier for people to work on the issue, and that means more chances that the issue gets fixed:

- **Be proactive**. If you are working with an old version of Kurento, please check with newer versions, specially with the *nightly version*. We can't emphasize this enough: *it's the first thing that we are going to ask*.

- **Be curious**. Has it been asked before? Is it really a bug? Everybody hates duplicated reports. The Search tool is your friend!

- **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 you think that 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.

13.2 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!

Kurento API

Table of Contents

    - *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*.

14.1 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* contains a different *HubPort* for each one of the Media Elements that are connected. Depending on the Hub type, there are different ways to control the media. For example, there is a Hub called *Composite* that merges all input video streams in a unique output video stream, with all inputs arranged in a grid.

- **Output Endpoints**: Media Elements capable of taking a media stream out of the Media 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 a source element is fed into one or more sink elements. Hence, the pipeline represents a "pipe" capable of performing a sequence of operations over a stream.
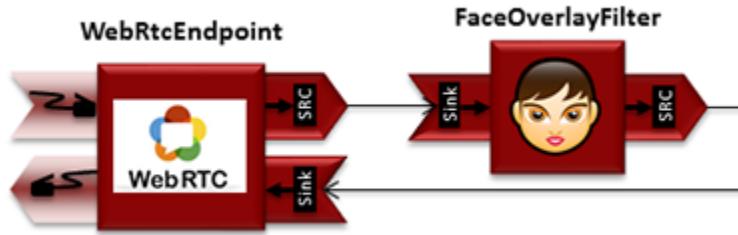
Fig. 1: *Example of a Media Pipeline implementing an interactive multimedia application receiving media from a WebRtcEndpoint, overlaying an image on the detected faces and sending back the resulting stream*

The Kurento API is . This means that it is based on Classes that can be instantiated in the form of Objects; these Objects provide *properties* that are a representation of the internal state of the Kurento server, and *methods* that expose the operations that can be performed by the server.

The following class diagram shows some of the relationships of the main classes in the Kurento API:

Fig. 2: Class diagram of main classes in Kurento API

14.2 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.

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:

14.3 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 *Code-FoundEvent*. 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.
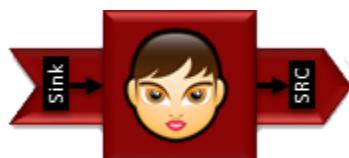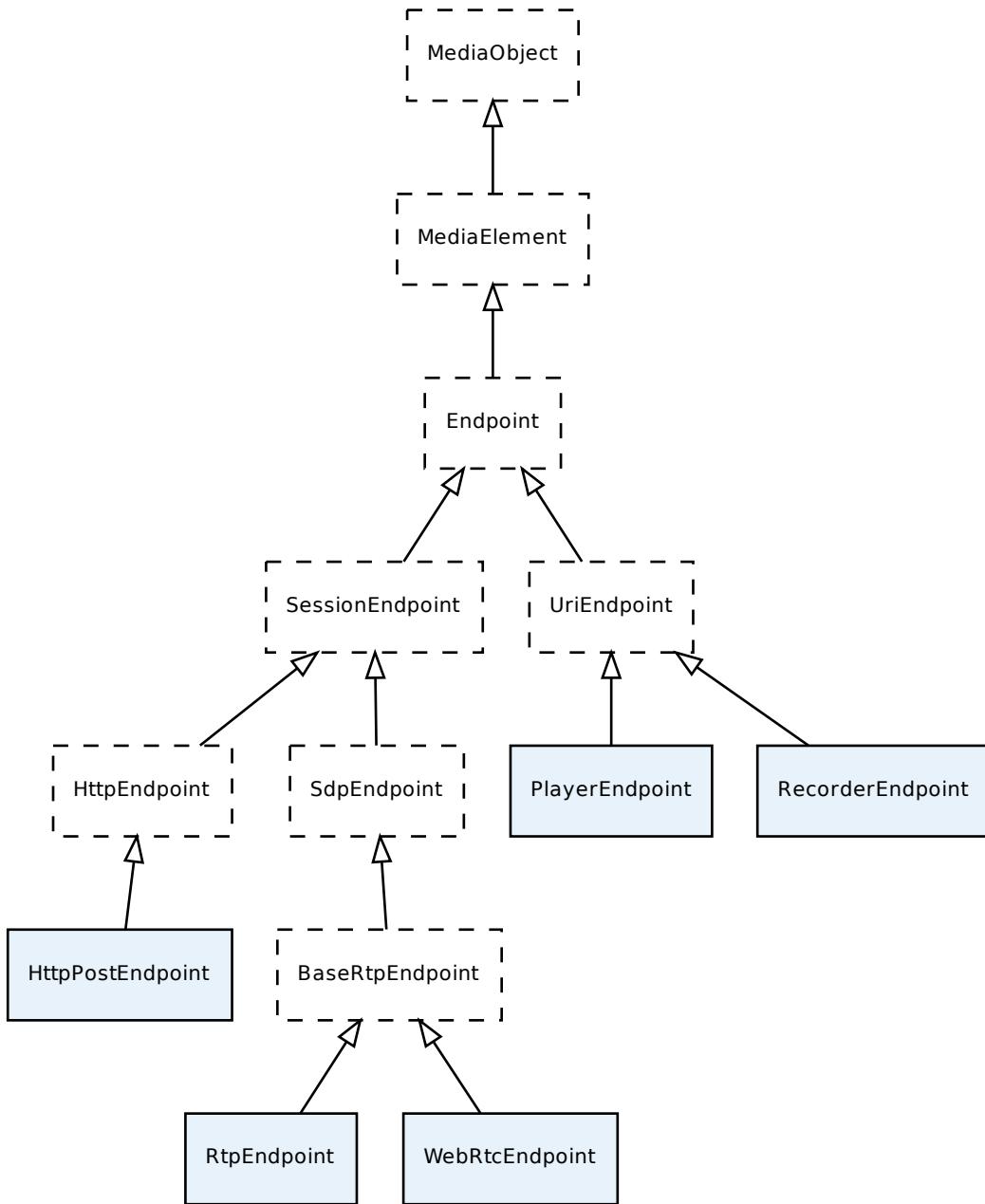
Fig. 3: Class diagram of Kurento Endpoints. In blue, the classes that a final API client will actually use.

GStreamerFilter is a generic filter interface that allows injecting any GStreamer element into a Kurento Media Pipeline. Note however that the current implementation of GStreamerFilter only allows single elements to be injected; one cannot indicate more than one at the same time; use several GStreamerFilters if you need to inject more than one element at the same time.

The following class diagram shows the relationships of the main filter classes:
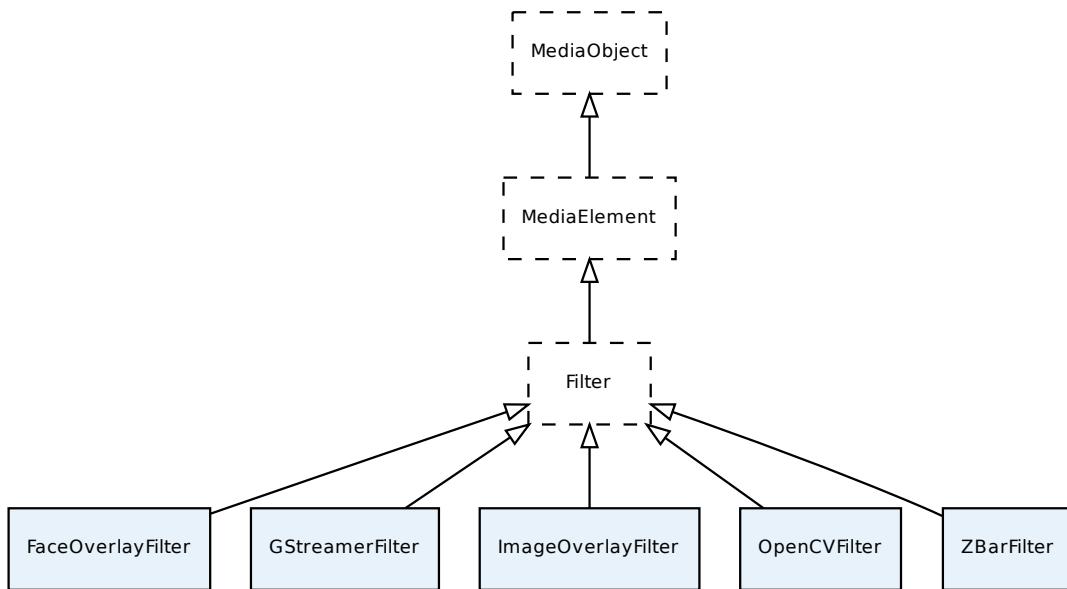
Fig. 4: Class diagram of Kurento Filters. In blue, the classes that a final API client will actually use.

14.4 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:
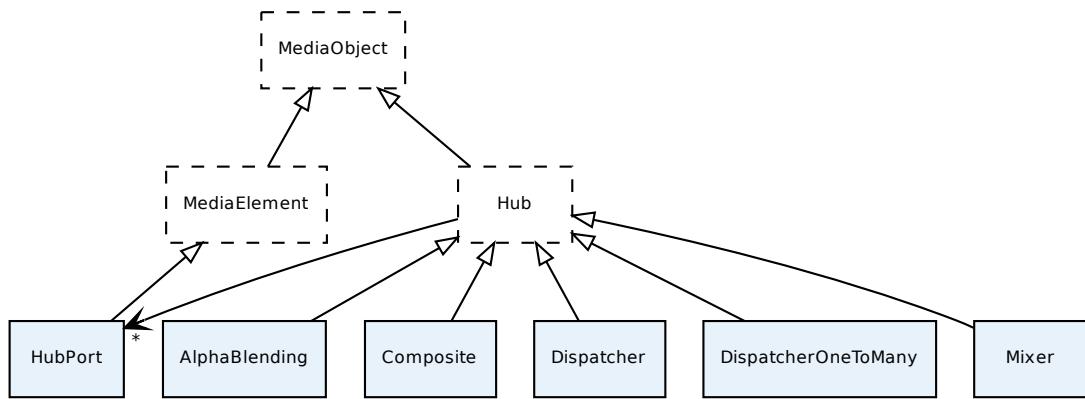
Fig. 5: Class diagram of Kurento Hubs. In blue, the classes that a final API client will actually use.

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.

15.1 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.

15.2 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.

15.3 Reference Documentation

- Kurento Client JavaDoc
- Kurento Client JsDoc

- Kurento Js Utils: a JavaScript utility library aimed to simplify the development of WebRTC applications.

Kurento Protocol

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*.

16.1 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.

16.1.1 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"
  }
}
```

16.1.2 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"
  }
}
```

16.1.3 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"
  }
}
```

16.2 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 releases 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.

16.2.1 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 *interval* parameter is the time available to receive the *pong* message from the server, in milliseconds. By default this value is 240000 (**4 minutes**).

This is an example of a *ping* request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "ping",
  "params": {
    "interval": 240000
  }
}
```

The response to a *ping* request must contain a *result* object with the parameter `value:  pong`. The following snippet shows the *pong* response to the previous *ping* request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "value": "pong"
  }
}
```

16.2.2 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*:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "create",
  "params": {
    "type": "MediaPipeline",
    "constructorParams": {},
    "properties": {}
  }
}
```

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:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "value": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

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):

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "create",
  "params": {
    "type": "WebRtcEndpoint",
    "constructorParams": {
      "mediaPipeline": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline"
    },
    "properties": {},
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

The response to this request message is as follows:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "value": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/087b7777-
→aab5-4787-816f-f0de19e5b1d9_kurento.WebRtcEndpoint",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

16.2.3 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*:

```
{
  "jsonrpc": "2.0",
  "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"
  }
}
```

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):

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

16.2.4 Release

This message requests releasing the resources of the specified object. The parameter *object* indicates the *id* of the object to be released:

```
{
  "jsonrpc": "2.0",
  "id": 36,
  "method": "release",
  "params": {
    "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

The response message only contains the *sessionId*:

```
{
  "jsonrpc": "2.0",
  "id": 36,
  "result": {
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

16.2.5 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:

```
{
  "jsonrpc": "2.0",
  "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"
  }
}
```

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:

```
{
  "jsonrpc": "2.0",
  "id": 11,
  "result": {
    "value": "052061c1-0d87-4fbd-9cc9-66b57c3e1280",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

16.2.6 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*:

```
{
  "jsonrpc": "2.0",
  "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"
  }
}
```

The response message only contains the *sessionId*:

```
{
  "jsonrpc": "2.0",
  "id": 38,
  "result": {
```

```
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    }
}
```

16.2.7 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**: [**DEPRECATED**: Use timestampMillis] The timestamp associated with this event: Seconds elapsed since the UNIX Epoch (Jan 1, 1970, UTC).

- **timestampMillis**: The timestamp associated with this event: Milliseconds elapsed since the UNIX Epoch (Jan 1, 1970, UTC).

- **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",
        "timestampMillis": "1441277150433",
        "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.

16.3 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 a 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"
}
```

16.4 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:

```json
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "create",
  "params": {
    "type": "MediaPipeline",
    "constructorParams": {},
    "properties": {}
  }
}
```

2. KMS sends a response message with the identifier for the Media Pipeline and the Media Session:

```json
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "value": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline",
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  }
}
```

3. Client sends a request to create a *WebRtcEndpoint*:

```json
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "create",
  "params": {
    "type": "WebRtcEndpoint",
    "constructorParams": {
      "mediaPipeline": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline"
    },
    "properties": {},
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  }
}
```

4. KMS creates the *WebRtcEndpoint* and sends back to the client the Media Element identifier:

```json
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "value": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-48ff-
→99ef-02f7fadabaf7_WebRtcEndpoint",
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  }
}
```

5. Client invokes the *connect* primitive in the *WebRtcEndpoint* in order to create a loopback:

```json
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "invoke",
```

```json
  "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"
  }
}
```

6. KMS carries out the connection and acknowledges the operation:

```json
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  }
}
```

7. Client invokes the *processOffer* primitive in the *WebRtcEndpoint* in order to start the *SDP Offer/Answer* negotiation for WebRTC:

```json
{
  "jsonrpc": "2.0",
  "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"
  }
}
```

8. KMS carries out the SDP negotiation and returns the SDP Answer:

```json
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "value": "SDP"
  }
}
```

16.5 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 at IDL files that define 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

16.5.1 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 also be 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 update && sudo apt-get install --yes 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.

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:
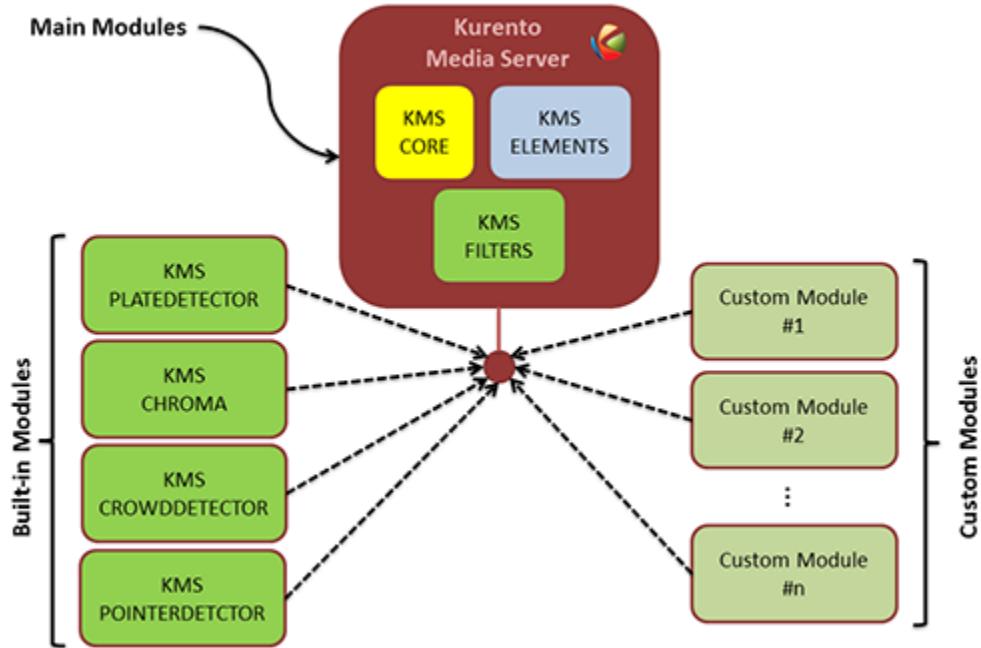
Fig. 1: **Kurento modules architecture** *Kurento Media Server can be extended with built-in modules (crowddetector, pointerdetector, chroma, platedetector) and also with other custom modules.*

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.

17.1 Module Tutorial - Pointer Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a pointer-tracking filter element.

17.1.1 Java Module - Pointer Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a pointer tracking filter element.

> **Note:** Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.
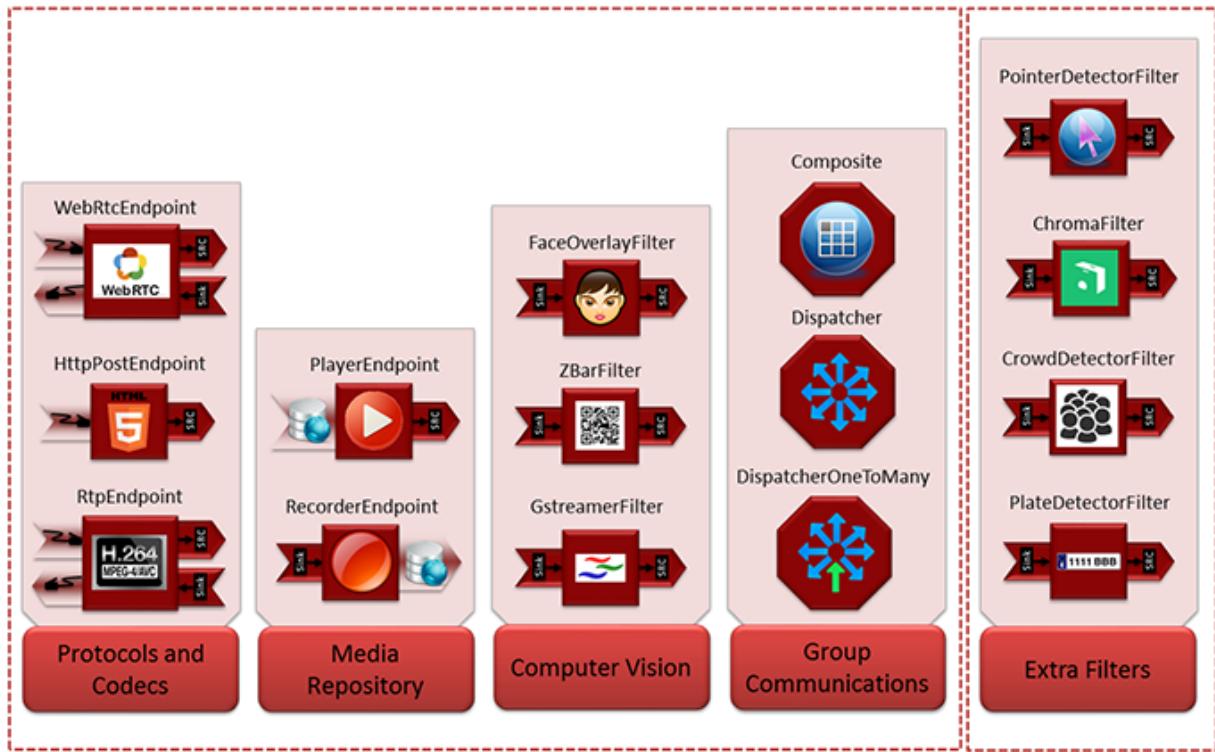
Fig. 2: **Extended Kurento Toolbox** *The basic Kurento toolbox (left side of the picture) is extended with more Computer Vision and Augmented Reality filters (right side of the picture) provided by the built-in modules.*

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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. 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 thorugh the filter, as follows:
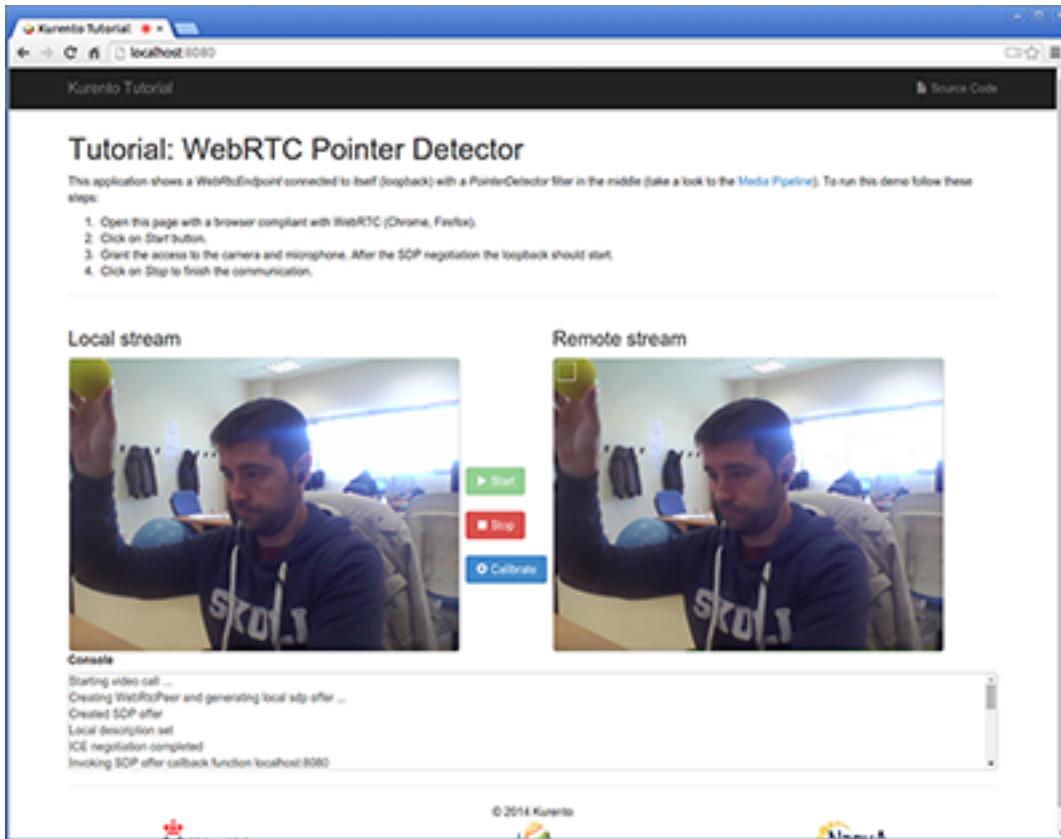
Fig. 4: *Pointer calibration stage*

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>() {
```

(continues on next page)

```java
    @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 {
```

```java
            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:
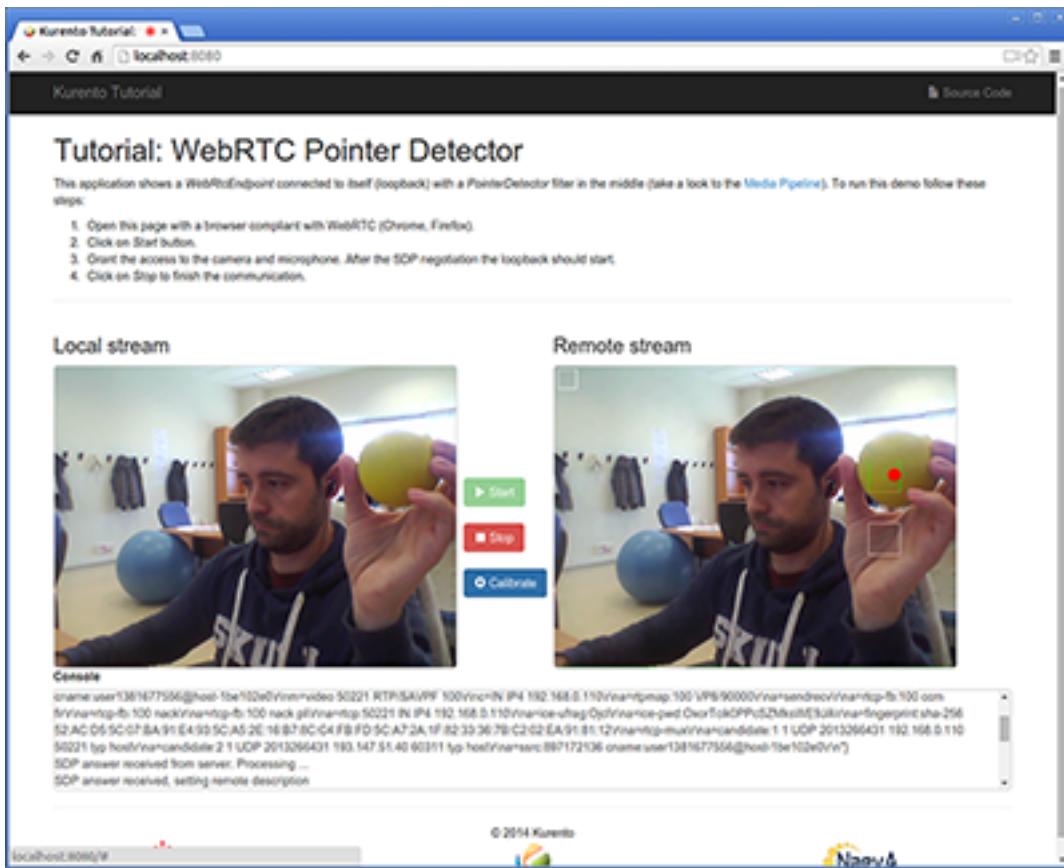
Fig. 5: *Pointer tracking over a window*

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();
   }
}
```

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>
```

(continues on next page)

```
    </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>
```

17.1.2 JavaScript Module - Pointer Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a pointer tracking filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-pointerdetector/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

  ```
  https://localhost:8443/
  ```

- If KMS is running in a remote machine:

```
https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

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. 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:

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.

Fig. 7: *Pointer calibration stage*

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:

```javascript
...
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,
```

(continues on next page)

```
        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:

```
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"}]
```
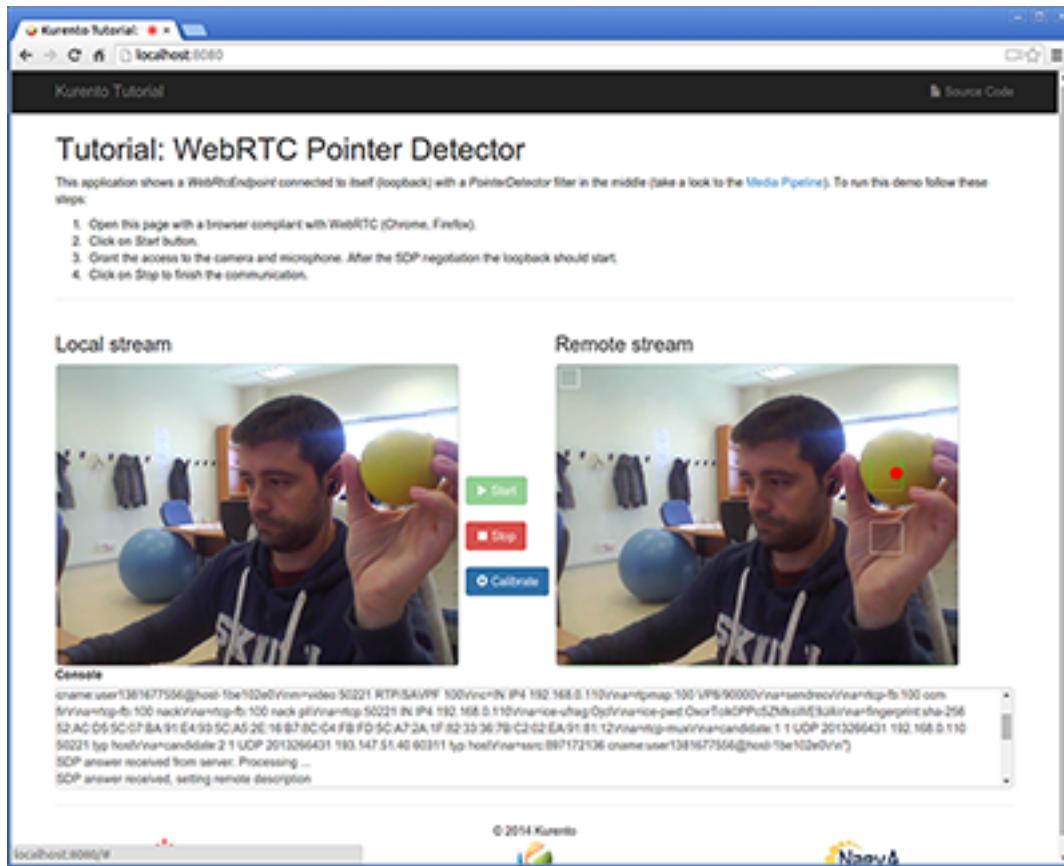
Fig. 8: *Pointer tracking over a window*

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": "6.15.0",
   "kurento-utils": "6.15.0"
   "kurento-module-pointerdetector": "6.15.0"
}
```

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.

17.1.3 Node.js Module - Pointer Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a pointer tracking filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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. 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.

In that precise moment, a calibration operation should be carried out. This is done by clicking on the *Calibrate* blue button of the GUI.

Fig. 10: *Pointer calibration stage*

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:

```javascript
...
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
                        }));
                    });

                    filter.on('WindowIn', function (_data) {
```

(continues on next page)

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

```javascript
    });
}

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:

```javascript
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:

```json
"dependencies": {
   "kurento-client" : "6.15.0"
}
```

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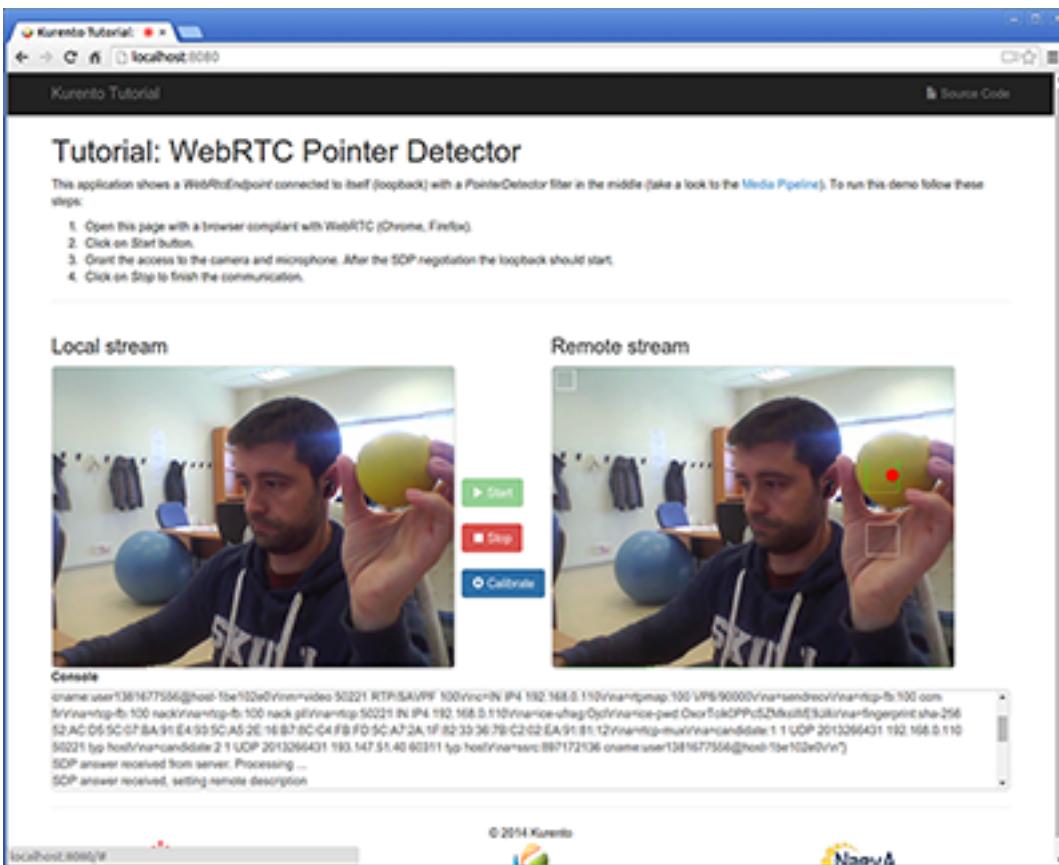

Fig. 11: *Pointer tracking over a window*

```
"dependencies": {
    "kurento-utils" : "6.15.0",
    "kurento-module-pointerdetector": "6.15.0"
}
```

Note: We are in active development. You can find the latest versions at npm and Bower.

17.2 Module Tutorial - Chroma Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

17.2.1 Java Module - Chroma Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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. 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:

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:

The media pipeline of this demo is is implemented in the server-side logic as follows:

```java
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 {
```

(continues on next page)
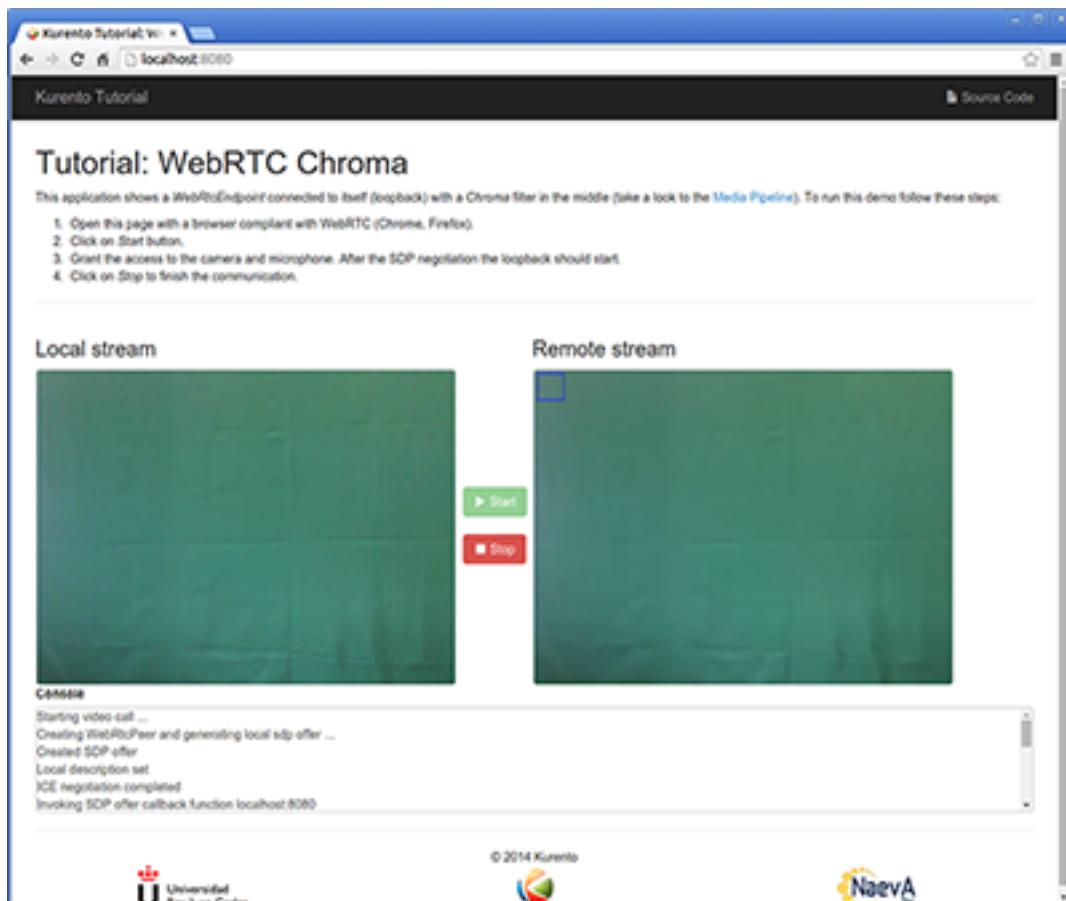
Fig. 13: *Chroma calibration stage*
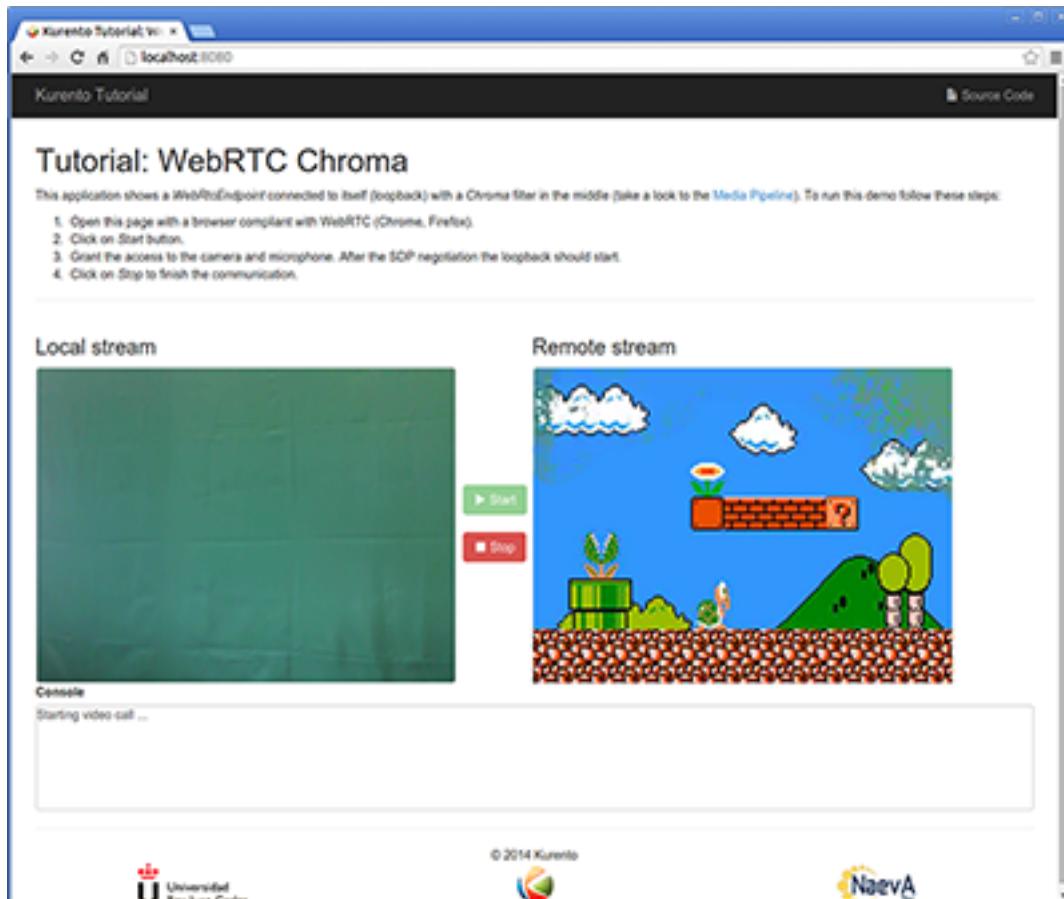
Fig. 14: *Chroma filter in action*

```java
                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:

```xml
<dependencies>
   <dependency>
      <groupId>org.kurento</groupId>
      <artifactId>kurento-client</artifactId>
   </dependency>
   <dependency>
      <groupId>org.kurento</groupId>
      <artifactId>kurento-utils-js</artifactId>
   </dependency>
   <dependency>
```

```xml
        <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:

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

17.2.2 JavaScript Module - Chroma Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-chroma/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

```
https://localhost:8443/
```

- If KMS is running in a remote machine:

```
https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

Note: By default, this tutorial assumes that Kurento Media Server can download the overlay image from a `localhost` web server. *It will fail if the web server is remote* (from the point of view of KMS). This includes the case of running KMS from Docker.

If you want to run this tutorial with a **remote Kurento Media Server** (including running KMS from Docker), then you have to provide it with the correct IP address of the application's web server:

- In *index.js*, change `bg_uri` to the correct one where KMS can reach the web server.

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:
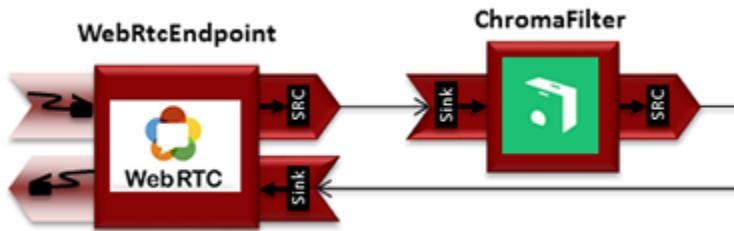
Fig. 15: *WebRTC with Chroma filter Media Pipeline*

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:
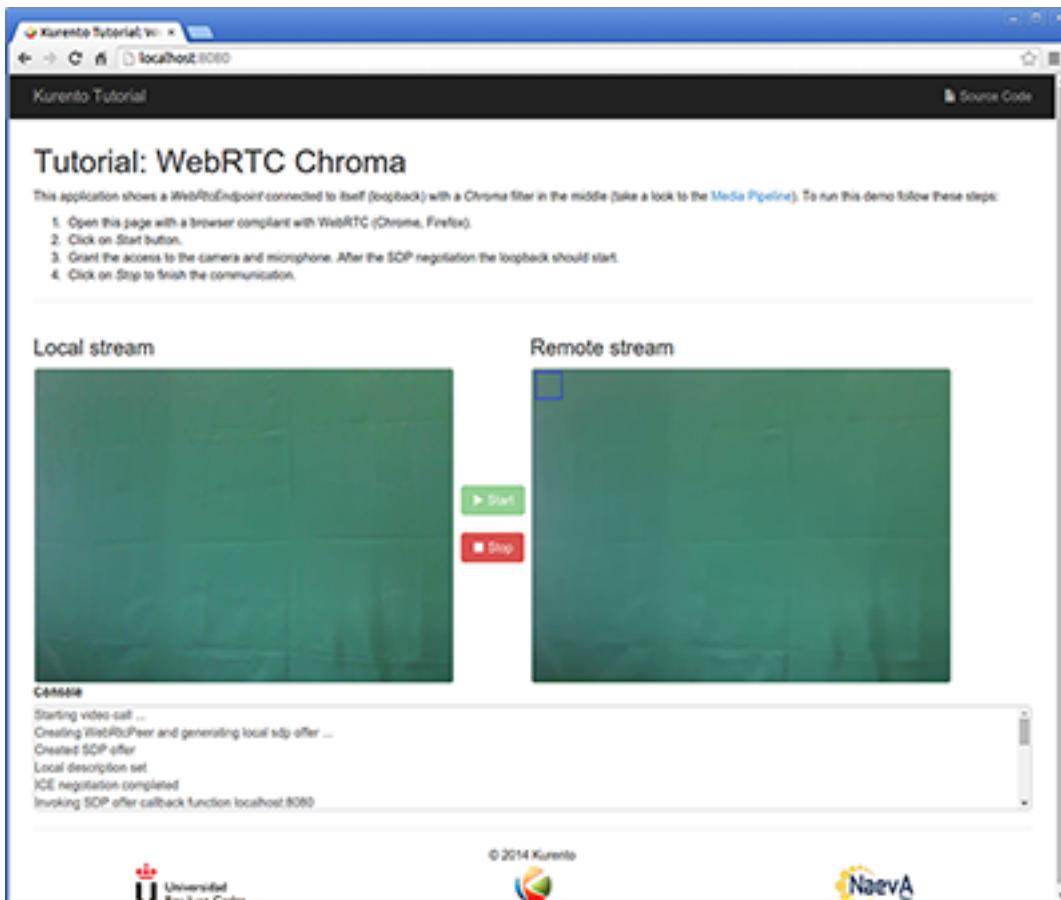
Fig. 16: *Chroma calibration stage*

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:
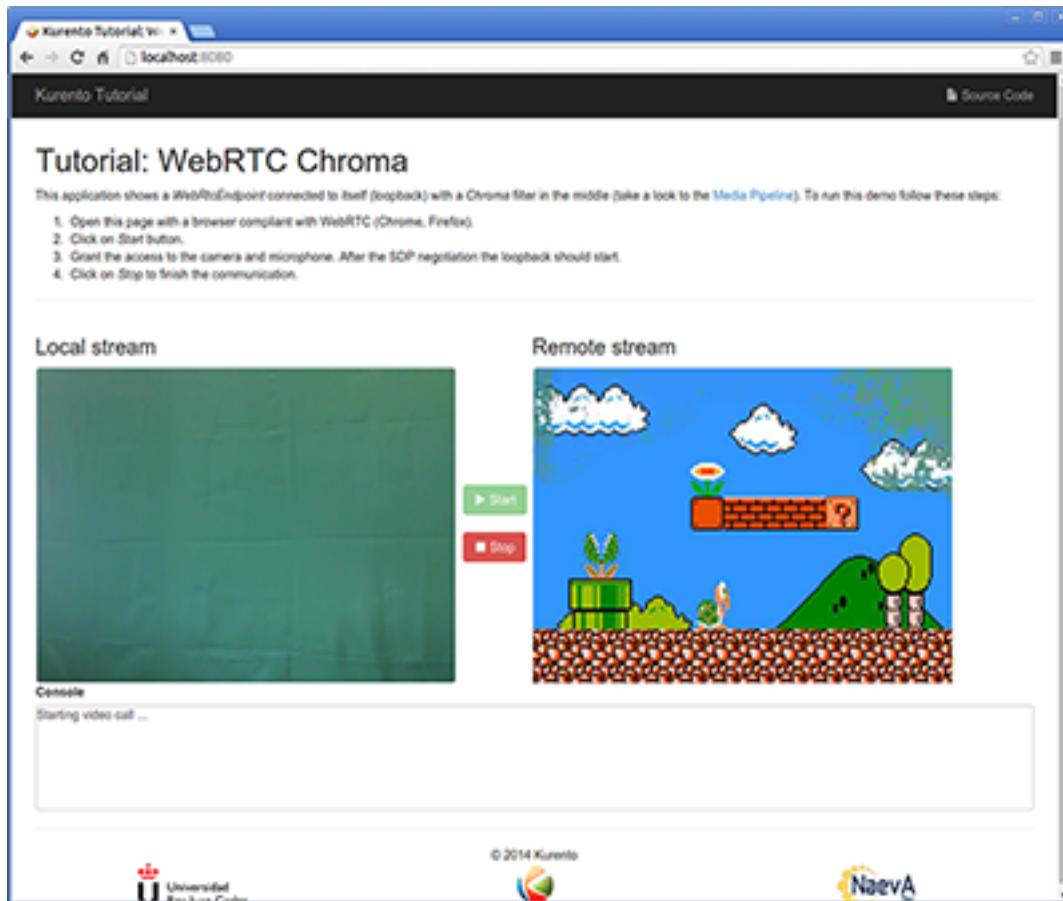
Fig. 17: *Chroma filter in action*

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 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);

  client.create('MediaPipeline', function(error, _pipeline) {
    if (error) return onError(error);

    pipeline = _pipeline;
```

(continues on next page)

```javascript
    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": "6.15.0",
   "kurento-utils": "6.15.0"
   "kurento-module-pointerdetector": "6.15.0"
}
```

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.

17.2.3 Node.js Module - Chroma Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a chroma filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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. 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:
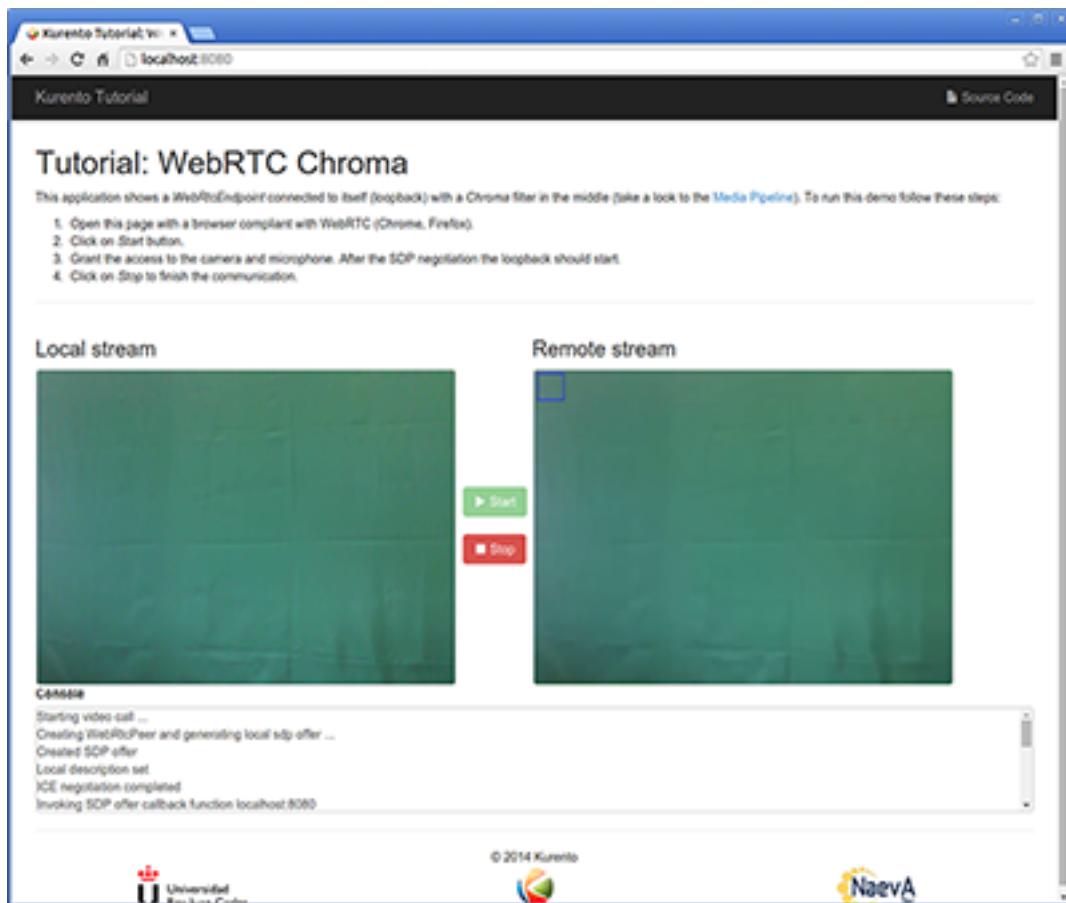
Fig. 19: *Chroma calibration stage*

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:
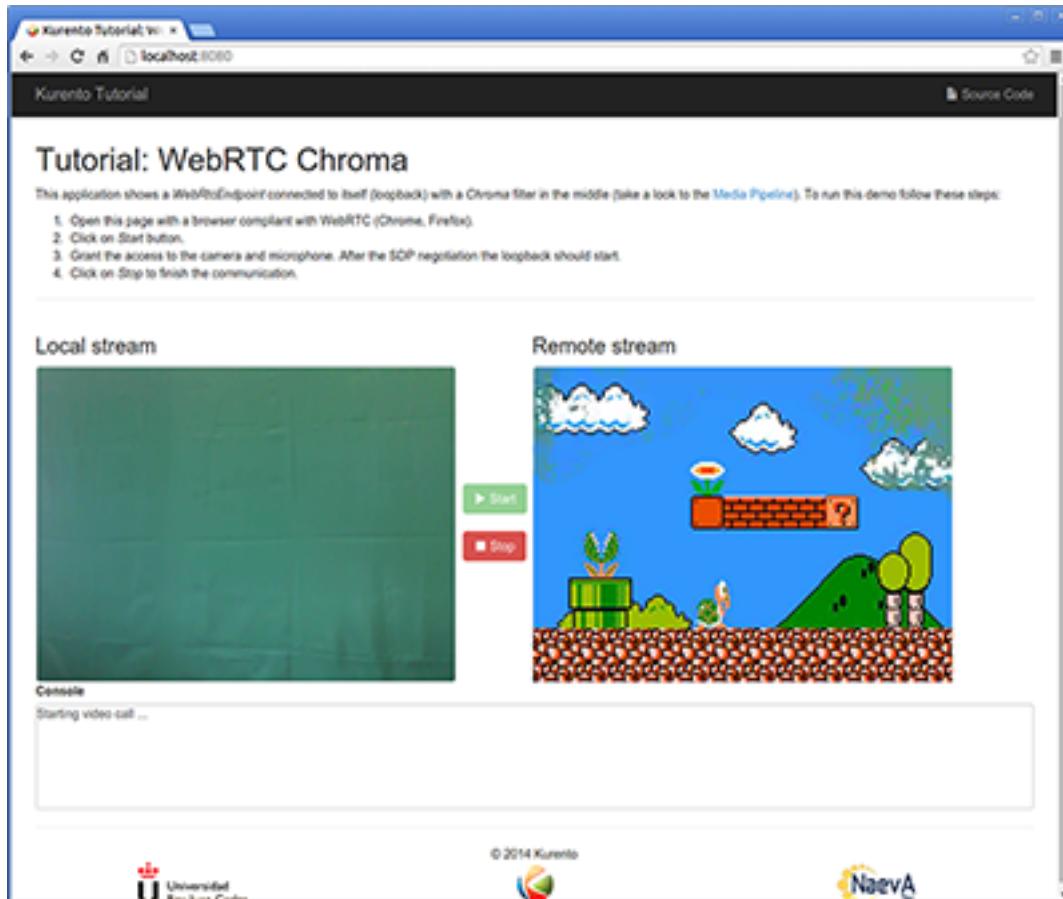
Fig. 20: *Chroma filter in action*

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 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) {
```

(continues on next page)

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

```javascript
        });
    });
}

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" : "6.15.0"
}
```

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" : "6.15.0",
    "kurento-module-pointerdetector": "6.15.0"
}
```

Note: We are in active development. You can find the latest versions at npm and Bower.

17.3 Module Tutorial - Crowd Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects people agglomeration in video streams.

17.3.1 Java Module - Crowd Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects clusters of people in video streams.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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. 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.

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.

Fig. 22: *Crowd detection sample*

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 (0-360º), 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 (0º=north, 90º=east, 180º=south, 270º=west).

All in all, the media pipeline of this demo is implemented as follows:

```java
// 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()));
```

(continues on next page)

```java
        }
      } 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());
    }
  }
  });
```

```java
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:

```xml
<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:

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

17.3.2 JavaScript Module - Crowd Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects people agglomeration in video streams.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-crowddetector/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

  ```
  https://localhost:8443/
  ```

- If KMS is running in a remote machine:

  ```
  https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
  ```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

Note: By default, this tutorial assumes that Kurento Media Server can download the overlay image from a `localhost` web server. *It will fail if the web server is remote* (from the point of view of KMS). This includes the case of running KMS from Docker.

If you want to run this tutorial with a **remote Kurento Media Server** (including running KMS from Docker), then you have to provide it with the correct IP address of the application's web server:

- In *index.js*, change `logo_uri` to the correct one where KMS can reach the web server.

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. 23: *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).

Regarding direction, it is quantified as an angle (0-360º), 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.

Fig. 24: *Crowd detection sample*

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 (0º=north, 90º=east, 180º=south, 270º=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 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;
```

(continues on next page)

```javascript
    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...");
```

```javascript
      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": "6.15.0",
   "kurento-utils": "6.15.0"
   "kurento-module-pointerdetector": "6.15.0"
}
```

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.

17.3.3 Node.js Module - Crowd Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a crowd detector filter. This filter detects people agglomeration in video streams.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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. 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).

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.

Fig. 26: *Crowd detection sample*

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 (0º=north, 90º=east, 180º=south, 270º=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 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);
```

(continues on next page)

```javascript
        }

      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
```

```javascript
                }
                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" : "6.15.0"
}
```

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" : "6.15.0",
    "kurento-module-pointerdetector": "6.15.0"
}
```

Note: We are in active development. You can find the latest versions at npm and Bower.

17.4 Module Tutorial - Plate Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

17.4.1 Java Module - Plate Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Java server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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 6.15.0
mvn -U clean spring-boot:run
```

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 -U clean spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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. 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.

Fig. 28: *Plate detector demo in action*

```java
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());
                  }
               }
            });

      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);
```

(continues on next page)

```java
        // 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:

```xml
<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>
```

17.4.2 JavaScript Module - Plate Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure JavaScript applications to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

Running this example

First of all, install Kurento Media Server: *Installation Guide*. Start the media server and leave it running in the background.

Install *Node.js*, *Bower*, and a web server in your system:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
sudo npm install -g http-server
```

Here, we suggest using the simple Node.js `http-server`, but you could use any other web server.

You also need the source code of this tutorial. Clone it from GitHub, then start the web server:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-platedetector/
git checkout 6.15.0
bower install
http-server -p 8443 --ssl --cert keys/server.crt --key keys/server.key
```

When your web server is up and running, use a WebRTC compatible browser (Firefox, Chrome) to open the tutorial page:

- If KMS is running in your local machine:

  ```
  https://localhost:8443/
  ```

- If KMS is running in a remote machine:

```
https://localhost:8443/index.html?ws_uri=ws://{KMS_HOST}:8888/kurento
```

Note: By default, this tutorial works out of the box by using non-secure WebSocket (`ws://`) to establish a client connection between the browser and KMS. This only works for `localhost`. *It will fail if the web server is remote.*

If you want to run this tutorial from a **remote web server**, then you have to do 3 things:

1. Configure **Secure WebSocket** in KMS. For instructions, check *Signaling Plane security (WebSocket)*.

2. In *index.js*, change the `ws_uri` to use Secure WebSocket (`wss://` instead of `ws://`) and the correct KMS port (TCP 8433 instead of TCP 8888).

3. As explained in the link from step 1, if you configured KMS to use Secure WebSocket with a self-signed certificate you now have to browse to `https://{KMS_HOST}:8433/kurento` and click to accept the untrusted certificate.

Note: This demo uses the **kms-platedetector** module, which must be installed in the media server. That module is available in the Kurento Apt repositories, so it is possible to install it with this command:

```
sudo apt-get update && sudo apt-get install --yes kms-platedetector
```

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:
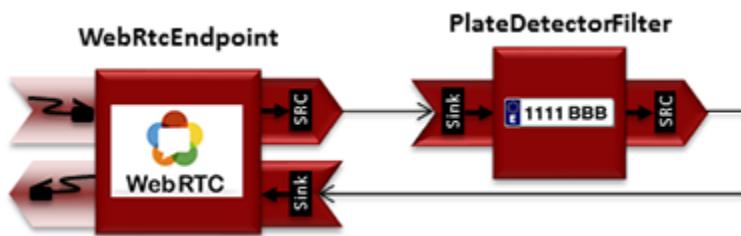
Fig. 29: *WebRTC with plateDetector filter Media Pipeline*

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.
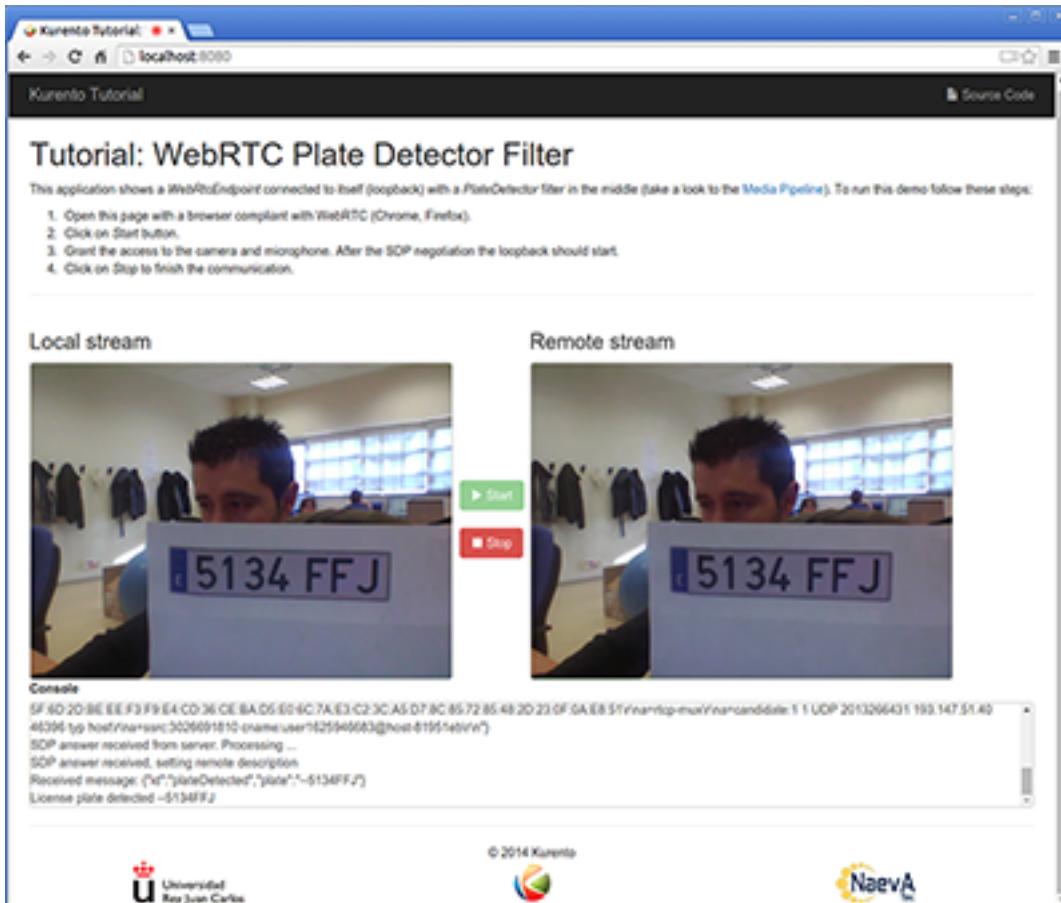
Fig. 30: *Plate detector demo in action*

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");

    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": "6.15.0",
    "kurento-utils": "6.15.0"
    "kurento-module-pointerdetector": "6.15.0"
}
```

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.

17.4.3 Node.js Module - Plate Detector Filter

This web application consists of a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

Note: Web browsers require using *HTTPS* to enable WebRTC, so the web server must use SSL and a certificate file. For instructions, check *Configure a Node server to use HTTPS*.

For convenience, this tutorial already provides dummy self-signed certificates (which will cause a security warning in the browser).

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_8.x | sudo -E 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 6.15.0
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}:8888/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. 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:
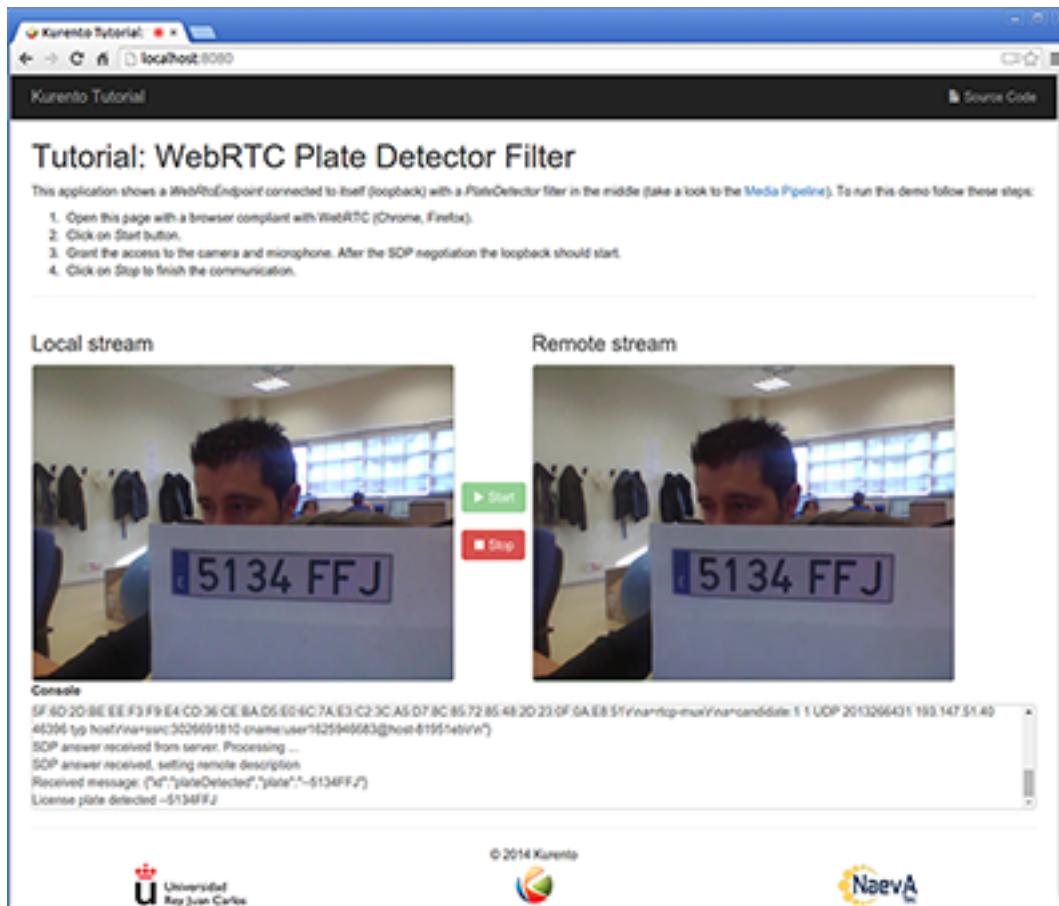
Fig. 32: *Plate detector demo in action*

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 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) {
```

```javascript
        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
                        }));
                    });

                    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" : "6.15.0"
}
```

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" : "6.15.0",
    "kurento-module-pointerdetector": "6.15.0"
}
```

Note: We are in active development. You can find the latest versions at npm and Bower.

Kurento Utils JS

[TODO full review]

18.1 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.

18.2 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>
```

18.3 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 RTCPeer-Connection 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.

```javascript
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 one 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.

```javascript
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

```javascript
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

```javascript
webRtcPeer.addIceCandidate(candidate);
```

Following the previous steps, we have:

- Sent an 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.

18.4 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.

```javascript
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.

18.5 Reference documentation

18.5.1 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
    - *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 onviked 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 appropriate 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.

18.5.2 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.

18.6 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_8.x | sudo -E 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
```

18.7 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
```

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

19.1 MediaObject events

This is the base interface used to manage capabilities common to all Kurento elements, including both *MediaElement* and *MediaPipeline*.

19.1.1 Error

Some error has occurred. Check the event parameters (such as *description*, *errorCode*, and *type*) to get information about what happened.

19.2 MediaElement events

These events indicate some low level information about the state of GStreamer, the underlying multimedia framework.

19.2.1 ElementConnected

[TODO - add contents]

19.2.2 ElementDisconnected

[TODO - add contents]

19.2.3 MediaFlowInStateChange

• State = *Flowing*: Data is arriving from the KMS Pipeline, and **flowing into** the Element. Technically, this means that there are GStreamer Buffers flowing from the Pipeline to the Element'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.

19.2.4 MediaFlowOutStateChange

• State = *Flowing*: There is data **flowing out** from the Element towards the KMS Pipeline. Technically, this means that there are GStreamer Buffers flowing from the Element'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.

19.2.5 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 **automatically done** in order to match the incoming and outgoing media formats. 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 don't match, the internal transcoding module will get enabled to convert between them.

For example: If a *WebRtcEndpoint* receives a *VP8* video stream from a Chrome browser, and has to send it to a Safari browser (which only supports the *H.264* codec), then the media needs to be transcoded. The *WebRtcEndpoint* will automatically do it.

- State = *Transcoding*: The *MediaElement* will transcode the incoming media, because its format is not compatible with the requested output.

- State = *NotTranscoding*: The *MediaElement* will *not* transcode the incoming media, because its format is compatible with the requested output.

19.3 BaseRtpEndpoint events

These events provide information about the state of the RTP connection for each stream in the WebRTC call.

Note that the *MediaStateChanged* event is not 100% reliable to check if a RTP connection is active: RTCP packets do not usually flow at a constant rate. For example, minimizing a browser window with an *RTCPeerConnection* might affect this interval.

19.3.1 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
```

19.3.2 MediaStateChanged

- State = *Connected*: At least *one* of the audio or video RTP streams in the session is still alive (sending or receiving RTCP packets). Equivalent to the signal GstRtpBin::"on-ssrc-active", which gets triggered whenever the GstRtpBin receives an *RTCP Sender Report* (*RTCP SR*) or *RTCP Receiver Report* (*RTCP RR*).

- State = *Disconnected*: None of the RTP streams belonging to the session is alive (ie. no RTCP packets are sent or 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*).

19.4 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.

19.4.1 DataChannelClose

[TODO - add contents]

19.4.2 DataChannelOpen

[TODO - add contents]

19.4.3 IceCandidateFound

A new local candidate has been found, after the ICE Gathering process was started. Equivalent to the signal NiceAgent::"new-candidate-full".

19.4.4 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 idle.

  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. These checks will always start as soon as possible (i.e. whenever the very first remote candidates arrive), so don't assume that the candidate gathering has already finished, because it will probably still be running in parallel; some (possibly better) candidates might still be waiting to be found and gathered.

  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*. When this event triggers, the effective communication between peers can start, and usually this means that media will start flowing between them. However, the candidate gathering hasn't really finished yet, which means that some (possibly better) candidates might still be waiting to be found, gathered, checked for connectivity, and if that completes successfully, selected as new candidate pair.

  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 (source):

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 *STUN*/*TURN* server will cause the state to go back, *Ready* and *Failed* are **not** terminal states.

19.4.5 IceGatheringDone

All local candidates have been found, so the gathering process is finished for this peer. Note this doesn't imply that the remote peer has finished its own gathering, so more remote candidates might still arrive. Equivalent to the signal NiceAgent::"candidate-gathering-done".

19.4.6 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".

19.5 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.
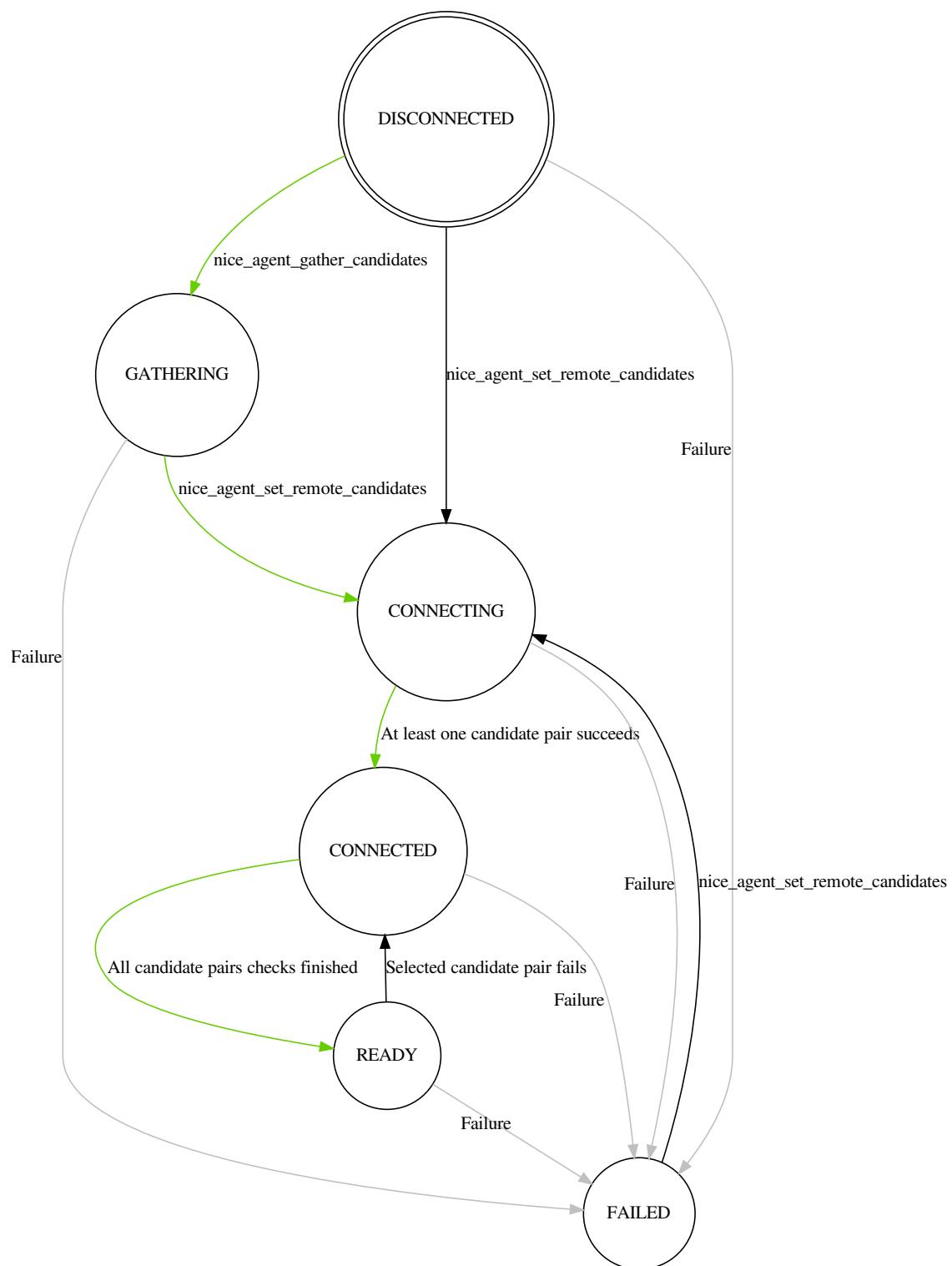

Fig. 1: libnice state transition diagram for NiceComponentState

- 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:

```
IceCandidateFound
IceComponentStateChanged (Gathering)
AddIceCandidate
IceComponentStateChanged (Connecting)
AddIceCandidate
IceCandidateFound
NewCandidatePairSelected
IceComponentStateChanged (Connected)
NewCandidatePairSelected
IceGatheringDone
IceComponentStateChanged: (Ready)
```

1. *IceCandidateFound*

   Repeated multiple times; tipically, 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. *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. *AddIceCandidate*

   Repeated multiple times; 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. *IceComponentStateChanged* (state: *Connecting*)

   After receiving the very first of the remote candidates, the ICE Agent starts with the connectivity checks.

5. *AddIceCandidate*

   Repeated multiple times; the remote peer will continue sending its own gathered candidates, of any type: *host*, *srflx* (*STUN*), *relay* (*TURN*).

6. *IceCandidateFound*

   Repeated multiple times; 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.

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*.

20.1 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.

20.2 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.

20.2.1 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 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 2: SDP Answer

```
v=0
o=- 3696336115 3696336115 IN IP4 198.51.100.1
s=Kurento Media Server
c=IN IP4 198.51.100.1
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 *198.51.100.1*; 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.

Securing Kurento Applications

21.1 Securing Application Servers

21.1.1 Configure a Java server to use HTTPS

- Obtain a certificate. For this, either request one from a trusted Certification Authority (*CA*), or generate your own one as explained here: *Generating a self-signed certificate*.

- Convert your PEM certificate to either Java KeyStore (*JKS*) or PKCS#12. The former is a proprietary format limited to the Java ecosystem, while the latter is an industry-wide used format. To make a PKCS#12 file from an already existing PEM certificate, run these commands:

```
openssl pkcs12 \
    -export \
    -in cert.pem -inkey key.pem \
    -out cert.p12 -passout pass:123456

chmod 440 *.p12
```

- Use the certificate in your application.

  Place your PKCS#12 file *cert.p12* in `src/main/resources/`, and add this to the *application.properties* file:

```
server.ssl.key-store=classpath:cert.p12
server.ssl.key-store-password=123456
server.ssl.key-store-type=PKCS12
```

- Start the Spring Boot application:

```
mvn spring-boot:run \
    -Dspring-boot.run.jvmArguments="-Dkms.url=ws://{KMS_HOST}:8888/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.

21.1.2 Configure a Node server to use HTTPS

- Obtain a certificate. For this, either request one from a trusted Certification Authority (*CA*), or generate your own one as explained here: *Generating a self-signed certificate*.

- Add the following changes to your *server.js*, in order to enable HTTPS:

```javascript
...
var express = require('express');
var ws      = require('ws');
var fs       = require('fs');
var https   = require('https');
...

var options =
{
  cert: fs.readFileSync('cert.pem'),
  key:  fs.readFileSync('key.pem'),
};

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
```

21.1.3 Configure JavaScript applications to use HTTPS

WebRTC requires HTTPS, so your JavaScript application must be served by a secure web server. You can use whichever one you prefer, such as Nginx or Apache. For quick tests, a very straightforward option is to use the simple, zero-configuration http-server based on Node.js:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install --yes nodejs
sudo npm install -g http-server
```

- Obtain a certificate. For this, either request one from a trusted Certification Authority (*CA*), or generate your own one as explained here: *Generating a self-signed certificate*.

- Start the HTTPS web server, using the SSL certificate:

```
http-server -p 8443 --ssl --cert cert.pem --key key.pem
```

21.2 Securing Kurento Media Server

21.2.1 Signaling Plane authorization

You should protect the JSON-RPC API control port (WebSocket port, by default TCP 8888) of your Kurento Media Server instances from unauthorized access from public networks.

The Kurento WebSocket server supports using SSL certificates in order to guarantee secure communications between clients and server; however, at the time no authentication mechanism is provided. Kurento doesn't reinvent the wheel here including its own mechanism, and instead it relies on layers of security that already exist at the system level. This is something we may add (contributions are welcomed!) but for now here are some tips on how other big players are protecting KMS from unauthorized use.

Think of KMS like you would of a database in a traditional web application; there are two levels:

1. The **application level**. We usually call this the "*Application Server*" of Kurento Media Server. It usually is a web application that uses *Kurento Client* to access *Kurento API*.

2. The **media level** (actual audio/video transmissions to/from KMS).

The idea is that nobody unauthorized should be able to access the exchanged media. At the application level we can use all the available techniques used to protect any web server, for example with a custom user/password mechanism. Regarding KMS, the idea is that only the *Application Server* can access KMS. We can restrict that at the system level, for example using iptables to restrict all incoming WebSocket connections to KMS only from a given host, or a given subnet, similar to this: Iptables Essentials: Common Firewall Rules and Commands (archive). It may be a good idea to have the *Application Server* running in the same host than the Media Server, and in that case just restrict incoming connections to the same host.

If you need more flexibility, one idea is to restrict KMS connections to the same host using iptables and then implement a WebSocket proxy in the same machine (e.g. using nginx) that has its resources secured, as in NGINX as a WebSocket Proxy (archive) or WebSocket proxying (archive); this way, the *Application Server* connects to the WebSocket proxy that can indeed be secured, and thus only authenticated users from remote hosts can gain access to KMS.

21.2.2 Signaling Plane security (WebSocket)

With the default configuration, Kurento Media Server will use the `ws://` URI scheme for non-secure WebSocket connections, listening on the port TCP 8888. Application Servers (Kurento clients) will establish a WebSocket connection with KMS, in order to control the media server and send messages conforming to the *Kurento API*.

This is fine for initial stages of application development, but before deploying on production environments you'll probably want to move to `wss://` connections, i.e. using Secure WebSocket, which by default uses the port TCP 8433.

To enable Secure WebSocket, edit the main KMS configuration file (`/etc/kurento/kurento.conf.json`), and un-comment the following lines:

```
"secure": {
  "port": 8433,
  "certificate": "cert+key.pem",
  "password": "KEY_PASSWORD"
}
```

If you use a signed certificate issued by a trusted Certification Authority (*CA*) such as Verisign or Let's Encrypt, then you are done. Just skip to the next section: *Connecting to Secure WebSocket*.

However, if you are going to use an untrusted self-signed certificate (typically during development), there is still more work to do.

Generate your own certificate as explained here: *Generating a self-signed certificate*. Now, because self-signed certificates are untrusted by nature, client browsers and server applications will reject it by default. You'll need to force all consumers of the certificate to accept it:

- **Java applications**. Follow the instructions of this link: SunCertPathBuilderException: unable to find valid certification path to requested target (archive).

  Get *InstallCert.java* from here: https://github.com/escline/InstallCert.

  You'll need to instruct the *KurentoClient* to allow using certificates. For this purpose, create an *JsonRpcClient*:

  ```
  SslContextFactory sec = new SslContextFactory(true);
  sec.setValidateCerts(false);
  JsonRpcClientWebSocket rpcClient = new JsonRpcClientWebSocket(uri, sec);
  KurentoClient kurentoClient = KurentoClient.createFromJsonRpcClient(rpcClient);
  ```

- **Node applications**. Take a look at this page: Painless Self Signed Certificates in node.js (archive).

  For a faster but *INSECURE* alternative, configure Node to accept (instead of reject) invalid TLS certificates by default, setting the environment variable flag NODE_TLS_REJECT_UNAUTHORIZED to *0*; this will disable the TLS validation for your whole Node app. You can set this environment variable before executing your app, or directly in your app code by adding the following line before performing the connection:

  ```
  process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0;
  ```

- **Browser JavaScript**. Similar to what happens with self-signed certificates used for HTTPS, browsers also require the user to accept a security warning before Secure WebSocket connections can be established. This is done by *directly opening* the KMS WebSocket URL: `https://{KMS_HOST}:8433/kurento`.

Connecting to Secure WebSocket

Now that KMS is listening for Secure WebSocket connections, and (if using a self-signed certificate) your Application Server is configured to accept the certificate used in KMS, you have to change the WebSocket URL used in your application logic.

Make sure your application uses a WebSocket URL that starts with `wss://` instead of `ws://`. Depending on the platform, this is done in different ways:

- **Java**: Launch with a *kms.url* property. For example:

  ```
  mvn spring-boot:run \
      -Dspring-boot.run.jvmArguments="-Dkms.url=wss://{KMS_HOST}:8433/kurento"
  ```

- **Node**: Launch with the *ws_uri* command-line argument. For example:

```
npm start -- --ws_uri="wss://{KMS_HOST}:8433/kurento"
```

- **Browser JavaScript**: Application-specific method. For example, using hardcoded values:

```
const ws_uri: "wss://" + location.hostname + ":8433/kurento";
```

21.2.3 Media Plane security (DTLS)

WebRTC uses for media data authentication. By default, if no certificate is provided for this, Kurento Media Server will auto-generate its own self-signed certificate for every WebRtcEndpoint instance, but it is also possible to provide an already existing certificate to be used for all endpoints.

To do so, edit the file `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini` and set either *pemCertificateRSA* or *pemCertificateECDSA* with a file containing the concatenation of your certificate (chain) file(s) and the private key.

Setting a custom certificate for DTLS is needed, for example, for situations where you have to manage multiple media servers and want to make sure that all of them use the same certificate for their connections. Some browsers, such as Firefox, require this in order to allow multiple WebRTC connections from the same tab to different KMS instances.

21.3 Generating a self-signed certificate

You need to provide a valid SSL certificate in order to enable all sorts of security features, ranging from HTTPS to Secure WebSocket (`wss://`). For this, there are two alternatives:

- Obtain a certificate from a trusted Certification Authority (*CA*). This should be your primary choice, and will be necessary for production-grade deployments.

- Create your own untrusted self-signed certificate. This can ease operations during the phase of software development. You can search articles online that explain how to do this, for example this one.

  Alternatively, it is much easier and convenient to use a self-signed certificate generation tool, such as mkcert. This kind of tools already take into account the requisites and limitations of most popular applications and browsers, so that you don't need to.

  Note that while a self-signed certificate can be used for web development, browsers will show a big security warning. Users will see this warning, and must click to accept the unsafe certificate before proceeding to the page.

  To generate new certificate files with mkcert, run these commands:

```
# Generate new untrusted self-signed certificate files:
CAROOT="$PWD" mkcert -cert-file cert.pem -key-file key.pem \
    "127.0.0.1" \
    "::1"       \
    "localhost" \
    "*.test.local"

# Make a single file to be used with Kurento Media Server:
cat cert.pem key.pem > cert+key.pem

# Protect against writes
chmod 440 *.pem
```

This command is just an example, and includes some useful things for local development: access from localhost in its IPv4, IPv6, and hostname forms; and also the `*.test.local` wildcard, meant to allow adding any desired subdomains to the `/etc/hosts` file in your development computer(s), so these cert files can be used not only for localhost but also for remote tests in your LAN.

Note: Simply using `*.test` would be nice, but wildcards are forbidden for global TLDs, so it wouldn't work. For example, MacOS 10.15 (*Catalina*) would reject such certificate (see mkcert bug 206). For this reason, we propose using `*.test.local`.

You can also publish a new Zeroconf local domain for any development machine. For example, running this in Ubuntu:

```
# Get and publish the IP address to the default network gateway.
IP_ADDRESS="$(ip -4 -oneline route get 1 | grep -Po 'src \K([\d.]+)')"
avahi-publish --address --no-reverse -v "dev.test.local" "$IP_ADDRESS"
```

- (Optional) Convert your untrusted self-signed certificate into a trusted one. This is done by installing the Root CA into the client device.

  On computers, installing the Root CA is easy because mkcert does it for you:

  ```
  CAROOT="$PWD" mkcert -install
  ```

  Installing the Root CA on mobile devices is a bit more difficult, because you cannot simply run mkcert:

  - On iOS, you can either use AirDrop, email the CA to yourself, or serve it from an HTTP server. After installing it, you must enable full trust in it. **Note**: earlier versions of mkcert ran into an iOS bug, if you can't see the root in "Certificate Trust Settings" you might have to update mkcert and regenerate the root.

    Note that only AirDrop, Apple Mail, or Safari are allowed to download and install certificates on iOS. Other applications will not work for this.

  - For Android, you will have to install the CA and then enable user roots in the development build of your app. See this StackOverflow answer.

WebRTC Statistics

[TODO full review]

22.1 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.

22.2 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*:
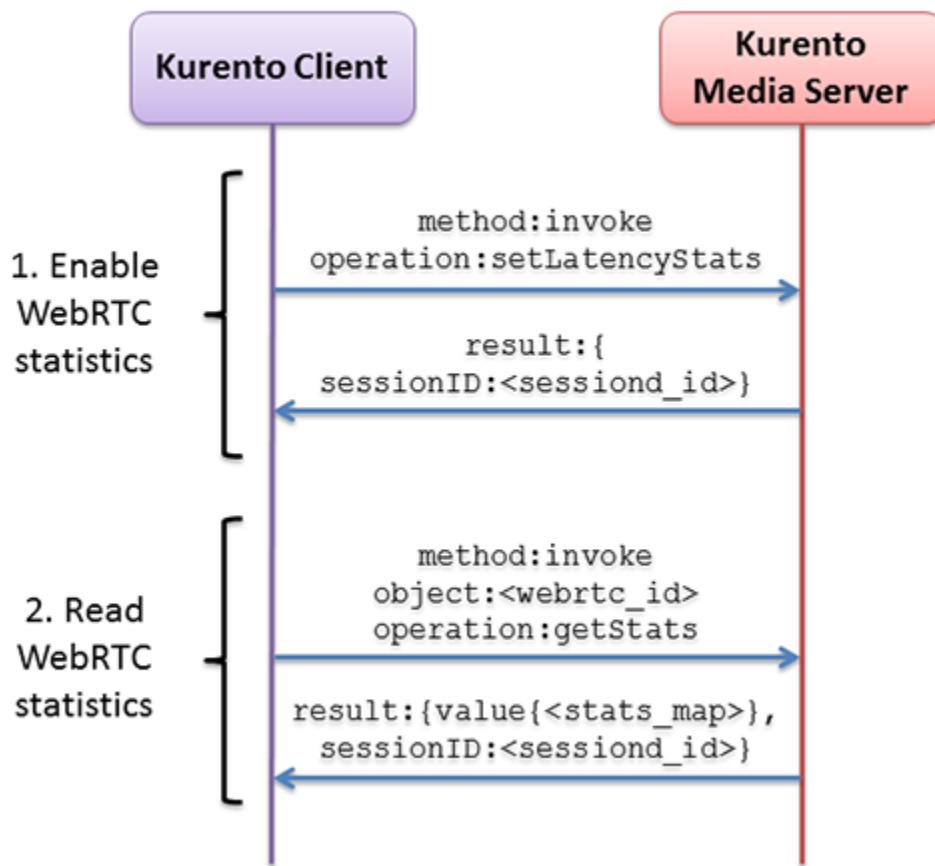
Fig. 1: *Sequence diagram for gathering WebRTC statistics in KMS*

22.3 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, client-side (i.e. browser WebRTC peer) stats are also gathered by the application. This is done using the standard method provided by the *RTCPeerConnection* 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

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

KMS e2e lantency: 0.278746 seconds

Fig. 2: *Statistics results in the kurento-loopback-stats demo GUI*

Debug Logging

23.1 Logs Location

When running Kurento Media Server manually with `/usr/bin/kurento-media-server`, all logging messages are by default printed to standard out (*stdout*).

The KMS native packages modify this behavior, placing logs in a more conventional location for the platform: `/var/log/kurento-media-server/`. This path can be customized in the service settings file, `/etc/default/kurento-media-server` (for Debian/Ubuntu packages).

Log files are named as follows:

```
{DateTime}.{LogNumber}.pid{PID}.log
```

- *{DateTime}*: Logging file creation date and time, in Extended Notation for the date, and Basic Notation for the time. For example: *2018-12-31T235959*.

- *{LogNumber}*: Log file number. A new one will be created whenever the maximum size limit is reached (100 MB by default).

- *{PID}*: Process Identifier of *kurento-media-sever*.

When the KMS service starts correctly, a log file such as this one will be created:

```
2018-06-14T194426.00000.pid13006.log
```

Besides normal log files, an *errors.log* file stores error messages and stack traces, in case KMS crashes.

Note: Log files in this folder are rotated, and old files will get eventually deleted when new ones are created. This helps with preventing that all available disk space ends up filled with logs.

23.2 Logs Format

Each line in a log file has a fixed structure:

```
{DateTime} {PID} {ThreadID} {Level} {Component} {FileLine} {Function} {Object}?
↪{Message}
```

- *{DateTime}*: Date and time of the logging message, in Extended Notation, with six decimal places for the seconds fraction. For example: *2018-12-31T23:59:59,123456*.

- *{PID}*: Process Identifier of *kurento-media-sever*.

- *{ThreadID}*: Thread ID from which the message was issued. For example: *0x000111122223333*.

- *{Level}*: Logging level. This value will typically be *INFO* or *DEBUG*. If unexpected error situations happen, the *WARNING* and *ERROR* levels will contain information about the problem.

- *{Component}*: Name of the component that generated the log line. For example: *KurentoModuleManager*, *webrtcendpoint*, *qtmux*, etc.

- *{FileLine}*: File name and line number, separated by a colon. For example: *main.cpp:255*.

- *{Function}*: Name of the function in which the log message was generated. For example: *main()*, *loadModule()*, *kms_webrtc_endpoint_gather_candidates()*, etc.

- *{Object}*: [Optional] Name of the object that issued the message, if one was specified for the log message. For example: *<kmswebrtcendpoint0>*, *<fakesink1>*, *<audiotestsrc0:src>*, etc.

- *{Message}*: The actual log message.

For example, when KMS starts correctly, a message like this will be printed:

```
2018-06-14T19:44:26,918243  13006  0x00007f59401f5880  info  KurentoMediaServer  main.
↪cpp:255  main()  Kurento Media Server started
```

23.3 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 filtered by two methods:

- Use a command-line argument if you are manually running KMS. For example, run:

```
/usr/bin/kurento-media-server \
  --gst-debug-level=3 \
  --gst-debug="Kurento*:4,kms*:4"
```

- You can also replace the command-line arguments with the environment variable *GST_DEBUG*. For example, run:

```
export GST_DEBUG="3,Kurento*:4,kms*:4"
/usr/bin/kurento-media-server
```

If you are using the native packages (installing KMS with *apt-get*) and running KMS as a system service, then you can also configure the *GST_DEBUG* variable in the KMS service settings file, /etc/default/ kurento-media-server:

```
# Logging level.
export GST_DEBUG="3,Kurento*:4,kms*:4"
```

Logs will be colored by default, but colors can be explicitly disabled in the same two ways: either with --gst-debug-no-color or with export GST_DEBUG_NO_COLOR=1. When running KMS as a system service, this option is enabled in order to generate clean logs without strange terminal ANSI color escape sequences.

23.4 Suggested levels

Here is a list of some logging levels that could be the most useful for typical troubleshooting of KMS. These are set as *environment variables*, so it is possible to copy them directly into the KMS service settings file, /etc/default/ kurento-media-server, or use them in a console session.

First, **start from the default levels**:

```
export GST_DEBUG="3,Kurento*:4,kms*:4,sdp*:4,webrtc*:4,*rtpendpoint:4,
↪rtp*handler:4,rtpsynchronizer:4,agnosticbin:4"
```

Then **add new levels** according to your needs:

- **Flowing of media**:

```
export GST_DEBUG="$GST_DEBUG,KurentoMediaElementImpl:5"
```

  - "KurentoMediaElementImpl:5" shows *MediaFlowIn* and *MediaFlowOut* state changes, allowing know if media is actually flowing between endpoints (see *MediaElement events*).

- **Transcoding of media**:

```
export GST_DEBUG="$GST_DEBUG,KurentoMediaElementImpl:5,agnosticbin*:5"
```

- "KurentoMediaElementImpl:5" shows *MediaTranscoding* state changes.

- "agnosticbin*:5" shows the requested and available codecs on Endpoints. When there is a mismatch, transcoding is automatically enabled.

- **WebRtcEndpoint** and **RtpEndpoint**:

```
export GST_DEBUG="$GST_DEBUG,Kurento*:5,KurentoWebSocket*:4"
export GST_DEBUG="$GST_DEBUG,kmssdpsession:5"
export GST_DEBUG="$GST_DEBUG,sdp*:5"
export GST_DEBUG="$GST_DEBUG,webrtcendpoint:5,kmswebrtcsession:5,kmsiceniceagent:5
↪"
```

- "Kurento*:5" shows all state changes (*MediaFlowIn*, *MediaFlowOut*, *MediaTranscoding*, etc). Use "KurentoWebSocket*:4" to avoid getting all verbose logs about the WebSocket communications.

- "kmssdpsession:5" prints the SDP messages (SDP Offer/Answer negotiation) processed by KMS.

- "sdp*:5" shows internal messages related to the construction of SDP messages and media handlers.

- "webrtcendpoint:5", "kmswebrtcsession:5", and "kmsiceniceagent:5" all contain the logic that governs ICE gathering and ICE candidate selection for WebRTC.

> **Note:**
>
> * See also *libnice* to enable advanced ICE logging for WebRTC.

You can also see messages about the REMB congestion control algorithm for WebRTC. However these will constantly be filling the log, so you shouldn't enable them unless explicitly working out an issue with REMB:

```
export GST_DEBUG="$GST_DEBUG,kmsremb:5"
```

- **PlayerEndpoint**:

```
export GST_DEBUG="$GST_DEBUG,kmselement:5,playerendpoint:5,appsrc:4,
↪agnosticbin*:5,uridecodebin:6,rtspsrc:5,souphttpsrc:5,*CAPS*:3"
```

- **RecorderEndpoint**:

```
export GST_DEBUG="$GST_DEBUG,basemediamuxer:5,KurentoRecorderEndpointImpl:4,
↪recorderendpoint:5,qtmux:5,curl*:5"
```

Other less commonly used logging levels are:

- **imageoverlay**, **logooverlay** (as used, for example, in some *Kurento Tutorials*):

```
export GST_DEBUG="$GST_DEBUG,imageoverlay:5,logooverlay:5"
```

- **RTP Synchronization**:

```
export GST_DEBUG="$GST_DEBUG,kmsutils:5,rtpsynchronizer:5,rtpsynccontext:5,
↪basertpendpoint:5"
```

- **JSON-RPC** API server calls:

```
export GST_DEBUG="$GST_DEBUG,KurentoWebSocket*:5"
```

- **Unit tests**:

```
export GST_DEBUG="$GST_DEBUG,check:5,test_base:5"
```

23.5 3rd-Party libraries

23.5.1 libnice

libnice is the GLib implementation of *ICE*, the standard method used by *WebRTC* to solve the issue of *NAT Traversal*.

This library uses the standard *GLib* logging functions, which 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.

To enable debug logging on *libnice*, set the environment variable *G_MESSAGES_DEBUG* with one or more of these values (separated by commas):

- *libnice*: Required in order to enable logging in libnice.

- *libnice-verbose*: Enable extra verbose messages.

- *libnice-stun*: Log messages related to the *STUN* protocol.

- *libnice-pseudotcp*: Log messages from the ICE-TCP module.

- *libnice-pseudotcp-verbose*: Enable extra verbose messages from ICE-TCP.

- *all*: Equivalent to using all previous flags.

After doing this, GLib messages themselves must be enabled in the Kurento logging system, by setting an appropriate level for the *glib* component.

Example:

```
export G_MESSAGES_DEBUG="libnice,libnice-stun"
export GST_DEBUG="$GST_DEBUG,glib:5"
/usr/bin/kurento-media-server
```

You can also set this configuration in the Kurento service settings file, which gets installed at /etc/default/kurento-media-server.

23.5.2 libsoup

libsoup is the GNOME HTTP client/server library. It is used to perform HTTP requests, and currently this is used in Kurento by the *KmsImageOverlay* and the *KmsLogoOverlay* filters.

It is possible to enable detailed debug logging of the HTTP request/response headers, by defining the environment variable SOUP_DEBUG=1 before running KMS:

```
export SOUP_DEBUG=1
/usr/bin/kurento-media-server
```

CHAPTER 24

Kurento 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 has worked to keep all out integration tests up to date, and he also wrote tutorials and test documentation.

- Pablo Fuente works in Kurento Room, OpenVidu, and also in Kurento's Java and JavaScript client libraries.

- Juan Navarro is our Kurento Media Server guy. He loves working close to the metal with C/C++ and GStreamer. Follow him on Twitter to get updates on new Kurento releases and the low-level stuff he is working on.

- 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.

Contribution Guide

The Kurento project accepts contributions from third parties in all kinds of forms:

- Bug reports

- Bug fixes

- New features

- Code enhancements

- Improvements to the *documentation*

- Improvements to the testability of the code itself

The way you can do this is through reporting bugs in the Issue Tracker, proposing changes via Pull Requests, or discussing other topics in the Kurento Public Mailing List.

Kurento team members will probably ask for further explanations, tests or validations of any code contributed to the project before it gets incorporated into its codebase. You must be ready to address these concerns before having your code approved for inclusion.

Please mind the following contribution guidelines:

25.1 Did you find a bug?

- **Ensure the bug was not already reported** by searching in our Issue Tracker.

- If you're unable to find an open issue addressing the problem, open a new one. Include a **title and clear description**, as much relevant information as possible, and a **code sample** or an **executable test case** that can be used to demonstrate the unexpected behavior.

- For more detailed information on submitting a bug report and creating an issue, visit our reporting guidelines.

25.2 Did you fix a bug?

- Open a new GitHub Pull Request with the patch.

- Ensure the PR description clearly describes the problem and its solution. Include the relevant Issue number, if applicable.

25.3 Did you fix whitespace, format code, or make a purely cosmetic patch?

Changes that are cosmetic in nature and do not add anything substantial to the stability, functionality, or testability of Kurento will generally not be accepted.

25.4 Do you intend to add a new feature or change an existing one?

- Before contributing a piece of work, we strongly suggest that you first write about your intentions in the Kurento Public Mailing List, so we can talk about the need and value of your changes.

- Specify the contents of your contribution:
  - **What problem is it trying to solve?**
  - **What are the consequences of the changes?**

- Specify the licensing restrictions of the code you contribute.

- By having some work contributed for incorporation in the Kurento project, you will be implicitly accepting Kurento to own the code copyright, so that the open-source nature of the project can be guaranteed.

- Remember that the Kurento project has no obligations in relation to accepting contributions from third parties.

25.5 Thanks for helping

Really, this wouldn't be possible without you!

Kindly, the Kurento Team

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.

Release Notes

27.1 6.15.0 (November 2020)

To install Kurento Media Server: *Installation Guide*.

27.1.1 Added

SDP generateOffer(): offerToReceiveAudio, offerToReceiveVideo

Albeit the way applications are typically written have the SDP Offer/Answer being started from the client (i.e. a client browser sends an SDP Offer to KMS), it has always been possible to do the opposite, and start the WebRTC negotiation from KMS, with the client method `generateOffer()` (i.e. KMS generates an SDP Offer that is sent to the client browser).

However, SDP Offers generated by KMS would always contain **one audio** and **one video** media-level sections (those lines starting with `m=` in the SDP). This would make it impossible to send an audio-only or video-only offer to the clients.

The WebRTC implementation in Kurento strives to be familiar to JavaScript developers who know about the RTCPeerConnection browser API, and in this case we are introducing the `offerToReceiveAudio` / `offerToReceiveVideo` options that are well known from the RTCPeerConnection.createOffer() method. These options can be used to change the SDP Offer generation in KMS, so it is now possible to control whether the offer includes audio or video, separately.

Example Java code, for an audio-only offer:

```java
import org.kurento.client.OfferOptions;
[...]

OfferOptions options = new OfferOptions();
options.setOfferToReceiveAudio(true);
options.setOfferToReceiveVideo(false);
```

```
String sdpOffer = webRtcEp.generateOffer(options);
```

WebRTC: externalIPv4 & externalIPv6 instead of externalAddress

The `externalAddress` parameter, introduced in Kurento 6.13.0, offered a way to tell the media server about its own public IP, without the need to use an additional STUN server. This is a nice feature because then the WebRTC connectivity checks could save some time trying to find out about the public IP in the first place. However, `externalAddress` would replace *all of IPv4 and IPv6* addresses in the ICE candidate gathering phase, which is problematic because the actual type of address wasn't being taken into account for the substitution.

With the new parameters, `externalIPv4` and `externalIPv6`, it is now possible to define either of static public IPv4 and/or IPv6 addresses. It is now possible to specify any of those, or both at the same time. As usual, this can be done upon installation, by editing the Kurento configuration files, or otherwise it can be set dynamically by the Client Application, via an API call offered by the Java or JavaScript client SDKs.

These parameters supersede the `externalAddress`, which now is **deprecated** and scheduled for removal in the next major release of Kurento.

Thanks to @prlanzarin (Paulo Lanzarin) for contributing these new parameters in Kurento/kms-elements#27 (*Add externalIPv4/externalIPv6 configs and API methods for them*).

Missing modules now available again

Some of the Kurento example modules had been broken for a while (since around version 6.10). This was caused by incompatibilities with newer versions of system libraries in Ubuntu 18.04, so we had to disable the build system for these modules due to a lack of maintainers that could get them up to date.

This situation has now been corrected and all of the additional modules can be installed with `apt` and are also included by default in Docker images: kms-chroma, kms-crowddetector, kms-markerdetector, kms-platedetector, kms-pointerdetector.

All plugins using HTTP are working again, too. This includes the HTTP recording capabilities of the **RecorderEndpoint**, and some plugins such as the **ImageOverlayFilter** which in turn is used by **FaceOverlayFilter** (both available from kms-filters); a demo of the *FaceOverlayFilter* can be seen with the WebRTC Magic Mirror tutorial.

Remember that these example modules are provided **exclusively for demonstration purposes**.

27.1.2 Changed

TURN port configuration in Coturn and Kurento.

Previous versions of the Kurento FAQ contained this sentence:

> *> Port ranges must match between Coturn and Kurento Media Server. > Check the files /etc/turnserver.conf and /etc/kurento/modules/kurento/BaseRtpEndpoint.conf.ini, to verify that both will be using the same set of ports.*

This was *wrong*. Normally, when a TURN server is in use it will be deployed in a network segment that is different from both the one where KMS is, and where the WebRTC peers are. The TURN server acts as a relay which allows connecting both of those networks when a direct connection cannot be achieved between them. As such, there is no reason why the TURN server should be using the same port range than the WebRTC peers, of which KMS is one.

Service stop script now waits until actually exiting.

The scripts that run when Kurento is started as a system service (either with `systemctl` or the traditional `service kurento-media-server (start|stop|restart)`) would not wait until the KMS process released all its resources and exited, when issuing a stop request. Instead, stopping the service would return immediately, regardless of the time KMS took to stop. This meant that the `restart` operation was broken: the current KMS instance would be told to stop, and a new one would be launched immediately, causing conflicts between both instances if the old one took some time to exit.

This has now been changed so the `stop` and `restart` actions wait a maximum of 25 seconds for the KMS process to stop; after that time, if the process didn't exit yet, it will be forcefully killed.

Remember this only applies to KMS instances that are initiated using the system service tools; other kinds of deployments (such as Docker containers) have their own way to manage the lifetime of the media server processes.

Updated OpenH264 to v1.5.0.

OpenH264 is the H.264 codec implementation offered by Cisco. The maximum version of OpenH264 that we can build is 1.5.0. Cisco introduced a breaking change starting from 1.6.0, which means our fork of gst-plugins-bad won't build with that version. In order to update OpenH264 further than 1.5.0, we need to drop our old GStreamer forks and move to more up to date versions.

27.1.3 Deprecated

- `externalAddress` should be replaced by `externalIPv4` and/or `externalIPv6`.

27.1.4 Fixed

SDP generateOffer() H.264 profile-level-id.

When a WebRTC implementation creates a new SDP Offer in order to negotiate usage of the H.264 video codec, this must include some attributes that Kurento wasn't including in its own offers. This has now been fixed, and SDP Offers created by Kurento will include the required attributes for H.264: `level-asymmetry-allowed=1; packetization-mode=1;profile-level-id=42e01f`.

RecorderEndpoint HTTP docs.

Client docs ([Java](), [JavaScript]()) wrongly stated that the HTTP recording mode requires an HTTP server with support for the `PUT` method. This was wrong, and now it correctly explains that the HTTP method is `POST` in chunked mode (using the HTTP header `Transfer-Encoding: chunked`).

GStreamer Merge Request 38.

All commits from this change (https://gitlab.freedesktop.org/gstreamer/gst-plugins-good/-/merge_requests/38) have been backported from the upstream repository, fixing a memory leak related to incorrect handling of RTCP packets, as described in issue 522 (https://gitlab.freedesktop.org/gstreamer/gst-plugins-good/-/issues/522).

Fixed libnice DDoS

Some users reported a worrying bug that could end up used to force DDoS attacks to machines running Kurento Media Server. The summary of it is that simply sending any bit of data to one of the TCP-Passive ports where the WebRTC engine was listening for connections, would cause a thread to spin-lock and use 100% of a CPU core.

The issue was studied and located to be in [libnice](), a 3rd-party library that Kurento uses to implement the ICE protocol which is part of WebRTC. *libnice* developers had a great reaction time and this issue got fixed in a matter of days!

Kurento now comes with libnice 0.1.18, the newest version of this library, meaning that our favorite media server is now safer and more robust.

Thanks to @darrenhp for reporting this issue in Kurento/bugtracker#486 (*libnice cause cpu 100% and don't restore after 'attacked' by a spider or security-scaner*).

Fixed Recorder synchronization on streaming gaps

The **RecorderEndpoint** suffered from an audio/video synchronization issue that multiple users have noticed over time. The root cause of this issue was packet loss in the network: any missing packet would cause a gap in the sequence of audio timestamps that got stored to file. These gaps would make the audio and video tracks to drift over time, ending up with a noticeable difference and with a broken lipsync.

A new method to avoid timestamp gaps has been added to the recorder: now, whenever a gap is detected, the recorder will overwrite the timestamps in order to take into account the missing data. This way, both audio and video will still match and be synchronized during playback.

27.1.5 Other changes

This list includes other changes and fixes contributed by users and/or fellow developers, who merit our sincere appreciation and thanks for sharing their work with the Kurento project:

kms-core

- @mariogasparoni (Mario Junior) for Kurento/kms-core#13 (*Remove audio delay/pause when connecting new endpoints to composite*).

kms-elements

- @prlanzarin (Paulo Lanzarin) for Kurento/kms-elements#23 (*Fix STUN server usage when stunServerPort isn't set and the default value is to be used*).

- @t-mullen (Thomas Mullen) for Kurento/kms-elements#24 (*Fix incorrect cropping in AlphaBlending element*).

kurento-client-js

- @JoseGoncalves (José Miguel Gonçalves) for Kurento/kurento-client-js#8 (*Clear ping-pong timer when closing client*).

- @marcosdourado (Marcos Dourado) for Kurento/kurento-client-js#10 (*remove lib error-tojson and add method locally to stringify error*).

- @tuttieee (Yuichiro Tsuchiya) for Kurento/kurento-client-js#11 (*Fix register() calls inside lib/index.js*).

kurento-tutorial-js

- @tuttieee (Yuichiro Tsuchiya) for Kurento/kurento-tutorial-js#9 (*Use IceCandidateFound event instead of deprecated OnIceCandidate event*).

27.2 6.14.0 (June 2020)

This new release of Kurento is following the previously set path of focusing in stability and fixing bugs. Some critical issues have been solved, related to high CPU usage on real-world deployments that were supporting lots of users, so don't hesitate to update your media servers!

To install Kurento Media Server: *Installation Guide*.

27.2.1 Changed

- **Deleted inline WebSocket++ sources**.

  WebSocket++ (*websocketpp*) is the library that provides WebSocket support for Kurento Media Server. Up until now, the whole source code of this library was included in the Kurento Media Server source tree, but it wasn't being updated accordingly to the new developments of other dependent libraries, such as newer versions of OpenSSL. This was causing build issues for some users, so starting from Kurento 6.14.0, we've deleted all websocketpp source code from our tree, and instead will be depending on the websocketpp packages that are provided by the Operating System itself (Ubuntu package *libwebsocketpp-dev*).

  The change doesn't affect client applications of Kurento, but it will be good news for people who were building KMS from sources.

- **Reduced default log levels for WebRTC-related modules**.

  Some basic operations, such as SDP Offer/Answer negotiations, ended up logging thousands of lines that don't really convey anything useful in a typical deployment, and can make it harder to find other more interesting messages. For this reason, the bulk of general messages that are part of WebRTC have been modified from `INFO` to `DEBUG` log level (see *Debug Logging*).

- **Flexible TLS negotiation for Secure WebSocket**. Kurento 6.13.2 moved towards using **TLS 1.2** with its Secure WebSocket control endpoint, as a response to all of the most used browsers moving away from previous versions of TLS. This applies to applications that are written in Browser JavaScript, and want to connect directly from the browser to the JSON-RPC port of Kurento media Server.

  However, for greater flexibility, we have now changed this to allow a flexible security negotiation with clients. Instead of forcing that clients connect with the Secure WebSocket using a fixed version of TLS 1.2, the server (OpenSSL behind the scenes) is now allowed to negotiate the highest version of TLS supported by the client, among TLS 1.0, TLS 1.1, and TLS 1.2.

  Note that TLS 1.3 is not supported yet by OpenSSL 1.0.2, which is the version used on Ubuntu 16.04 Xenial. However, when we finish migration to more modern OS versions, newer OpenSSL versions in the system will mean that Kurento will implicitly support TLS 1.3 too.

27.2.2 Fixed

- **Locked 100% CPU when releasing multiple Endpoints**.

  This issue was affecting a good number of installations. There was a severe performance penalty in the disconnection process between Endpoints, so in scenarios with a lot of connections, there was a good probability that the CPU got locked in a permanent 100% CPU usage loop.

  An example of this would be one-to-many scenarios (1:N), were a single presenter would be sending video to some hundredth consumers. When the producer disconnected their WebRtcEndpoint, trying to disconnect so many consumers would almost always trigger the issue.

  More info can be found in the related Pull Request: Workerpool rewrite.

- **Fix recording of AAC audio with MKV**.

  The MKV profile uses AAC audio, but due to wrong settings it was actually not working. The Agnosticbin component in Kurento would get confused and fail to find an acceptable encoder to perform the audio transcoding, and the recording would end up not working with an infinite stream of ERROR messages.

27.2.3 Thank You

There were other changes and fixes contributed by users, who merit our sincere appreciation and thanks for sharing their work with the Kurento project:

doc-kurento

- @alexnum (Alessandro) for Kurento/doc-kurento#4 (*Handling self-signed certificates in nodeJs*).

- @piyushwadhwani for Kurento/doc-kurento#5 (*adapting to newer version of spring boot*).

kurento-jsonrpc-js

- @JoseGoncalves (José Miguel Gonçalves) for Kurento/kurento-jsonrpc-js#6 (*Fix transportMessage*).

kurento-tutorial-java

- @hgcummings (Harry Cummings) for Kurento/kurento-tutorial-java#14 (*Increase max text message buffer size for websocket connections*).

kurento-utils-js

- @simoebenhida (Mohamed Benhida) for Kurento/kurento-utils-js#30 (*Refactor deprecated syntax*), and @oscar-mnfuentes for Kurento/kurento-utils-js#32 (*Fix promise call in setDescription to make compatible with safari*).

27.3 6.13.2 (May 2020)

To install Kurento Media Server: *Installation Guide*.

27.3.1 Added

- The `kms-datachannelexample` plugin package has been restored and is now again available in the Apt package repository. This plugin is needed to run the WebRTC DataChannel *Tutorials*.

27.3.2 Changed

- All Kurento Media Server Docker images now come with *Debug Symbols* already installed. This means that in case of a server crash, the process output will contain a useful stack trace that includes file names and line numbers of the exact point in source code where the error happened.

  This is very helpful for writing useful bug reports.

- `BaseRtpEndpoint`'s event MediaStateChangedEvent documentation has been rewritten, so now it contains a better explanation of what exactly is reported by this event.

- Media server's Secure WebSocket connections (`wss://` as opposed to `ws://`) is now using TLS 1.2 instead of the old TLS 1.0, which has been deprecated by most web browsers.

  This should help make the browser-JavaScript tutorials to work again (*Tutorials*); before this, some browsers (such as Safari since version 13.1) would reject to establish a connection between the browser and KMS, due to the old version of TLS in use by the server.

27.3.3 Fixed

- `BaseRtpEndpoint`'s method getMediaState has been changed to really return the current value of `MediaState` (see *BaseRtpEndpoint events*).

  Previously, this method was wrongly returning the current value of a different property: `MediaFlowIn`.

- The server's **WebSocket** connection now seamlessly falls back to IPv4 when IPv6 was enabled in the config but the system didn't support it (commit Kurento/kurento-media-server@4e543d0).

  Before this fix, having enabled IPv6 in the config (`/etc/kurento/kurento.conf.json`) would mean that the WebSocket connection *had* to be done through IPv6. Otherwise, an unhelpful message "`Underlying Transport Error`" would show up and the server wouldn't start.

- Properly reject multiple `m=` lines in SDP messages (commit Kurento/kms-core@6a47630).

  With the current state of affairs regarding the official browser JavaScript API for WebRTC, developers face a confusing mix of *stream*-based, *track*-based, and *transceiver*-based API methods. The way these APIs work together is not always clear cut, and some users have been mistakenly sending to Kurento SDP messages with multiple media-level video sections, which is not supported by the media server. This was poorly handled before, and now the server will correctly reject the second and further video (or audio) media lines in an SDP message.

- Ignore invalid mDNS ICE candidates. This will prevent trying to handle unresolved mDNS candidates from Chrome or any other browser that might generate such kind of candidates (commit Kurento/kms-elements@44ca3de).

  With this change, any mDNS candidates that don't belong to the local network, or otherwise cannot be resolved to an IP address for whatever reason, will be silently ignored instead of causing warning messages in the log.

27.3.4 libnice 0.1.16

The libnice library is a core part of our WebRTC implementation, because it handless all of the ICE process (candidate gathering and connectivity checking). It has now been updated to the latest version **0.1.16**, which brings lots of improvements and fixes.

Some of the most relevant changes are:

- Bug 95 (*ERROR:conncheck.c:1899:priv_mark_pair_nominated: assertion failed: (pair->state == NICE_CHECK_DISCOVERED) crash*) has been fixed with commit 6afcb580.

  This bug was affecting some users of Kurento, and caused that the whole process was aborted when libnice encountered an unexpected change of internal states. This now should be fixed, making the server more robust and reliable.

- Commit 099ff65c introduced the feature of ignoring by default network interfaces from virtual machines and containers. So, from now on Kurento will ignore interface names that start with "*docker*", "*veth*", "*virbr*", and "*vnet*".

  This change reduces the amount of work that the ICE protocol needs to do when trying to establish WebRTC connections with remote peers, thus having the connectivity tests be much quicker. It also prevents some edge cases where libnice selected a virtual network interface as the best possible candidate, which would have Kurento sending and/or receiving streams in convoluted loopbacks through the local virtual devices.

  You can complement this with the `networkInterfaces` parameter of WebRtcEndpoint (either with `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`, Java, or JavaScript), which allows to select the exact network interface(s) that Kurento Media Server should use for WebRTC connections.

- Miscellaneous improvements, such as connectivity keepalives, peer-reflexive candidates, memory leaks, and lots of other small fixes.

- Version **0.1.16** of libnice also introduced a backwards-breaking change that might negatively affect applications that use this library: nice_agent_remove_stream() was silently made asynchronous.

  This change broke the usage inside Kurento, which assumed the previous behavior of assuming a synchronous method. For sample code that shows how we worked around this issue, have a look at commit Kurento/kms-elements@a4c9f35.

  This point is noted here to warn other application owners about this issue. If you want to track progress on this change, see the previous link for the bug report we opened.

27.3.5 Thank You

There were other changes and fixes contributed by users, who merit our sincere appreciation and thanks for sharing their work with the Kurento project:

kurento-docker

- @tuttieee for Kurento/kurento-docker#14 (*Add KMS_MIN_PORT and KMS_MAX_PORT env vars*).

kms-elements

- @prlanzarin (Paulo Lanzarin) for Kurento/kms-elements#23 (*Fix STUN server usage when stunServerPort isn't set and the default value is to be used*).

27.4 6.13.0 (December 2019)

Kurento Media Server **6.13.0** has been released! It comes with some new API methods that allow to query Kurento about its own resource usage, as well as some new configuration parameters that can be used to fine-tune some aspects of how the server chooses ICE candidates during WebRTC initialization.

To install Kurento Media Server: *Installation Guide*.

These Release Notes were also posted on the Kurento blog.

27.4.1 Added

- **WebRTC**: Add externalAddress to WebRtcEndpoint config & client API.

  Allows to specify an external IP address, so Kurento doesn't need to auto-discover it during WebRTC initialization. This saves time, and also removes the need for configuring external STUN or TURN servers.

  The effect of this parameter is that all local ICE candidates that are gathered will be mangled to contain the provided external IP address instead of the local one, before being sent to the remote peer. Thanks to this, remote peers are able to know about the external or public IP address of Kurento.

  Use this parameter if you know beforehand what will be the external or public IP address of the media server (e.g. because your deployment has an static IP), although keep in mind that some types of networks will still need you to install a TURN server. Best thing to do is to try with this option enabled, and if WebRTC fails, then default to the standard method of installing and configuring Coturn.

  Kurento Client API docs: Java, JavaScript.

- **WebRTC**: Add networkInterfaces to WebRtcEndpoint config & client API.

  If you know which network interfaces should be used to perform ICE (for WebRTC connectivity), you can define them here. Doing so has several advantages:

– The WebRTC ICE gathering process will be much quicker. Normally, it needs to gather local candidates for all of the network interfaces, but this step can be made faster if you limit it to only the interface that you know will work.

– It will ensure that the media server always decides to use the correct network interface. With WebRTC ICE gathering it's possible that, under some circumstances (in systems with virtual network interfaces such as "docker0") the ICE process ends up choosing the wrong local IP.

There is the long-running issue of how libnice gathers all possible local IP addresses for its ICE candidates, which introduces latency or connectivity problems for some many-networks deployments (like Amazon EC2, or Docker/Kubernetes): Kurento generates too many ICE candidates, and that results in the situation that sometimes (quite often, in practice) it fails to choose correct pair of ICE candidates and uses those ones from private networks, leading to non-obvious bugs and video stability problems.

More rationale for this feature can be found here: Kurento/bugtracker#278 (*RFC: Add WebRtcEndpoint.externalIPs configuration parameter*).

Kurento Client API docs: Java, JavaScript.

- **WebRTC / RTP**: Add `mtu` to BaseRtpEndpoint config & client API.

  Allows configuring the network MTU that Kurento will use for RTP transmissions, in both RtpEndpoint and WebRtcEndpoint. This parameter ends up configured in the GStreamer RTP payloader (`rtpvp8pay`, `rtph264pay`).

  Kurento Client API docs: Java, JavaScript.

- **RTP**: Add support for `a=rtcp:{Port}` in SDP messages.

  Allows a remote peer using non-consecutive RTCP ports. Normally, the RTCP port is just RTP+1, but with an `a=rtcp` attribute, the RTCP port can be set to anything.

  Eg. with this SDP media line:

  ```
  m=video 5004 RTP/AVP 96
  ```

  RTP listen port is set to 5004, and RTCP listen port is implicitly set to 5005.

  However, with these SDP media lines:

  ```
  m=video 5004 RTP/AVP 96
  a=rtcp:5020
  ```

  RTP listen port is set to 5004, but RTCP listen port is 5020.

  This allows interoperation with other RTP endpoints that require using arbitrary RTCP ports.

- **ServerManager**: Add `getCpuCount()` and `getUsedCpu()` methods to the client API.

  These new methods can be called to obtain information about the number of CPU cores that are being used by Kurento, together with the average CPU usage that is being used in a given time interval:

  ```java
  import org.kurento.client.KurentoClient;
  [...]
  private KurentoClient kurento;
  [...]
  ServerManager sm = kurento.getServerManager();
  log.info("CPU COUNT: {}", sm.getCpuCount()); // Allowed CPUs available to use by
  ↪Kurento
  log.info("CPU USAGE: {}", sm.getUsedCpu(1000)); // Average CPU usage over 1 second
  log.info("RAM USAGE: {}", sm.getUsedMemory()); // Resident memory used by the
  ↪Kurento process
  ```

Kurento Client API docs: Java, JavaScript.

27.4.2 Changed

- **kurento-utils.js**: Dropped use of legacy `offerToReceiveAudio` / `offerToReceiveVideo` in RTCPeerConnection.createOffer(), in favor of the **Transceiver API**.

  This was needed because Safari does not implement the legacy attributes. As of this writing, all of Firefox, Chrome and Safari have good working support for RTCPeerConnection transceivers, with RTCPeerConnection.addTransceiver().

- **WebRTC**: Don't ERROR or WARN with unresolved mDNS candidates during WebRTC ICE candidate gathering.

  mDNS candidates from outside networks (such as the other peer's local networks) will be unresolvable in our local networks. This is, after all, the main purpose of mDNS! To conceal your local IPs behind a random hostname, such that others cannot resolve it into an IP address.

  In other words, mDNS candidates are only of type "host", and are only useful with Local LAN WebRTC connections. It makes no sense to show an error or a warning each and every time an mDNS candidate cannot be resolved, because the majority of use cases involve remote WebRTC connections.

- **WebRTC / RTP**: Change default `maxVideoRecvBandwidth` to 0 ("unlimited").

  It doesn't make much sense that Kurento purposely limits the incoming bitrate to such a low value. Better leave it to negotiate the best bitrate by using congestion control (REMB).

  Kurento Client API docs: Java, JavaScript.

- **ServerManager**: The client API method `getUsedMemory()` now returns **resident (RSS)** instead of **virtual (VSZ)** memory.

  Resident memory is a more useful measurement because it tells the physical used memory, which is usually what users want to know about their server. Giving virtual size here wouldn't be of much use, as the server (or any of its libraries) could map a huge area, then not use it, and the reported VSZ would be huge for no real benefit.

  RSS gives a good view about how many MB are being used by KMS at any given time. This is also what users check on `htop` or `top` so see how much memory is used by KMS. However, keep in mind that if you are trying to establish whether Kurento Media Server has a memory leak, then neither `top` nor `ps` are the right tool for the job; Valgrind is.

  Kurento Client API docs: Java, JavaScript.

- **Documentation**: Rewritten all the `{Min,Max}` bandwidth / bitrate texts for BaseRtpEndpoint and WebRtcEndpoint.

  Kurento defaults to a very conservative maximum bitrate for outgoing streams; most applications will want to raise this value, but API documentation was not very clear so these sections needed a good review.

27.4.3 Fixed

- **Node.js tutorials**: Fix broken usage of the WebSocket module.

  The dependency package **ws** had introduced since version 3.0.0 a breaking change in the `connection` event handler. We are now using latest versions of this package, so the tutorial code needed to be updated for this change.

27.5 6.12.0 (October 2019)

Kurento Media Server **6.12** has been released!

To install it: *Installation Guide*.

27.5.1 Added

- Added support for the null ICE candidate that Firefox sends to signal the end of Trickle ICE.

- Resolve mDNS candidate names that Chrome uses in place of local candidate IP Addresses for privacy reasons (PSA: mDNS and .local ICE candidates are coming).

- New package *kurento-dbg* (to be installed with `apt-get`), that installs all optional debugging symbols needed for Kurento crash reports.

- Add support for compilation with **Clang** and **UndefinedSanitizer** in the Kurento helper build script: Kurento/kms-omni-build/bin/kms-build-run.sh. It now allows to choose between GCC and Clang compilers, together with support for AddressSanitizer, ThreadSanitizer, and UndefinedBehaviorSanitizer.

27.5.2 Changed

- *libsrtp* fork updated to version 1.6.0; custom patch reviewed to fix "**unprotect failed code 9**" warning messages (Kurento/bugtracker#246).

- User **kurento** gets now created with an User ID belonging to the *system* category (which gets an UID >= `100`), instead of the *user* category (which gets an UID >= `1000`). This fixes the issue of the user *kurento* showing up in the Ubuntu login screen.

- The home directory for the user *kurento* has been moved from `/var/kurento` to `/var/lib/kurento`, which complies with the Linux Foundation's Filesystem Hierarchy Standard (FHS) (Kurento/bugtracker#364).

- Enable C++14 language spec. when building C++ code.

27.5.3 Deprecated

- The old home directory for the user *kurento*, which was located at `/var/kurento`, is now re-created as a symbolic link to the new location in `/var/lib/kurento`. **Applications should migrate to the new location**. The next major release of Kurento will stop providing the fallback link in the old path.

27.5.4 Fixed

- Fix the **MediaFlowInStateChange crash**. Kurento/bugtracker#393 (*MediaFlowInStateChange Seg Fault*).

- Fix leaks and posible crash in PlayerEndpoint. Kurento/bugtracker#198 (*PlayerEndpoint leaks 2 sockets*).

- Fix GStreamer memory leak in DTLS handling.

- Fix memory leak in classes auto-generated by Kurento Module Creator.

- Fix potential uncaught exceptions in *kms-core* when parsing SDP messages.

27.6 6.11.0 (July 2019)

Kurento Media Server **6.11** has been released! This new version brings several improvements and fixes that have happened while we work on moving KMS to use the newer GStreamer 1.14 in Ubuntu Bionic.

To install it: *Installation Guide*.

27.6.1 New SDP syntax for WebRTC DataChannels

Firefox moved to the newer *SDP* syntax for SCTP (WebRTC DataChannels), and soon enough Chrome will also do the same. It was just a matter of time until support for DataChannels was totally broken (and it already started to be with Firefox), so this was a much needed update in Kurento.

This article explains the change: How to avoid Data Channel breaking.

Old style SDP syntax was like this:

```
m=application 54111 DTLS/SCTP 5000
a=sctpmap:5000 webrtc-datachannel 16
```

The new syntax is *same same, but different*:

```
m=application 54111 UDP/DTLS/SCTP webrtc-datachannel
a=sctp-port:5000
```

And the fix was implemented in this pull request. We're maintaining backwards compatibility, just like the Mozilla article explains: *old offer results in old answer, and new offer results in new answer*.

27.6.2 Tutorials fix for Node.js 10.x LTS

We had received several reports of the *Tutorials* not working properly with the latest Node.js *Long Term Support* release, **10.x**. This was due to some very outdated dependencies listed in our `package.json` files.

This issue should now be fixed, as the broken dependencies have been cleaned up, removing unneeded ones (`utf-8-validate`) and updating the others (`bufferutil`, `ws`) to latest versions.

Thanks

- @VuThuyThuy97 for Kurento/bugtracker#362 (npm install error: Node-gyp rebuild).
- @oisnot for Kurento/kurento-jsonrpc-js#2 (npm install kurento-client failed because this package used ancient p...)
- @alQlagin (Alex Kulagin) for Kurento/kurento-jsonrpc-js#3 (Remove blocking dependency).
- @yonghongren (Yonghong Ren) for Kurento/kurento-jsonrpc-js#4 (update utf-8-validate to the latest).
- @sm2017 for Kurento/bugtracker#386, Kurento/kurento-jsonrpc-js#5 (Upgraded dependencies).

27.6.3 Spring Boot 2 and Java 8

We've updated the Kurento Java client and tutorials, to make use of Spring Boot 2. This has forced us to leave Java 7 behind, and now Java 8 is the new standard version on which Kurento and OpenVidu projects will be developed and deployed.

27.6.4 Fix GStreamer memory leaks

The *gst-plugins-bad* module was leaking some memory for each new DTLS connection that was established. This meant that each and every WebRTC session was leaking memory!

After some effort spent with debuggers and delving into the library's code, a very simple merge request was issued to the upstream project, which got accepted right away: gstreamer/gst-plugins-bad!422 (Fix leaked dtlscertificate in dtlsagent).

The memory increments got hugely reduced, but there is still some continuous increase on used memory (this time it is 10x smaller than previously, but it is still there). We'll keep working on this, and should find out more about the remaining leaks during next weeks.

27.6.5 More thanks

There were many other changes and fixes contributed by users; these maybe didn't have an impact big enough to be featured in these Release Notes, but nevertheless their authors merit our sincere appreciation and thanks for sharing their work with the greater community:

kurento-utils-js

- @fishg for Kurento/kurento-utils-js#23 (fix WebRtcPeerRecvonly mode in safari).
- @aenriquezgentile for Kurento/kurento-utils-js#24 (Updating uuid to 3.2.1 version).
- @rpuerta85 for Kurento/kurento-utils-js#25 (Enable to make it work on IE using Temasys Plugin).
- @kindritskyiMax (Kindritskiy Maksym) for Kurento/kurento-utils-js#26 (allow usage in node).
- @solomax (Maxim Solodovnik) for Kurento/kurento-utils-js#28 (Safari 12 support).

kurento-client-js

- @tomhouman (Tom Houman) for Kurento/kurento-client-js#2 (Websocket connection options).

kurento-tutorial-js

- @xenyou (Yoshihiro Kikuchi) for Kurento/kurento-tutorial-js#3 (fix kurento-recorder demo to record video).
- @neilyoung for Kurento/kurento-tutorial-js#5 (Fixed stat E2E latency ms display issue).
- @soogly for Kurento/kurento-tutorial-js#7 (Issue with needless parameter).

kurento-tutorial-java

- @Kr0oked (Philipp Bobek) for Kurento/kurento-tutorial-java#8 (fix typo).

27.7 6.10.0 (Apr 2019)

Kurento Media Server **6.10** is seeing the light with some important news!

To install it: *Installation Guide*.

27.7.1 Hello Ubuntu Bionic

Preliminary support for Ubuntu 18.04 LTS (Bionic Beaver) has landed in Kurento, and all the CI machinery is already prepared to compile and generate Debian packages into a new repository.

To install KMS on this version of Ubuntu, just follow the usual *installation instructions*:

```
DISTRO="bionic"  # KMS for Ubuntu 18.04 (Bionic)

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/6.15.0 $DISTRO kms6
EOF

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

Mostly everything is already ported to Bionic and working properly, but **the port is not 100% finished yet**. You can track progress in this board: https://github.com/orgs/Kurento/projects/1

The two biggest omissions so far are:

- None of the *extra* modules have been ported yet; i.e. the example plugins that are provided for demonstration purposes, such as *kms-chroma*, *kms-crowddetector*, *kms-platedetector*, *kms-pointerdetector*.

- OpenCV plugins in Kurento still uses the old C API. This still worked out fine in Ubuntu 16.04, but it doesn't any more in 18.04 for plugins that use external training resouces, such as the HAAR filters.

  Plugins that need to load external OpenCV training data files won't work. For now, the only plugin affected by this limitation in KMS seems to be **facedetector** because it won't be able to load the newer training data sets provided by OpenCV 3.2.0 on Ubuntu 18.04. Consequently, other plugins that depend on this one, such as the **faceoverlay** filter and its FaceOverlayFilter API, won't work either.

  Not much time has been invested in these two plugins, given that they are just simple demonstrations and no production application should be built upon them. Possibly the only way to solve this problem would be to re-write these plugins from scratch, using OpenCV's newer C++ API, which has support for the newer training data files provided in recent versions of OpenCV.

  This issue probably affects others of the *extra* modules mentioned earlier. Those haven't even started to be ported.

27.7.2 Hello Chrome 74

Google is moving forward the security of WebRTC world by dropping support for the old DTLS 1.0 protocol. Starting from Chrome 74, DTLS 1.2 will be required for all WebRTC connections, and all endpoints that want to keep compatibility must be updated to use this version of the protocol.

Our current target operating system, Ubuntu 16.04 (Xenial), provides the library OpenSSL version **1.0.2g**, which already offers support for DTLS 1.2[1]. So, the only change that was needed to bring Kurento up to date in compatibility with Chrome was to actually make use of this newer version of the DTLS protocol.

27.7.3 Bye Ubuntu Trusty

Our resources are pretty limited here and the simpler is our CI pipeline, and the less work we need to dedicate making sure KMS works as expected in all Operating Systems, the best effort we'll be able to make improving the stability and features of the server.

Our old friend **Ubuntu 14.04 LTS (Trusty Tahr)** is reaching its deprecation and End Of Life date in April 2019 (source: https://wiki.ubuntu.com/Releases) so this seems like the best time to drop support for this version of Ubuntu.

---

[1] DTLS 1.2 was added in OpenSSL 1.0.2: Major changes between OpenSSL 1.0.1l and OpenSSL 1.0.2 [22 Jan 2015].

Canonical is not the only one setting an end to Trusty with their lifecycle schedules; Google is also doing so because by requiring DTLS 1.2 support for WebRTC connections, Trusty is left out of the game, given that it only provides OpenSSL **1.0.1f** which doesn't support DTLS 1.2[1].

27.7.4 Reducing forks to a minimum

Moving on to Ubuntu 18.04 and dropping support of Ubuntu 14.04 are efforts which pave the road for a longer-term target of dropping custom-built tools and libraries, while standarizing the dependencies that are used in the Kurento project.

These are some of the objectives that we'd like to approach:

- Using standard tools to create Ubuntu packages.

  We're dropping the custom compile_project.py tool and instead will be pushing the use of git-buildpackage. Our newer kurento-buildpackage.sh uses *git-buildpackage* for the actual creation of Debian packages from Git repositories.

- Dropping as many forks as possible. As shown in *Code repositories*, Kurento uses a good number of forked libraries that are packaged and distributed as part of the usual releases, via the Kurento package repositories.

  Keeping all these forks alive is a tedious task, very error-prone most of the times. Some are definitely needed, such as openh264 or usrsctp because those are **not** distributed by Ubuntu itself so we need to do it.

  Some others, such as the fork of libsrtp, have already been dropped and we're back to using the official versions provided by Ubuntu (yay!)

  Lastly, the big elephant in the room is all the GStreamer forks, which are stuck in an old version of GStreamer (**1.8**) and would probably benefit hugely from moving to newer releases.

  We hope that moving to Ubuntu 18.04 can ease the transition from our forks of each library to the officially provided versions.

- Ultimately, a big purpose we're striving for is to **have Kurento packages included among the official ones in Ubuntu**, although that seems like a bit far away for now.

27.7.5 Clearer Transcoding log messages

Codec transcoding is always a controversial feature, because it is *needed* for some cases which cannot be resolved in any other way, but it is *undesired* because it will consume a lot of CPU power.

All debug log messages related to transcoding have been reviewed to make them as clear as possible, and the section *Troubleshooting Issues* has been updated accordingly.

If you see that transcoding is active at some point, you may get a bit more information about why, by enabling this line:

```
export GST_DEBUG="${GST_DEBUG:-3},Kurento*:5,agnosticbin*:5"
```

in your daemon settings file, `/etc/default/kurento-media-server`.

Then look for these messages in the media server log output:

- `Upstream provided caps:  (caps)`
- `Downstream wanted caps:  (caps)`
- `Find TreeBin with wanted caps:  (caps)`

Which will end up with either of these sets of messages:

- If source codec is compatible with destination:

    - `TreeBin found! Use it for (audio|video)`

    - `TRANSCODING INACTIVE for (audio|video)`

- If source codec is **not** compatible with destination:

    - `TreeBin not found! Transcoding required for (audio|video)`

    - `TRANSCODING ACTIVE for (audio|video)`

These messages can help understand what codec settings are being received by Kurento ("*Upstream provided caps*") and what is being expected at the other side by the stream receiver ("*Downstream wanted caps*").

27.7.6 Recording with Matroska

It's now possible, thanks to a user contribution, to configure the RecorderEndpoint to use the Matroska multimedia container (MKV), using the H.264 codec for video.

This has big implications for the robustness of the recording, because with the MP4 container format it was possible to lose the whole file if the recorder process crashed for any reason. MP4 stores its metadata only at the end of the file, so if the file gets truncated it means that it won't be playable. Matroska improves the situation here, and a truncated file will still be readable.

For more information about the issues of the MP4 container, have a look a then new knowledge section: *H.264 video codec*.

27.7.7 New JSON settings parser

Kurento uses the JSON parser that comes with the Boost C++ library; this parser accepted comments in JSON files, so we could comment out some lines when needed. The most common example of this was to force using only VP8 or H.264 video codecs in the Kurento settings file, `/etc/kurento/modules/kurento/SdpEndpoint.conf.json`:

```
"videoCodecs" : [
  {
    "name" : "VP8/90000"
  },
  {
    "name" : "H264/90000"
  }
]
```

This is the default form of the mentioned file, allowing Kurento to use either VP8 or H.264, as needed. To disable VP8, this would change as follows:

```
"videoCodecs" : [
//  {
//    "name" : "VP8/90000"
//  },
  {
    "name" : "H264/90000"
  }
]
```

And it worked fine. The Boost JSON parser would ignore all lines starting with `//`, disregarding them as comments.

However, starting from Boost version 1.59.0, the Boost JSON parser gained the great ability of not allowing comments; it was rewritten without any consideration for backwards-compatibility (yeah, it wouldn't hurt the Boost devs if they practiced a bit of "*Do NOT Break Users*" philosophy from Linus Torvalds, or at least followed Semantic Versioning. . .)

The devised workaround has been to allow inline comment characters inside the JSON attribute fields, so the former comment can now be done like this:

```
"videoCodecs": [
    { "//name": "VP8/90000" },
    { "name": "H264/90000" }
]
```

Whenever you want to comment out some line in a JSON settings file, just append the `//` characters to the beginning of the field name.

27.7.8 Code Sanitizers and Valgrind

If you are developing Kurento, you'll probably benefit from running AddressSanitizer, ThreadSanitizer, and other related tools that help finding memory, threading, and other kinds of bugs.

kms-cmake-utils includes now the arsenm/sanitizers-cmake tool in order to integrate the CMake build system with the mentioned compiler utilities. You'll also find some useful **suppressions** for these tools in the kms-omni-build dir.

Similarly, if you want to run KMS under Valgrind, kms-omni-build contains some utility scripts that can prove to be very handy.

27.7.9 Special Thanks

A great community is a key part of what makes any open source project special. From bug fixes, patches, and features, to those that help new users in the forum / mailing list and GitHub issues, we'd like to say: **Thanks!**

Additionally, special thanks to these awesome community members for their contributions:

- @prlanzarin (Paulo Lanzarin) for:
  - *Add API support for MKV profile for recordings* Kurento/kms-core#14, Kurento/kms-elements#13.
  - *Fixed config-interval prop type checking in basertpendpoint and rtppaytreebin* Kurento/kms-core#15 and @leetal (Alexander Widerberg) for reporting #321.
  - *rtph26[45]depay: Don't handle NALs inside STAP units twice (cherry-picked from upstream)* Kurento/gst-plugins-good#2.
- @tioperez (Luis Alfredo Perez Medina) for reporting #349 and sharing his results with RTSP and Docker.
- @goroya for Kurento/kurento-media-server#10.

27.8 6.9.0 (Dec 2018)

27.8.1 (lib)Nicer and performant

Howdy!

A new release of Kurento Media Server brings huge stability and performance improvements.

Until recently, some Kurento users had been suffering a fatal bug which would cause crashes and segmentation faults in some specific configurations, related to abruptly interrupted TCP connections. Thanks to the invaluable help of

Olivier Crête, who is the main maintainer and developer of the **libnice** library, this bug has been fixed and all projects that make use of this library are now able to benefit from the fix.

Kurento is of course one of these projects, making use of the *libnice* library to provide WebRTC compatibility. Starting from version 6.9.0, the latest improvements made to the *libnice* library have been integrated, and now all related crashes have stopped happening. Once again, we want to send **a huge thank you to Olivier** for his great work!

Another relevant topic for this release is all the effort directed at creating proper test infrastructure to perform **load and stress testing** with Kurento Media Server. We pushed forward to unify some work that had been laying around in different branches, and another high-impact patch ended up merged into mainline *libnice*. This time, removing a global lock that caused a performance cap.

Our intention has been to raise the number of simultaneous streams that the media server is able to sustain, as a direct response to the lackluster results that Kurento obtained in a recent benchmark / comparison between WebRTC implementations, published in webrtcH4cKS a couple of months ago. For this purpose, one of the hard tasks was to create a custom stress test that could be used as proof for Kurento capabilities; at first we were able to reproduce the reduced performance shown in the published benchmark, but as of latest weeks, Kurento has been more on par with other solutions:

- Kurento was able to support a maximum of **133** simultaneously connected users (browsers): **19 conference rooms with 7 users each**.

- Due to the SFU architecture, this means that KMS sustained **133 inbound streams**, and **798 outbound streams**.

- The test machine was an **8-core**, **16GB RAM** instance on AWS.

- Test videos are **540×360** in size.

- All participants were simulated with Chrome browsers, so the video formats were always compatible between senders and receivers. This is a very important detail because it means that video *transcoding* ("*on-the-fly adaptation of codecs and formats between incompatible clients*") was not necessary and thus it was inactive.

During these stress tests, CPU usage grew with number of participants, and RAM usage stayed at around 25% (4GB):
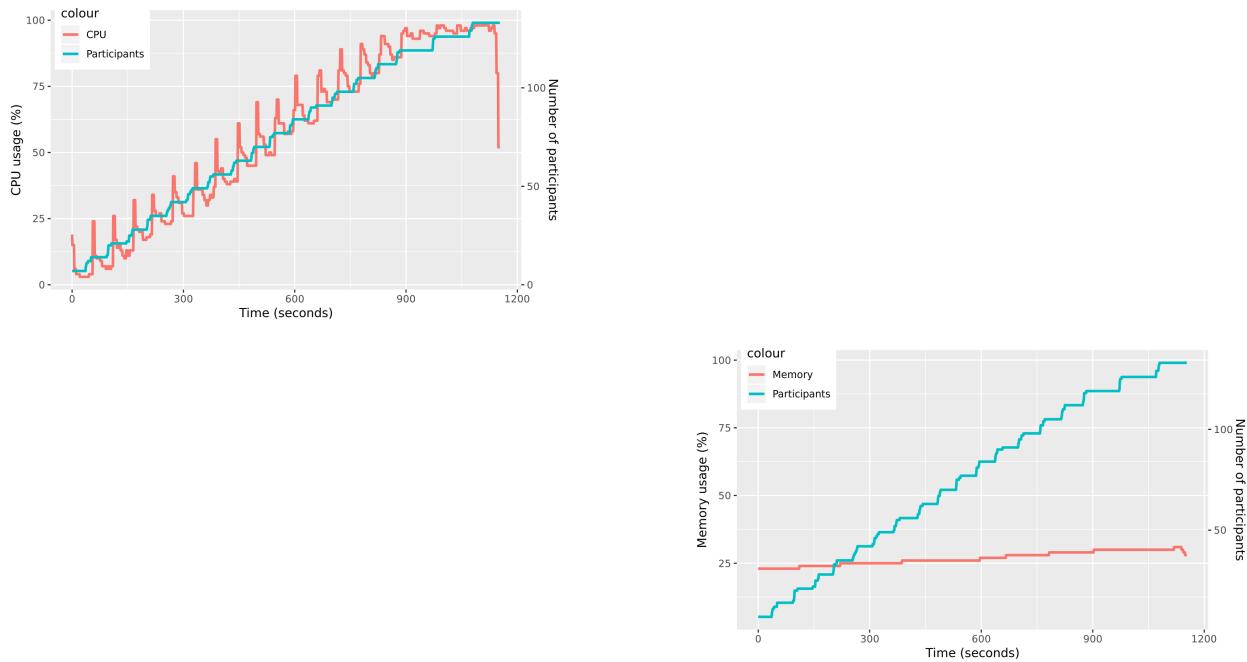
Next is a summary of all additions, changes and/or fixes that are included in this release:

27.8.2 Added

- [kms-elements] The port range that *PlayerEndpoint* uses for incoming RTP streams can now be configured with the `rtspClientPortRange` property in `/etc/kurento/modules/kurento/PlayerEndpoint.conf.ini`. Thanks Mislav Čakarić for the Pull Request #14!

- [kms-omni-build] The `bin/` folder now contains a set of scripts that help developers run Valgrind to find memory leaks and access errors.

- [doc-kurento] Our documentation has received a big bump in the project management section, this time related to how our End-to-End tests are designed and how Kurento developers can use them. This has little to no relevance to end users, but it's a first step into the long-term objective of letting developers who use Kurento have better knowledge about our testing software.

  If you want to have a look, check out the *Testing section*. Thanks Boni García for all the work!

27.8.3 Changed

- [kms-core] The default port range for incoming RTP streams (applicable to both *RtpEndpoint* and *WebRtcEndpoint*) was `[0, 65535]`. This changed to exclude all privileged ports, which are system-protected and require root permissions to be used; now the port range is `[1024, 65535]`.

  Of course, you are still able to configure a different range of ports by setting the *minPort* and/or *maxPort* properties in `/etc/kurento/modules/kurento/BaseRtpEndpoint.conf.ini`.

- [kurento-media-server] The daemon settings file in `/etc/default/kurento-media-server` now disables colored output by default. This is useful to produce a clean error log file in `/var/log/kurento-media-server/errors.log`.

- [kurento-media-server] There was some confusion surrounding how to configure the System Limits that Kurento needs for working, specifically the **Open File Descriptors limit**. These are configured by the daemon files during the service startup, but there was no error control and these limits could end up being *silently left with*

their default values (which is a bad thing and a sure way to have performance issues with the media server). There are now controls in place that will set up some valid limits.

- [kurento-media-server] Error messages in `/var/log/kurento-media-server/errors.log` are now separated between executions of Kurento Media Server, with this line:

```
<TIMESTAMP> -- New execution
```

Before this change, all errors were appended without separation, which could become confusing and users had problems identifying where older executions finished and where the latest one started.

27.8.4 Fixed

- [kurento-media-server] The daemon would crash if the debug log files couldn't be created for any reason (e.g. because of permission problems). After this fix, if log files cannot be created or accessed, file logging will be disabled and logging will revert back to being printed to *stdout*.

27.9 6.8.1 (Oct 2018)

Release 6.8.0 contained a critical bug that was immediately patched into 6.8.1, so these Release Notes include changes from both versions.

This release goes hand-in-hand with a new version of **OpenVidu**, Kurento's sister project. Check OpenVidu 2.5.0 Release Notes if you are interested in building any of the common use cases that are covered by that project, such as conference calls and video chat rooms.

27.9.1 Added

- Log messages that come from *GLib*-based libraries are now integrated into the general Kurento logging system.

    Previous to this addition, the only way to obtain debug logs from the *libnice* library was to run KMS directly on console; even after enabling debug logging, the relevant messages would not appear in the Kurento logs because *libnice* was just printing its messages in the standard output.

    Starting from KMS 6.8.0, all messages will be redirected to Kurento logs, located at `/var/log/kurento-media-server/`. Remember that specific 3rd-party libraries such as *libnice* still require that their logging functions are explicitly enabled; check *libnice* for more details.

- **Hub** and **HubPort** elements now support for DATA streams. This means that a WebRTC DataChannels stream can be processed through a *Hub*, for example a **Composite**, and the DATA stream will be available for the element to process.

- Thanks to the previous addition, **Composite** element has now support for merging multiple DataChannel streams.

- **GStreamerFilter** is now able to set its inner element's properties "on the fly" during runtime. For example, if you used a **coloreffects** filter, before this addition you would need to configure the video parameters beforehand, and they would stay the same during the whole execution of the Kurento pipeline. Now, it is possible to change the filter's properties at any time, during the execution of the pipeline.

    The OpenVidu project is using this capability to offer real-time audio/video filtering during WebRTC calls; check Voice and video filters for more details.

27.9.2 Changed

- Output logs now use standard format ISO 8601 for all timestamps. This affects both log files names, and their contents:

  Log files will now be named such as this:

  ```
  2018-06-14T194426.00000.pid13006.log
  ```

  And each individual log message will contain timestamps of this form:

  ```
  2018-06-14T19:44:26,918243
  ```

- `disableRequestCache` is now exposed in settings file (*kurento.conf.json*). This can be used to disable the RPC Request Cache for troubleshooting or debugging purposes.

- Clearer log messages about what is going on when the maximum resource usage threshold is reached.

- System service configuration file `/etc/default/kurento-media-server` now contains more useful examples and explanations for each option.

- **libnice** has been updated from the old version 0.1.13 to the newer **0.1.15** (snapshot - not officially released yet). This should help with fixing a crash issue that KMS has been suffering lately. For more details, see "*Known Issues*" below.

27.9.3 Fixed

- System service init files will now **append** to the error log file `/var/log/kurento-media-server/errors.log`, instead of truncating it on each restart.

27.9.4 Known Issues

- *libnice* is a 3rd-party library used by Kurento, which in the last months has been suffering of a serious bug that caused crashes in some specific conditions. There are several places where this issue has been discussed, such as the issue #247 in Kurento, the issue #33, and the mail topic in the Kurento List.

  In our release **6.8.1** we have upgraded *libnice* to the development/snapshot build that will at some point become the official **libnice 0.1.15**; this seems to work much better than the older version 0.1.13 that we had been using so far in all previous releases of Kurento Media Server.

  We haven't been able to repeat the crash with this latest version of *libnice*, however we'll wait until this new version has been tested more extensively, and only then will consider this issue as closed.

  If you are interested or have been affected by this issue, note that in the days previous to writing these Release Notes there has been some movement in the issue #33 from the *libnice* project. Hopefully we are closer to a definite fix for this!

27.10 6.7.2 (May 2018)

27.10.1 Added

- *WebRtcEndpoint* now allows to specify the desired direction for generated SDP Offers.

The most common use case for WebRTC calls is to generate an SDP Offer in the Browser and then pass it over to process in Kurento (which in turn generates an SDP Answer). However the opposite workflow is also possible: to have Kurento generating the SDP Offer, with the browser being the one which gets to process it.

This feature has been part of Kurento for a while, but there was a small bit missing in the picture: the SDP Offers generated by Kurento had always the *direction* attribute set for a two-way media stream, i.e. both receiving and sending media. Now, it is possible to configure the *WebRtcEndpoint* to generate SDP Offers with all three possible directions: `a=sendrecv`, `a=sendonly`, `a=recvonly`, for send-receive, send-only, and receive-only, respectively.

27.10.2 Changed

- *WebRtcEndpoint* now doesn't use reduced minimum interval for RTCP packets when the *REMB* network congestion control algorithm is not in use.

  RFC 3550 section 6.2 (*RTCP Transmission Interval*) recommends an initial value of 5 seconds for the interval between sending of RTCP packets. However, Kurento configures a much lower value, and RTCP packets are sent every 500 milliseconds. The reasoning for this is that the congestion control algorithm (*Congestion Control / REMB*) should be able to react as soon as possible when congestion occurs, and this is helped by a faster RTCP interval.

  The use case for this change is audio-only calls; Kurento doesn't support using the network congestion algorithm for audio streams, so it doesn't make sense to force sending high-frequency RTCP packets in this case. This has the secondary effect of greatly reducing consumed bandwidth when an audio track is muted via one of the browser's *RTCPeerConnection* methods.

27.10.3 Fixed

- Send mandatory attribute `a=setup:actpass` in SDP Offers. **RFC 3550** section 5 (*Establishing a Secure Channel*) states:

```
The endpoint that is the offerer MUST use the setup attribute
value of setup:actpass
```

  This was not the case in Kurento; Chrome was permissive about this mistake, but Firefox was complaining about the lack of this attribute in SDP Offers generated by the *WebRtcEndpoint*.

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*.

28.1 Introduction

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 build tool of choice, and is used to build all modules.
- Source code is versioned in several GitHub repositories.
- The officially supported platforms are Long-Term Support (*LTS*) versions of Ubuntu: **Ubuntu 16.04 (Xenial)** and **Ubuntu 18.04 (Bionic)** (64-bits only).
- The GStreamer multimedia framework sits at the heart of Kurento Media Server.
- In addition to GStreamer, KMS uses other libraries like boost, jsoncpp, libnice, etc.

28.2 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.

An overview of the relationships between all repos forming the Kurento Media Server:
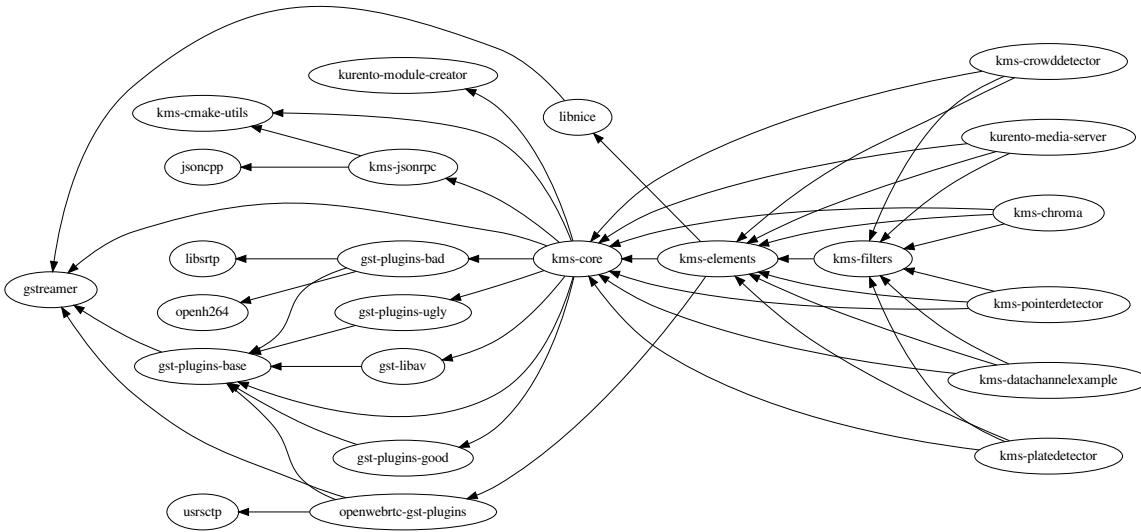
Fig. 1: All dependency relationships

As the dependency graph is not strictly linear, there are multiple possible ways to order all modules into a linear dependency list; this section provides one possible ordered list, which will be consistently used through all Kurento documents.

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.

- jsoncpp
- libsrtp
- openh264
- usrsctp
- gstreamer (produces libgstreamer1.5)
- gst-plugins-base
- gst-plugins-good
- gst-plugins-bad
- gst-plugins-ugly
- gst-libav
- openwebrtc-gst-plugins
- libnice (produces gstreamer1.0-nice, gstreamer1.5-nice)

Main repositories

- 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-jsonrpc: Kurento protocol is based on JsonRpc, and makes use of a JsonRpc library contained in this repository. It has C++ code.

- 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.

- kurento-media-server: Contains the main entry point of KMS. That is, the main() function for the server executable code. This program depends on libraries located in the above repositories. It has mainly C++ code.

Extra repositories

KMS is distributed with some basic GStreamer pipeline elements, but other elements are available in form of modules. These modules are *demos* of what third party modules could be written and integrated into Kurento. These are just for instructional purposes, and shouldn't be used in production servers.

- kms-chroma

- kms-crowddetector

- kms-platedetector

- kms-pointerdetector

Omni-Build repository

This repository is a 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.

- kms-omni-build

Client repositories

Application Servers 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.

- kurento-client-js (Node.js Application Servers, browser JavaScript)

- kurento-java (Java Application Servers)

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-opencv-plugin-sample

- kms-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 theK MS Main repositories, while it's unusual to make changes in Fork repositories except for updating their upstream versions.

28.3 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.

28.3.1 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 used when compiling programs 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)
```

28.3.2 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 *apt-get*, 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. Programs 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`.

28.3.3 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 projects 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.

28.4 Build from sources

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:

1. Install required tools, like Git.

2. Add the Kurento repository to your system configuration.

3. Clone **kms-omni-build**.

4. Install build dependencies: tools like GCC, CMake, etc., and KMS development libraries.

5. Build with CMake and Make.

6. Run the newly compiled KMS.

7. Run KMS tests.

28.4.1 Install required tools

This command will install the basic set of tools that are needed for the next steps:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    build-essential \
    ca-certificates \
    cmake \
    git \
    gnupg
```

28.4.2 Add Kurento repository

Run these commands to add the Kurento repository to your system configuration:

```
# Import the Kurento repository signing key
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 5AFA7A83

# Get Ubuntu version definitions
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add the repository to Apt
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 $DISTRIB_CODENAME kms6
EOF

sudo apt-get update
```

28.4.3 Install build dependencies

Run:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    kurento-media-server-dev
```

28.4.4 Download KMS source code

Run:

```
git clone https://github.com/Kurento/kms-omni-build.git
cd kms-omni-build
git submodule update --init --recursive
git submodule update --remote
```

Note: `--recursive` and `--remote` are not used together, because each individual submodule may have their own submodules that might be expected to check out some specific commit, and we don't want to update those.

OPTIONAL: Change to the *master* branch of each submodule, if you will be working with the latest version of the code:

```
REF=master
git checkout "$REF" || true
git submodule foreach "git checkout $REF || true"
```

You can also set *REF* to any other branch or tag, such as `REF=6.12.0`. This will bring the code to the state it had in that version release.

28.4.5 Build and run KMS

Make sure your current directory is already *kms-omni-build*, then run this command:

```
export MAKEFLAGS="-j$(nproc)"
./bin/kms-build-run.sh
```

By default, the script kms-build-run.sh will set up the environment and settings to make a Debug build of KMS. You can inspect that script to learn about all the other options it offers, including builds for AddressSanitizer, selection between GCC and Clang compilers, and other modes.

You can set the logging level of specific categories by exporting the environment variable *GST_DEBUG* before running this script (see *Debug Logging*).

28.4.6 Clean up 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|usrsctp).*'
    '^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 unknown package"; }
done
```

28.5 Install debug symbols

Whenever working with KMS source code itself, of during any analysis of crash in either the server or any 3rd-party library, you'll want to have debug symbols installed. These provide for full information about the source file name and line where problems are happening; this information is paramount for a successful debug session, and you'll also need to provide these details when requesting support or *filing a bug report*.

Installing the debug symbols does not impose any extra load to the system. So, it doesn't really hurt at all to have them installed even in production setups, where they will prove useful whenever an unexpected crash happens to bring the system down and a postmortem stack trace is automatically generated.

After having *installed Kurento*, first thing to do is to enable the Ubuntu's official **Debug Symbol Packages** repository:

```
# Import the Ubuntu debug repository signing key
sudo apt-key adv \
    --keyserver keyserver.ubuntu.com \
    --recv-keys F2EDC64DC5AEE1F6B9C621F0C8CAB6595FDFF622

# Get Ubuntu version definitions
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add the repository to Apt
sudo tee "/etc/apt/sources.list.d/ddebs.list" >/dev/null <<EOF
# Official Ubuntu repos with debug packages
deb http://ddebs.ubuntu.com ${DISTRIB_CODENAME} main restricted universe multiverse
deb http://ddebs.ubuntu.com ${DISTRIB_CODENAME}-updates main restricted universe↵
↪multiverse
EOF
```

Now, install all debug symbols that are relevant to KMS:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    kurento-dbg
```

28.5.1 Why are debug symbols useful?

Let's see a couple examples that show the difference between the same stack trace, as generated *before* installing the debug symbols, and *after* installing them. **Don't report a stack trace that looks like the first one in this example**:

```
# ==== NOT USEFUL: WITHOUT debug symbols ====

$ cat /var/log/kurento-media-server/errors.log
Segmentation fault (thread 139667051341568, pid 14132)
Stack trace:
[kurento::MediaElementImpl::mediaFlowInStateChange(int, char*, KmsElementPadType)]
/usr/lib/x86_64-linux-gnu/libkmscoreimpl.so.6:0x1025E0
[g_signal_emit]
/usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0:0x2B08F
[check_if_flow_media]
/usr/lib/x86_64-linux-gnu/libkmsgstcommons.so.6:0x1F9E4
[g_hook_list_marshal]
/lib/x86_64-linux-gnu/libglib-2.0.so.0:0x3A904
```

```
# ==== USEFUL: WITH debug symbols ====

$ cat /var/log/kurento-media-server/errors.log
Segmentation fault (thread 140672899761920, pid 15217)
Stack trace:
[kurento::MediaElementImpl::mediaFlowInStateChange(int, char*, KmsElementPadType)]
/home/kurento/kms-omni-build/kms-core/src/server/implementation/objects/
→MediaElementImpl.cpp:479
[g_signal_emit]
/build/glib2.0-prJhLS/glib2.0-2.48.2/./gobject/gsignal.c:3443
[cb_buffer_received]
/home/kurento/kms-omni-build/kms-core/src/gst-plugins/commons/kmselement.c:578
[g_hook_list_marshal]
/build/glib2.0-prJhLS/glib2.0-2.48.2/./glib/ghook.c:673
```

The second stack trace is much more helpful, because it indicates the exact file names and line numbers where the crash happened. With these, a developer will at least have a starting point where to start looking for any potential bug.

It's important to note that stack traces, while helpful, are not a 100% replacement of actually running the software under a debugger (**GDB**) or memory analyzer (**Valgrind**). Most crashes will need further investigation before they can be fixed.

28.6 Run and debug with GDB

GDB is a debugger that helps in understanding why and how a program is crashing. Among several other things, you can use GDB to obtain a **backtrace**, which is a detailed list of all functions that were running when the Kurento process failed.

You can build KMS from sources and then use GDB to execute and debug it. Alternatively, you can also use GDB with an already installed version of KMS.

28.6.1 From sources

1. Complete the previous instructions on how to build and run from sources: *Build from sources*.

2. Install debug symbols: *Install debug symbols*.

3. Build and run KMS with GDB.

   For this step, the easiest method is to use our launch script, *kms-build-run.sh*. It builds all sources, configures the environment, and starts up the debugger:

   ```
   ./bin/kms-build-run.sh --gdb
   # [... wait for build ...]
   (gdb)
   ```

4. Run GDB commands to *start KMS* and then get a *backtrace* (see indications in next section).

28.6.2 From installation

You don't *have* to build KMS from sources in order to run it with the GDB debugger. Using an already existing installation is perfectly fine, too, so it's possible to use GDB in your servers without much addition (apart from installing *gdb* itself, that is):

1. Assuming a machine where KMS is *installed*, go ahead and also install *gdb*.

2. Install debug symbols: *Install debug symbols*.

3. Define the *G_DEBUG* environment variable.

   This helps capturing assertions from 3rd-party libraries used by Kurento, such as *GLib* and *GStreamer*:

   ```
   export G_DEBUG=fatal-warnings
   ```

4. Load your service settings.

   You possibly did some changes in the KMS service settings file, /etc/default/
   kurento-media-server. This file contains shell code that can be sourced directly into your current session:

   ```
   source /etc/default/kurento-media-server
   ```

5. Ensure KMS is not already running as a service, and run it with GDB.

   ```
   sudo service kurento-media-server stop

   gdb /usr/bin/kurento-media-server
   # [ ... GDB starts up ...]
   (gdb)
   ```

6. Run GDB commands to *start KMS* and then get a *backtrace* (see indications in next section).

28.6.3 GDB commands

Once you see the `(gdb)` command prompt, you're already running a GDB session, and you can start issuing debug commands. Here, the most useful ones are *backtrace* and *info* variants (Examining the Stack). When you want to finish, stop execution with *Ctrl+C*, then type the *quit* command:

```
# Actually start running the KMS process
(gdb) run

# At this point, KMS is running; wait until the crash happens,
# which will return you to the "(gdb)" prompt.
#
# Or you can press "Ctrl+C" to force an interruption.
#
# You can also send the SIGSEGV signal to simulate a segmentation fault:
# sudo kill -SIGSEGV "$(pgrep -f kurento-media-server)"

# Obtain an execution backtrace
(gdb) backtrace

# Change to an interesting frame and get all details
(gdb) frame 3
(gdb) info frame
(gdb) info args
(gdb) info locals

# Quit GDB and return to the shell
(gdb) quit
```

Explaining GDB usage is out of scope for this documentation, but just note one thing: in the above text, `frame 3` is **just an example**; depending on the case, the backtrace needs to be examined first to decide which frame number is the most interesting. Typically (but not always), the interesting frame is the first one that involves Kurento's own code instead of 3rd-party code.

28.7 Work on a forked library

These are the two typical workflows used to work with fork libraries:

28.7.1 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.

28.7.2 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]

28.8 Create Deb packages

You can easily create Debian packages (*.deb* files) for KMS itself and for any of the forked libraries. Typically, Deb packages can be created directly by using standard system tools such as dpkg-buildpackage or debuild, but in order to integrate the build process with Git, we based our tooling on gbp (git-buildpackage).

28.8.1 kurento-buildpackage script

All Kurento packages are normally built in our CI servers, using a script aptly named kurento-buildpackage. When running this tool inside any project's directory, it will configure Kurento repositories, install dependencies, and finally use *git-buildpackage* to update the *debian/changelog* file, before actually building new Deb packages.

You can also use *kurento-buildpackage* locally, to build test packages while working on any of the Kurento projects; default options will generally be good enough. However, note that the script assumes all dependencies to either be installable from current Apt repositories, or be already installed in your system. If you want to allow the script to install any Kurento dependencies that you might be missing, run it with `--install-kurento <KurentoVersion>`, where *<KurentoVersion>* is the version of Kurento against which the project should be built.

For example, say you want to build the current *kms-core* development branch against Kurento 6.12.0. Run these commands:

```
git clone https://github.com/Kurento/adm-scripts.git
git clone https://github.com/Kurento/kms-core.git
cd kms-core/
../adm-scripts/kurento-buildpackage.sh \
    --install-kurento 6.12.0 \
    --apt-add-repo
```

Run `kurento-buildpackage.sh --help`, to read about what are the dependencies that you'll have to install to use this tool, and what are the command-line flags that can be used with it.

28.8.2 kurento-buildpackage Docker image

In an attempt to make it easier than ever to create Deb packages from Kurento repositories, we offer a Docker image that already contains everything needed to run the *kurento-buildpackage* tool. You can use this Docker image as if you were running the script itself, with the advantage that your system won't have to be modified to install any dependencies, your builds will be completely repeatable, and you will be able to create packages for different versions of Ubuntu.

To use the kurento-buildpackage Docker image, you'll need to bind-mount the project directory onto the `/hostdir` path inside the container. All other options to *kurento-buildpackage* remain the same.

For example, say you want to build the current *kms-core* development branch against Kurento 6.12.0, for *Ubuntu 16.04 (Xenial)* systems. Run these commands:

```
git clone https://github.com/Kurento/kms-core.git
cd kms-core/
docker run --rm \
    --mount type=bind,src="$PWD",dst=/hostdir \
    kurento/kurento-buildpackage:xenial \
        --install-kurento 6.12.0 \
        --apt-add-repo
```

28.9 Unit Tests

KMS uses the Check unit testing framework for C (https://libcheck.github.io/check/). If you are developing KMS and *building from sources*, you can build and run unit tests manually: just change the last one of the build commands from `make` to `make check`. All available tests will run, and a summary report will be shown at the end.

Note: It is recommended to first disable GStreamer log colors, that way the resulting log files won't contain extraneous escape sequences such as `^[[31;01m ^[[00m`. Also, it could be useful to specify a higher logging level than the default; set the environment variable *GST_DEBUG*, as explained in *Logging levels and components*.

The complete command would look like this:

```
export GST_DEBUG_NO_COLOR=1
export GST_DEBUG="3,check:5"
make check
```

The log output of the whole test suite will get saved into the file *./Testing/Temporary/LastTest.log*. To find the starting point of each individual test inside this log file, search for the words "**test start**". For the start of a specific test, search for "**{TestName}: test start**". For example:

```
webrtcendpoint.c:1848:test_vp8_sendrecv: test start
```

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
```

28.9.1 How to disable tests

Debian tools will automatically run unit tests as part of the *package creation* process. However, for special situations during development, we might want to temporarily disable testing before creating an experimental package. For example, say you are investigating an issue, and want to see what happens if you force a crash in some point of the code; or maybe you want to temporarily change a module's behavior but it breaks some unit test.

It is possible to skip building and running unit tests automatically, by editing the file `debian/rules` and changing the *auto_configure* rule from `-DGENERATE_TESTS=TRUE` to `-DGENERATE_TESTS=FALSE -DDISABLE_TESTS=TRUE`.

28.10 How-To

28.10.1 How to add or update external libraries

Add or change it in these files:

- *debian/control*.
- *CMakeLists.txt*.

28.10.2 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.

28.10.3 How to work with API changes

What to do when you are developing a new feature that spans across KMS and the public API? This is a summary of the actions done in CI by `adm-scripts/kurento_generate_java_module.sh` and `adm-scripts/kurento_maven_deploy.sh`:

1. Work on your changes, which may include changing the KMS files where the Kurento API is defined.

2. Generate client SDK dependencies:

```
cd <module>  # E.g. kms-filters
rm -rf build
mkdir build && cd build
cmake .. -DGENERATE_JAVA_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_GENERATION=TRUE
cd java
mvn clean install
```

3. Generate client SDK:

```
cd kurento-java
mvn clean install
```

4. At this point, the new Java packages have been generated and installed *in the local repository*. Your Java application can now make use of any changes that were introduced in the API.

28.10.4 Known problems

- 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 29

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 two Long-Term Support (*LTS*) versions of Ubuntu: **Ubuntu 16.04 (Xenial)** and **Ubuntu 18.04 (Bionic)** (64-bits only).

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:

- Release: `deb http://ubuntu.openvidu.io/<KmsVersion> <UbuntuCodename> kms6`

- Nightly: `deb http://ubuntu.openvidu.io/dev <UbuntuCodename> kms6`

Here, *<KmsVersion>* is any of the released versions of KMS (e.g. *6.7.2*, *6.8.1*, *6.9.0*, etc.) and *<UbuntuCodename>* is the name of each supported Ubuntu version (e.g. *xenial*, *bionic*, etc. if more added in the future.)

We also have several Continuous-Integration (*CI*) jobs such that new nightly packages can be built from each Git repository's *master* branch, to be then uploaded to the nightly repositories.

All scripts used by CI are stored in the Git repo adm-scripts.

Release Procedures

Table of Contents

30.1 Introduction

Kurento as a project spans across a multitude of different technologies and languages, each of them with their sets of conventions and *best practices*. This document aims to summarize all release procedures that apply to each one of the modules that compose the Kurento project. The main form of categorization is by technology type: C/C++ based modules, Java modules, JavaScript modules, and others.

30.2 General considerations

- Lists of projects in this document are sorted according to the repository lists given in *Code repositories*.

- Kurento projects to be released have supposedly been under development, and will have development version numbers:

  - In Java (Maven) projects, development versions are indicated by the suffix `-SNAPSHOT` after the version number. Example: `6.9.1-SNAPSHOT`.

  - In C/C++ (CMake) projects, development versions are indicated by the suffix `-dev` after the version number. Example: `6.9.1-dev`.

  These suffixes must be removed for release, and then recovered again to resume development.

- 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.

- Tags are named with the version number of the release. Example: `6.9.0`.

- Contrary to the project version, the Debian package versions don't contain development suffixes, and should always be of the form `1.2.3-0kurento1`:

  - The first part (*1.2.3*) is the project's **base version number**.

  - The second part (*0kurento1*) is the **Debian package revision**:

    * The number prefix (in this example: *0*) indicates the version relative to other same-name packages provided by the base system. When this number is *0*, it means that the package is original and only exists in Kurento, not in Debian or Ubuntu itself. This is typically the case for the projects owned or forked for the Kurento project.

    * The number suffix (in this example: *1*) means the number of times the same package has been re-packaged and re-published. *1* means that this is the *first* time a given project version was packaged.

  Example: Imagine that version *1.2.3* of your code is released for the first time. The full Debian package version will be: *1.2.3-0kurento1*. Then you realize the package doesn't install correctly in some machines, because of a bug in the package's post-install script. You fix it, and now it's time to re-publish the fixed package! But the project's source code itself has not changed at all, so it is not correct to increase the base version number: its version should still be *1.2.3*. The only changes have occurred in the packaging code itself (in `/debian/` dir). For this reason, the new package's full version will be *1.2.3-0kurento2*.

  Please check the Debian Policy Manual and this Ask Ubuntu answer for more information about the package versions.

- Kurento uses Semantic Versioning. Whenever you need to decide what is going to be the *final release version* for a new release, try to follow the SemVer guidelines:

```
Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
```

```
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.
```

Please refer to https://semver.org/ for more information.

Example

If the last Kurento release was **6.8.2** (with for example Debian package version *6.8.2-0kurento3*, because maybe the package itself had bugs, so it has been published 3 times) then after release the project versions should have been left as **6.8.3-dev** (or *6.8.3-SNAPSHOT* for Java components).

If the **next release** of Kurento only includes patches, then the next version number *6.8.3* is already good. However, maybe our release includes new functionality, which according to Semantic Versioning should be accompanied with a bump in the *minor* version number, so the next release version number should be *6.9.0*. The Debian package version is reset accordingly, so the full Debian version is **6.9.0-0kurento1**.

If you are re-packaging an already published version, without changes in the project's code itself, then just increment the Debian package revision: *0kurento1* becomes *0kurento2*, and so on.

Note: Made a mistake? Don't panic!

Do not be afraid of applying some Git magic to solve mistakes during the release process. Here are some which can be useful:

- How to remove a release tag?

    - Remove the local tag:

        ```
        git tag --delete <TagName>
        ```

    - Remove the remote tag:

        ```
        git push --delete origin <TagName>
        ```

- How to push just a local tag?

    ```
    git push origin <TagName>
    ```

- How to amend a commit and push it again?

    See: https://www.atlassian.com/git/tutorials/rewriting-history#git-commit--amend

    ```
    # <Remove Tag>
    # <Amend>
    # <Create Tag>
    git push --force origin <TagName>
    ```

Warning: As of this writing, there is a mix of methods in the CI scripts (adm-scripts) when it comes to handle the release versions. The instructions in this document favor creating and pushing git tags manually in the developer's computer, however some projects also make use of the script *kurento_check_version.sh*, which tries to detect when a project's version is *not* a development snapshot, then creates and pushes a git tag automatically. However if the tag already exists (created manually by the developer), then the `git tag` command fails, and this script prints a warning message before continuing with its work.

We've been toying with different methodologies between handling the tags automatically in CI or handling them manually by the developer before releasing new versions; both of these methods have pros and cons. For example,

if tags are handled manually by the developer, solving mistakes in the release process becomes simpler because there are no surprises from CI creating tags inadvertently; on the other hand, leaving them to be created by CI seems to simplify a bit the release process, but not really by a big margin.

30.3 Fork Repositories

This graph shows the dependencies between forked projects used by Kurento:
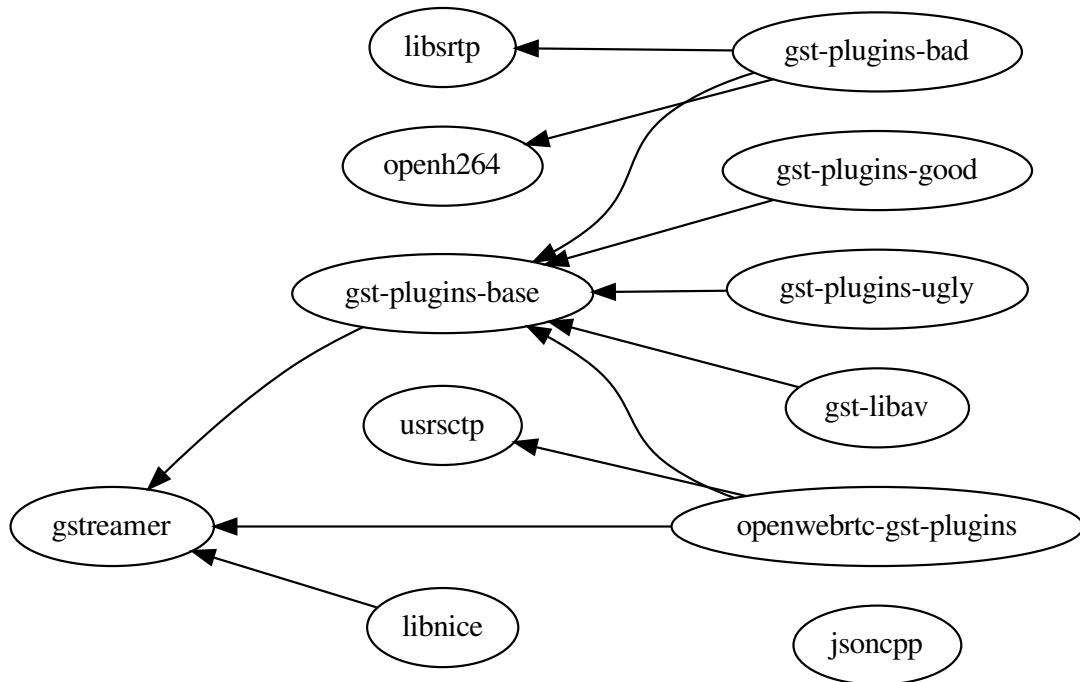
Fig. 1: Projects forked by Kurento

Release order:

- jsoncpp
- libsrtp
- openh264
- openh264-gst-plugin
- libusrsctp
- gstreamer
- gst-plugins-base
- gst-plugins-good

- gst-plugins-bad

- gst-plugins-ugly

- gst-libav

- openwebrtc-gst-plugins

- libnice

For each project above:

1. Prepare release.

2. Push a new tag to Git.

3. Move to next development version.

30.3.1 Release steps

1. Decide what is going to be the *final release version*. For this, follow the upstream version and the SemVer guidelines, as explained above in *General considerations*.

2. Set the final release version, commit the results, and create a tag.

```
# Change these
NEW_VERSION="<ReleaseVersion>"        # Eg.: 1.0.0
NEW_DEBIAN="<DebianRevision>"         # Eg.: 0kurento1

function do_release {
    local PACKAGE_VERSION="${NEW_VERSION}-${NEW_DEBIAN}"
    local COMMIT_MSG="Prepare release $PACKAGE_VERSION"

    local SNAPSHOT_ENTRY="* UNRELEASED"
    local RELEASE_ENTRY="* $COMMIT_MSG"

    gbp dch \
        --ignore-branch \
        --git-author \
        --spawn-editor=never \
        --new-version="$PACKAGE_VERSION" \
        \
        --release \
        --distribution="testing" \
        --force-distribution \
        \
        ./debian \
    || { echo "ERROR: Command failed: gbp dch"; return 1; }

    # First appearance of "UNRELEASED": Put our commit message
    sed -i "0,/${SNAPSHOT_ENTRY}/{s/${SNAPSHOT_ENTRY}/${RELEASE_ENTRY}/}" \
        ./debian/changelog \
    || { echo "ERROR: Command failed: sed"; return 2; }

    # Remaining appearances of "UNRELEASED" (if any): Delete line
    sed -i "/${SNAPSHOT_ENTRY}/d" \
        ./debian/changelog \
    || { echo "ERROR: Command failed: sed"; return 3; }

    git add debian/changelog \
```

(continues on next page)

```
    && git commit -m "$COMMIT_MSG" \
    && git push \
    && git tag -a -m "$COMMIT_MSG" "$PACKAGE_VERSION" \
    && git push origin "$PACKAGE_VERSION" \
    || { echo "ERROR: Command failed: git"; return 4; }

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

3. Follow on with releasing Kurento Media Server.

4. **AFTER THE WHOLE RELEASE HAS BEEN COMPLETED**: Set the next development version in all projects. To choose the next version number, increment the **Debian revision** number.

   The version number will only be changed when the fork gets updated from upstream code. When doing that, then change our version number so it matched the version number used in upstream.

```
# Change these
NEW_VERSION="<NextVersion>"          # Eg.: 1.0.0
NEW_DEBIAN="<NextDebianRevision>"    # Eg.: 0kurento2

function do_release {
    local PACKAGE_VERSION="${NEW_VERSION}-${NEW_DEBIAN}"
    local COMMIT_MSG="Bump development version to $PACKAGE_VERSION"

    gbp dch \
        --ignore-branch \
        --git-author \
        --spawn-editor=never \
        --new-version="$PACKAGE_VERSION" \
        ./debian \
    || { echo "ERROR: Command failed: gbp dch"; return 1; }

    git add debian/changelog \
    && git commit -m "$COMMIT_MSG" \
    && git push \
    || { echo "ERROR: Command failed: git"; return 2; }

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

30.4 Kurento Maven Plugin

1. Decide what is going to be the *final release version*. For this, follow the SemVer guidelines, as explained above in *General considerations*.

2. Set the final release version in *pom.xml*. Remove -SNAPSHOT.

```
    <groupId>org.kurento</groupId>
    <artifactId>kurento-maven-plugin</artifactId>
-   <version>6.9.0-SNAPSHOT</version>
+   <version>6.9.0</version>
```

3. Set the correct kurento-module-creator version in *pom.xml*.

```
    <groupId>org.kurento</groupId>
    <artifactId>kurento-module-creator</artifactId>
-   <version>6.8.2</version>
+   <version>6.9.0</version>
```

4. Git add, commit, tag, and push.

```
# Change this
NEW_VERSION="<ReleaseVersion>"        # Eg.: 1.0.0

function do_release {
    local COMMIT_MSG="Prepare release $NEW_VERSION"

    git add pom.xml \
    && git commit -m "$COMMIT_MSG" \
    && git push \
    && git tag -a -m "$COMMIT_MSG" "$NEW_VERSION" \
    && git push origin "$NEW_VERSION" \
    || echo "ERROR: Command failed: git"

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

5. The CI job should start automatically; some tests are run as a result of this commit, so you should wait for their completion.

   WARNING: This depends on Kurento Module Creator, which is not built until Kurento Media Server is done! So we have here a dependency loop that needs to be solved by hand: first force running the job for the Module Creator, then the job for Maven Plugin, and lastly the jobs for the Media Server.

6. [TODO: Sonatype]

7. **AFTER THE WHOLE RELEASE HAS BEEN COMPLETED**: Set the next development version in all projects. To choose the next version number, increment the **patch** number and add -SNAPSHOT.

```
    <groupId>org.kurento</groupId>
    <artifactId>kurento-maven-plugin</artifactId>
-   <version>6.9.0</version>
+   <version>6.9.1-SNAPSHOT</version>
```

6. Git add, commit, and push.

```
COMMIT_MSG="Prepare for next development iteration"

cd kurento-maven-plugin
git add pom.xml \
&& git commit -m "$COMMIT_MSG" \
```

```
&& git push \
|| echo "ERROR: Command failed: git"
```

## 30.5 Kurento Media Server

All KMS projects:
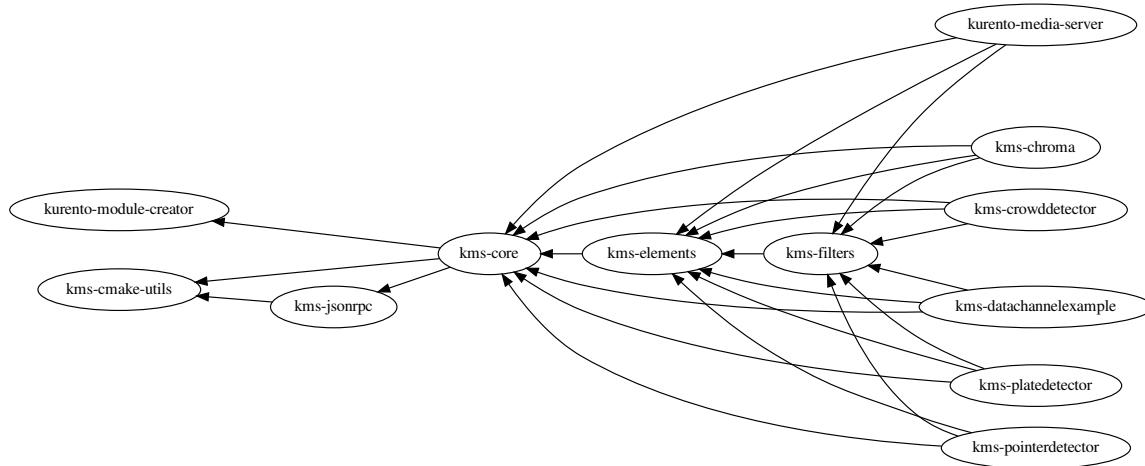
Fig. 2: Projects that are part of Kurento Media Server

Release order:

- kurento-module-creator

- kms-cmake-utils

- kms-jsonrpc

- kms-core

- kms-elements

- kms-filters

- kurento-media-server

- kms-chroma

- kms-crowddetector

- kms-datachannelexample

- kms-markerdetector

- kms-platedetector

- kms-pointerdetector

For each project above:

1. Prepare release.

2. Push a new tag to Git.

3. Move to next development version.

30.5.1 Preparation: Kurento Module Creator

If **kurento-maven-plugin** is going to get also a new release, then edit the file `kurento-module-creator/src/main/templates/maven/model_pom_xml.ftl` to update the plugin version in the auto-generation template:

```
    <groupId>org.kurento</groupId>
    <artifactId>kurento-maven-plugin</artifactId>
-   <version>6.8.2</version>
+   <version>6.9.0</version>
```

Then, proceed wih the normal release:

30.5.2 Preparation: KMS API Java modules

Test the KMS API Java module generation (local check).

```
apt-get update && apt-get install --no-install-recommends --yes \
    kurento-module-creator \
    kms-cmake-utils \
    kms-jsonrpc-dev \
    kms-core-dev \
    kms-elements-dev \
    kms-filters-dev

cd kms-omni-build

function do_release {
    local PROJECTS=(
        kms-core
        kms-elements
        kms-filters
    )

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 1; }

        mkdir build \
        && cd build \
        && cmake .. -DGENERATE_JAVA_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_
↪GENERATION=TRUE \
        && cd java \
        && mvn clean install -Dmaven.test.skip=false \
        || { echo "ERROR: Command failed"; return 1; }

        popd
    done

    echo "Done!"
}
```

(continues on next page)

```
# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

30.5.3 Release steps

1. Decide what is going to be the *final release version*. For this, follow the SemVer guidelines, as explained above in *General considerations*.

2. Set the final release version in all projects. Use the helper script kms-omni-build/bin/set-versions.sh to set version numbers, commit the results, and create a tag.

```
# Change these
NEW_VERSION="<ReleaseVersion>"        # Eg.: 1.0.0
NEW_DEBIAN="<DebianRevision>"         # Eg.: 0kurento1

cd kms-omni-build
./bin/set-versions.sh "$NEW_VERSION" --debian "$NEW_DEBIAN" \
    --release --commit --tag
```

Now push changes:

```
git submodule foreach 'git push --follow-tags'
```

3. Update the git-submodule references of the all-in-one repo *kms-omni-build*, and create a tag just like in all the other repos.

```
# Change this
NEW_VERSION="<ReleaseVersion>"      # Eg.: 1.0.0

function do_release {
    local COMMIT_MSG="Prepare release $NEW_VERSION"

    git add . \
    && git commit -m "$COMMIT_MSG" \
    && git push \
    && git tag -a -m "$COMMIT_MSG" "$NEW_VERSION" \
    && git push origin "$NEW_VERSION"

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

4. Start the KMS CI job with the parameters *JOB_RELEASE* **ENABLED** and *JOB_ONLY_KMS* **DISABLED**.

5. Wait until all packages get created and published correctly. Fix any issues that might appear.

Note: The KMS CI job is a *Jenkins MultiJob Project*. If it fails at any stage, after fixing the cause of the error there is no need to start the job again from the beginning. Instead, you can resume the build from the point it was before the failure.

For this, just open the latest build number that failed (with a red marker in the *Build History* panel at the left of the job page); in the description of the build, the action *Resume build* is available on the left side.

---

6. Check that the Auto-Generated API Client JavaScript repos have been updated (which should happen as part of the CI jobs for all Kurento Media Server modules that contain KMD API Definition files, `*.kmd`):

   - kms-core -> kurento-client-core-js

   - kms-elements -> kurento-client-elements-js

   - kms-filters -> kurento-client-filters-js

   - kms-chroma -> kurento-module-chroma-js

   - kms-crowddetector -> kurento-module-crowddetector-js

   - kms-datachannelexample -> kurento-module-datachannelexample-js

   - kms-markerdetector -> kurento-module-markerdetector-js

   - kms-platedetector -> kurento-module-platedetector-js

   - kms-pointerdetector -> kurento-module-pointerdetector-js

7. When all repos have been released, and CI jobs have finished successfully, publish the Java artifacts:

   - Open the Nexus Sonatype Staging Repositories section.

   - Select **kurento** repository.

   - Inspect **Content** to ensure they are as expected:

     - kurento-module-creator

     - kms-api-core

     - kms-api-elements

     - kms-api-filters

     - All of them must appear in the correct version, `$NEW_VERSION`.

   - **Close** repository.

   - Wait a bit.

   - **Refresh**.

   - **Release** repository.

   - Maven artifacts will be available after 10 minutes.

8. Also, check that the JavaScript modules have been published by CI:

   - Open each module's page in NPM, and check that the latest version corresponds to the current release:

     - NPM: kurento-client-core

     - NPM: kurento-client-elements

     - NPM: kurento-client-filters

   - If any of these are missing, it's probably due to the CI job not running (because the project didn't really contain any code difference from the previous version... happens sometimes when not all repos have changed since the last release). Open CI and run the jobs manually:

     - CI: kurento_client_core_js_merged

     - CI: kurento_client_elements_js_merged

     - CI: kurento_client_filters_js_merged

9. **AFTER THE WHOLE RELEASE HAS BEEN COMPLETED**: Set the next development version in all projects. To choose the next version number, reset the **Debian revision** number to *1*, and increment the **patch** number. Use the helper script `kms-omni-build/bin/set-versions.sh` to set version numbers and commit.

```
# Change these
NEW_VERSION="<NextVersion>"          # Eg.: 1.0.1
NEW_DEBIAN="<NextDebianRevision>"     # Eg.: 0kurento1

cd kms-omni-build
./bin/set-versions.sh "$NEW_VERSION" --debian "$NEW_DEBIAN" \
    --new-development --commit
```

Now push changes:

```
git submodule foreach 'git push'
```

10. Start the KMS CI job with the parameters *JOB_RELEASE* **DISABLED** and *JOB_ONLY_KMS* **DISABLED**.

30.6 Kurento JavaScript client

Release order:

- kurento-jsonrpc-js
- kurento-utils-js
- kurento-client-js
- kurento-tutorial-js
- kurento-tutorial-node

For each project above:

1. Prepare release.
2. Push a new tag to Git.
3. Move to next development version.

30.6.1 Release steps

1. Decide what is going to be the *final release version*. For this, follow the SemVer guidelines, as explained above in *General considerations*.

2. Ensure there are no uncommitted files.

```
git diff-index --quiet HEAD \
|| echo "ERROR: Uncommitted files not allowed!"
```

3. Set the final release version in project and dependencies. This operation is done in different files, depending on the project:

- `kurento-jsonrpc-js/package.json`
- `kurento-utils-js/package.json`
- `kurento-client-js/package.json`

- Each one in `kurento-tutorial-js/**/bower.json`
- Each one in `kurento-tutorial-node/**/package.json`

4. Review all dependencies to remove development versions.

   This command can be used to search for all development versions:

   ```
   grep . --exclude-dir='*node_modules' -Fr -e '-dev"' -e '"git+' \
   && echo "ERROR: Development versions not allowed!"
   ```

   For example: All dependencies to Kurento packages that point directly to their Git repos should be changed to point to a pinned SemVer number (or version range). Later, the Git URL can be restored for the next development iteration.

5. Test the build, to make sure the code is in a working state.

   ```
   npm install
   if [[ -x node_modules/.bin/grunt ]]; then
       node_modules/.bin/grunt jsbeautifier \
       && node_modules/.bin/grunt \
       && node_modules/.bin/grunt sync:bower \
       || echo "ERROR: Command failed: npm"
   fi
   ```

   To manually run the beautifier, do this:

   ```
   npm install

   # To run beautifier over all files, modifying in-place:
   node_modules/.bin/grunt jsbeautifier::default

   # To run beautifier over a specific file:
   node_modules/.bin/grunt jsbeautifier::file:<FilePath>.js
   ```

   Some times it happens that Grunt needs to be run a couple of times until it ends without errors.

6. **All-In-One** script:

   Note: You'll need to install the **jq** command-line JSON processor.

   ```
   # Change this
   NEW_VERSION="<ReleaseVersion>"        # Eg.: 1.0.0

   function do_release {
       local COMMIT_MSG="Prepare release $NEW_VERSION"

       local PROJECTS=(
           kurento-jsonrpc-js
           kurento-utils-js
           kurento-client-js
           kurento-tutorial-js
           kurento-tutorial-node
       )

       for PROJECT in "${PROJECTS[@]}"; do
           pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 1; }
   ```

   (continues on next page)

```
        # Ensure there are no uncommitted files
        git diff-index --quiet HEAD \
        || { echo "ERROR: Uncommitted files not allowed!"; return 2; }

        git pull --rebase \
        || { echo "ERROR: Command failed: git pull"; return 3; }

        # Set the final release version in project and dependencies
        JQ_PROGRAM="$(mktemp)"
        tee "$JQ_PROGRAM" >/dev/null <<EOF
# This is a program for the "jq" command-line JSON processor.
# Rules 2, 3, 4, 5 are for kurento-client-js;
# Rule 6 is for kurento-tutorial-node;
# Rules 7, 8, 9, 10, 11 are for kurento-tutorial-js.
if .version? then
    .version = "$NEW_VERSION"
else . end
| if .dependencies."kurento-client-core"? then
    .dependencies."kurento-client-core" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-client-elements"? then
    .dependencies."kurento-client-elements" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-client-filters"? then
    .dependencies."kurento-client-filters" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-jsonrpc"? then
    .dependencies."kurento-jsonrpc" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-client"? then
    .dependencies."kurento-client" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-utils"? then
    .dependencies."kurento-utils" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-module-chroma"? then
    .dependencies."kurento-module-chroma" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-module-crowddetector"? then
    .dependencies."kurento-module-crowddetector" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-module-platedetector"? then
    .dependencies."kurento-module-platedetector" = "$NEW_VERSION"
else . end
| if .dependencies."kurento-module-pointerdetector"? then
    .dependencies."kurento-module-pointerdetector" = "$NEW_VERSION"
else . end
EOF
        find . -path '*node_modules' -prune , -name '*.json' | while read FILE; do
            echo "Process file: $(realpath "$FILE")"
            TEMP="$(mktemp)"
            jq --from-file "$JQ_PROGRAM" "$FILE" >"$TEMP" \
            && mv "$TEMP" "$FILE" \
            || { echo "ERROR: Command failed: jq"; return 4; }

            git add "$FILE"
        done
```

```
        # Review all dependencies to remove development versions
        grep . --exclude-dir='*node_modules' -Fr -e '-dev"' -e '"git+' \
        && { echo "ERROR: Development versions not allowed!"; return 5; }

        # Test the build
        if [[ "$PROJECT" == "kurento-client-js" ]]; then
            # kurento-client-js depends on kurento-jsonrpc-js, so we'll use
            # `npm link` here to solve the dependency.
            # Use a custom Node prefix so `npm link` doesn't require root␣
→permissions.
            NPM_CONFIG_PREFIX=.npm npm link ../kurento-jsonrpc-js
        fi
        npm install || { echo "ERROR: Command failed: npm install"; return 6; }
        if [[ -x node_modules/.bin/grunt ]]; then
            node_modules/.bin/grunt jsbeautifier \
            && node_modules/.bin/grunt \
            && node_modules/.bin/grunt sync:bower \
            || { echo "ERROR: Command failed: grunt"; return 7; }
        fi

        popd
    done

    echo "Everything seems OK; proceed to commit and push"

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 8; }

        git commit -m "$COMMIT_MSG" \
        && git push \
        || { echo "ERROR: Command failed: git"; return 9; }

        # && git tag -a -m "$COMMIT_MSG" "$NEW_VERSION" \
        # && git push origin "$NEW_VERSION" \
        # NOTE: the CI jobs automatically tag the repos upon releases

        popd
    done

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

7. When all repos have been released, and CI jobs have finished successfully,

   - Open the Nexus Sonatype Staging Repositories section.

   - Select **kurento** repository.

   - Inspect **Content** to ensure they are as expected:

     – kurento-jsonrpc-js

     – kurento-utils-js

     – kurento-client-js

> – All of them must appear in the correct version, `$NEW_VERSION`.

- **Close** repository.

- Wait a bit.

- **Refresh**.

- **Release** repository.

- Maven artifacts will be available after 10 minutes.

8. **AFTER THE WHOLE RELEASE HAS BEEN COMPLETED**: Set the next development version in all projects. To choose the next version number, increment the **patch** number and add `-dev`.

All-In-One script:

```
# Change this
NEW_VERSION="<NextVersion>-dev"          # Eg.: 1.0.1-dev

function do_release {
    local COMMIT_MSG="Prepare for next development iteration"

    local PROJECTS=(
        kurento-jsonrpc-js
        kurento-utils-js
        kurento-client-js
        kurento-tutorial-js
        kurento-tutorial-node
    )

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 1; }

        git pull --rebase \
        || { echo "ERROR: Command failed: git pull"; return 2; }

        # Set the next development version in project and dependencies
        JQ_PROGRAM="$(mktemp)"
        tee "$JQ_PROGRAM" >/dev/null <<EOF
# This is a program for the "jq" command-line JSON processor.
# Rules 2, 3, 4, 5 are for kurento-client-js;
# Rule 6 is for kurento-tutorial-node;
# Rules 7, 8, 9, 10, 11 are for kurento-tutorial-js.
if .version? then
    .version = "$NEW_VERSION"
else . end
| if .dependencies."kurento-client-core"? then
    .dependencies."kurento-client-core" = "git+https://github.com/Kurento/kurento-
→client-core-js.git"
else . end
| if .dependencies."kurento-client-elements"? then
    .dependencies."kurento-client-elements" = "git+https://github.com/Kurento/
→kurento-client-elements-js.git"
else . end
| if .dependencies."kurento-client-filters"? then
    .dependencies."kurento-client-filters" = "git+https://github.com/Kurento/
→kurento-client-filters-js.git"
else . end
| if .dependencies."kurento-jsonrpc"? then
```

(continues on next page)

```
    .dependencies."kurento-jsonrpc" = "git+https://github.com/Kurento/kurento-
↪jsonrpc-js.git"
else . end
| if .dependencies."kurento-client"? then
    .dependencies."kurento-client" = "git+https://github.com/Kurento/kurento-
↪client-js.git"
else . end
| if .dependencies."kurento-utils"? then
    .dependencies."kurento-utils" = "git+https://github.com/Kurento/kurento-utils-
↪js.git"
else . end
| if .dependencies."kurento-module-chroma"? then
    .dependencies."kurento-module-chroma" = "git+https://github.com/Kurento/
↪kurento-module-chroma-js.git"
else . end
| if .dependencies."kurento-module-crowddetector"? then
    .dependencies."kurento-module-crowddetector" = "git+https://github.com/
↪Kurento/kurento-module-crowddetector-js.git"
else . end
| if .dependencies."kurento-module-platedetector"? then
    .dependencies."kurento-module-platedetector" = "git+https://github.com/
↪Kurento/kurento-module-platedetector-js.git"
else . end
| if .dependencies."kurento-module-pointerdetector"? then
    .dependencies."kurento-module-pointerdetector" = "git+https://github.com/
↪Kurento/kurento-module-pointerdetector-js.git"
else . end
EOF
        find . -path '*node_modules' -prune , -name '*.json' | while read FILE; do
            echo "Process file: $(realpath "$FILE")"
            TEMP="$(mktemp)"
            jq --from-file "$JQ_PROGRAM" "$FILE" >"$TEMP" \
            && mv "$TEMP" "$FILE" \
            || { echo "ERROR: Command failed: jq"; return 3; }

            git add "$FILE"
        done

        popd
    done

    echo "Everything seems OK; proceed to commit and push"

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 4; }

        git commit -m "$COMMIT_MSG" \
        && git push \
        || { echo "ERROR: Command failed: git"; return 5; }

        popd
    done

    echo "Done!"
}

# Run in a subshell where all commands are traced
```

```
(set -o xtrace; do_release)
```

30.7 Kurento Java client

Release order:

- kurento-qa-pom

- kurento-java

- kurento-tutorial-java

- kurento-tutorial-test

For each project above:

1. Prepare release.

2. Push a new tag to Git.

3. Move to next development version.

30.7.1 Preparation: kurento-java

If there have been changes in the API of Kurento Media Server modules (in the *.kmd* JSON files), update the corresponding versions in kurento-parent-pom/pom.xml:

```
    <properties>
-   <version.kms-api-core>6.8.2</version.kms-api-core>
-   <version.kms-api-elements>6.8.2</version.kms-api-elements>
-   <version.kms-api-filters>6.8.2</version.kms-api-filters>
+   <version.kms-api-core>6.9.0</version.kms-api-core>
+   <version.kms-api-elements>6.9.0</version.kms-api-elements>
+   <version.kms-api-filters>6.9.0</version.kms-api-filters>
```

Doing this ensures that the Java client gets generated according to the latest versions of the API definitions.

Similarly, update the version numbers of any other Kurento project that has been updated:

```
<version.kurento-utils-js>6.9.0</version.kurento-utils-js>
<version.kurento-maven-plugin>6.9.0</version.kurento-maven-plugin>

<version.kurento-chroma>6.9.0</version.kurento-chroma>
<version.kurento-crowddetector>6.9.0</version.kurento-crowddetector>
<version.kurento-markerdetector>6.9.0</version.kurento-markerdetector>
<version.kurento-platedetector>6.9.0</version.kurento-platedetector>
<version.kurento-pointerdetector>6.9.0</version.kurento-pointerdetector>
```

30.7.2 Release steps

1. Decide what is going to be the *final release version*. For this, follow the SemVer guidelines, as explained above in *General considerations*.

2. Ensure there are no uncommitted files.

```
git diff-index --quiet HEAD \
|| echo "ERROR: Uncommitted files not allowed!"
```

3. Set the final release version in project and dependencies. This operation varies between projects.

Note: Order matters. *kurento-tutorial-java* and *kurento-tutorial-test* require that *kurento-java* has been installed locally (with `mvn install`) before being able to change their version numbers programmatically with Maven.

---

4. Review all dependencies to remove development versions.

Note: In *kurento-java*, all dependencies are defined as properties in the file `kurento-parent-pom/pom.xml`.

This command can be used to search for all development versions:

```
grep . --include='pom.xml' -Fr -e '-SNAPSHOT' \
&& echo "ERROR: Development versions not allowed!"
```

5. Test the build, to make sure the code is in a working state.

Note: The profile '*kurento-release*' is used to enforce no development versions are present.

```
mvn -U clean install -Dmaven.test.skip=false -Pkurento-release \
|| echo "ERROR: Command failed: mvn clean install"
```

6. **All-In-One** script:

Note: Use `mvn --batch-mode` if you copy this to an actual script.

```
# Change this
NEW_VERSION="<ReleaseVersion>"        # Eg.: 1.0.1
KMS_VERSION="<KmsApiVersion>"         # Eg.: 1.0.0

function do_release {
    local COMMIT_MSG="Prepare release $NEW_VERSION"

    local PROJECTS=(
        kurento-qa-pom
        kurento-java
        kurento-tutorial-java

        # FIXME tests fail because Kurento Test Framework needs improvements
        # kurento-tutorial-test
    )

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 1; }

        # Ensure there are no uncommitted files
        git diff-index --quiet HEAD \
```

(continues on next page)

```
        || { echo "ERROR: Uncommitted files not allowed!"; return 2; }

        git pull --rebase \
        || { echo "ERROR: Command failed: git pull"; return 3; }

        # Set the final release version in project and dependencies
        if [[ "$PROJECT" == "kurento-qa-pom" ]]; then
            mvn \
                versions:set \
                -DgenerateBackupPoms=false \
                -DnewVersion="$NEW_VERSION" \
            || { echo "ERROR: Command failed: mvn versions:set"; return 4; }
        elif [[ "$PROJECT" == "kurento-java" ]]; then
            mvn --file kurento-parent-pom/pom.xml \
                versions:set \
                -DgenerateBackupPoms=false \
                -DnewVersion="$NEW_VERSION" \
            || { echo "ERROR: Command failed: mvn versions:set"; return 5; }

            mvn --file kurento-parent-pom/pom.xml \
                versions:set-property \
                -DgenerateBackupPoms=false \
                -Dproperty=version.kms-api-core \
                -DnewVersion="$KMS_VERSION"
            mvn --file kurento-parent-pom/pom.xml \
                versions:set-property \
                -DgenerateBackupPoms=false \
                -Dproperty=version.kms-api-elements \
                -DnewVersion="$KMS_VERSION"
            mvn --file kurento-parent-pom/pom.xml \
                versions:set-property \
                -DgenerateBackupPoms=false \
                -Dproperty=version.kms-api-filters \
                -DnewVersion="$KMS_VERSION"
        elif [[ "$PROJECT" == "kurento-tutorial-java" || "$PROJECT" == "kurento-
→tutorial-test" ]]; then
            mvn \
                versions:update-parent \
                -DgenerateBackupPoms=false \
                -DparentVersion="[${NEW_VERSION}]" \
            || { echo "ERROR: Command failed: mvn versions:update-parent"; return␣
→6; }

            mvn -N \
                versions:update-child-modules \
                -DgenerateBackupPoms=false \
            || { echo "ERROR: Command failed: mvn versions:update-child-modules";␣
→return 7; }
        fi

        # Review all dependencies to remove development versions
        grep . --include='pom.xml' -Fr -e '-SNAPSHOT' \
        && echo "ERROR: Development versions not allowed!"

        # Test the build
        mvn -U clean install -Dmaven.test.skip=false -Pkurento-release \
        || { echo "ERROR: Command failed: mvn clean install"; return 8; }
```

```
        popd
    done

    echo "Everything seems OK; proceed to commit and push"

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 9; }

        ( git ls-files --modified | grep -E '/?pom.xml$' | xargs -r git add ) ||␣
↪true \
        && git commit -m "$COMMIT_MSG" \
        && git push \
        && git tag -a -m "$COMMIT_MSG" "$NEW_VERSION" \
        && git push origin "$NEW_VERSION" \
        || { echo "ERROR: Command failed: git"; return 10; }

        popd
    done

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

7. When all repos have been released, and CI jobs have finished successfully:

   - Open the Nexus Sonatype Staging Repositories section.

   - Select **kurento** repositories.

   - Inspect **Content** to ensure they are as expected: *kurento-java*, etc.

   - **Close repositories**.

   - Wait a bit.

   - **Refresh**.

   - **Release repositories**.

   - Maven artifacts will be available after 10 minutes.

   - Open the Nexus Sonatype Staging Repositories section.

   - Select **kurento** repository.

   - Inspect **Content** to ensure they are as expected:

     – kurento-client

     – kurento-commons

     – kurento-integration-tests

     – kurento-java

     – kurento-jsonrpc

     – kurento-jsonrpc-client

     – kurento-jsonrpc-client-jetty

- – kurento-jsonrpc-server

- – kurento-parent-pom

- – kurento-repository (ABANDONED)

- – kurento-repository-client (ABANDONED)

- – kurento-repository-internal (ABANDONED)

- – kurento-test

- – All of them must appear in the correct version, `$NEW_VERSION`.

- • **Close** repository.

- • Wait a bit.

- • **Refresh**.

- • **Release** repository.

- • Maven artifacts will be available after 10 minutes.

8. **AFTER THE WHOLE RELEASE HAS BEEN COMPLETED**: Set the next development version in all projects. To choose the next version number, increment the **patch** number and add `-SNAPSHOT`.

Note: Maven can do this automatically with the Maven Versions Plugin.

Note: You should wait for a full nightly run of the Kurento Media Server pipeline, so the next development packages become available from KMS API modules: *kms-api-core*, *kms-api-elements*, and *kms-api-filters*. This way, the properties in `kurento-parent-pom/pom.xml` will get updated to the latest SNAPSHOT version.

All-In-One script:

Note: Use `mvn --batch-mode` if you copy this to an actual script.

```
function do_release {
    local COMMIT_MSG="Prepare for next development iteration"

    local PROJECTS=(
        kurento-qa-pom
        kurento-java
        kurento-tutorial-java
        kurento-tutorial-test
    )

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 1; }

        # Set the next development version in project and dependencies
        if [[ "$PROJECT" == "kurento-qa-pom" ]]; then
            mvn \
                versions:set \
                -DgenerateBackupPoms=false \
                -DnextSnapshot=true \
            || { echo "ERROR: Command failed: mvn versions:set"; return 2; }
```

(continues on next page)

```
        elif [[ "$PROJECT" == "kurento-java" ]]; then
            mvn --file kurento-parent-pom/pom.xml \
                versions:set \
                -DgenerateBackupPoms=false \
                -DnextSnapshot=true \
            || { echo "ERROR: Command failed: mvn versions:set"; return 3; }

            mvn --file kurento-parent-pom/pom.xml \
                versions:update-property \
                -DgenerateBackupPoms=false \
                -Dproperty=version.kms-api-core \
                -DallowSnapshots=true
            mvn --file kurento-parent-pom/pom.xml \
                versions:update-property \
                -DgenerateBackupPoms=false \
                -Dproperty=version.kms-api-elements \
                -DallowSnapshots=true
            mvn --file kurento-parent-pom/pom.xml \
                versions:update-property \
                -DgenerateBackupPoms=false \
                -Dproperty=version.kms-api-filters \
                -DallowSnapshots=true
        else # kurento-tutorial-java, kurento-tutorial-test
            mvn \
                versions:update-parent \
                -DgenerateBackupPoms=false \
                -DallowSnapshots=true \
            || { echo "ERROR: Command failed: mvn versions:update-parent"; return
→4; }

            mvn -N \
                versions:update-child-modules \
                -DgenerateBackupPoms=false \
                -DallowSnapshots=true \
            || { echo "ERROR: Command failed: mvn versions:update-child-modules";
→return 5; }
        fi

        # Test the build
        mvn -U clean install -Dmaven.test.skip=true \
        || { echo "ERROR: Command failed: mvn clean install"; return 6; }

        popd
    done

    echo "Everything seems OK; proceed to commit and push"

    for PROJECT in "${PROJECTS[@]}"; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 7; }

        ( git ls-files --modified | grep -E '/?pom.xml$' | xargs -r git add ) \
        && git commit -m "$COMMIT_MSG" \
        && git push \
        || { echo "ERROR: Command failed: git"; return 8; }

        popd
    done
```

(continued from previous page)

```
    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

30.8 Docker images

A new set of development images is deployed to Kurento Docker Hub on each nightly build. Besides, a release version will be published as part of the CI jobs chain when the KMS CI job is triggered.

The repository kurento-docker contains *Dockerfile*s for all the 'Kurento Docker images'_, however this repo shouldn't be tagged, because it is essentially a "multi-repo" and the tags would be meaningless (because *which one of the sub-dirs would the tag apply to?*).

30.9 Kurento documentation

The documentation scripts will download both Java and JavaScript clients, generate HTML Javadoc / Jsdoc pages from them, and embed everything into a static section.

For this reason, the documentation must be built only after all the other modules have been released.

1. Write the Release Notes in `doc-kurento/source/project/relnotes/`.

2. Ensure that the whole nightly CI chain works:

   Job *doc-kurento* -> job *doc-kurento-readthedocs* -> New build at ReadTheDocs.

3. Edit VERSIONS.conf.sh to set all relevant version numbers: version of the documentation itself, and all referred modules and client libraries.

   These numbers can be different because not all of the Kurento projects are necessarily released with the same frequency. Check each one of the Kurento repositories to verify what is the latest version of each one, and put it in the corresponding variable:

   - `[VERSION_KMS]`: Repo kurento-media-server.
   - `[VERSION_CLIENT_JAVA]`: Repo kurento-java.
   - `[VERSION_CLIENT_JS]`: Repo kurento-client-js.
   - `[VERSION_UTILS_JS]`: Repo kurento-utils-js.
   - `[VERSION_TUTORIAL_JAVA]`: Repo kurento-tutorial-java.
   - `[VERSION_TUTORIAL_JS]`: Repo kurento-tutorial-js.
   - `[VERSION_TUTORIAL_NODE]`: Repo kurento-tutorial-node.

4. In *VERSIONS.conf.sh*, set *VERSION_RELEASE* to *true*. Remember to set it again to *false* after the release, when starting a new development iteration.

5. Test the build locally, check everything works.

   ```
   make html
   ```

Note that the JavaDoc and JsDoc pages won't be generated locally if you don't have your system prepared to do so; also there are some Sphinx constructs or plugins that might fail if you don't have them ready to use, but the ReadTheDocs servers have them so they should end up working fine.

6. Git add, commit, and push. This will trigger a nightly build, where you can **check the result** of the documentation builds to have an idea of how the final release build will end up looking like, at https://doc-kurento.readthedocs.io/en/latest/.

```
# `--all` to include possibly deleted files.
git add --all \
    VERSIONS.conf.sh \
    source/project/relnotes/ \
&& git commit -m "$COMMIT_MSG" \
&& git push \
|| echo "ERROR: Command failed: git"
```

7. Run the doc-kurento CI job with the parameter *JOB_RELEASE* **ENABLED**.

8. CI automatically tags Release versions in both ReadTheDocs source repos doc-kurento and doc-kurento-readthedocs, so the release will show up in the ReadTheDocs dashboard.

Note: If you made a mistake and want to re-create the git tag with a different commit, remember that the re-tagging must be done manually in both *doc-kurento* and *doc-kurento-readthedocs* repos. ReadTheDocs CI servers will read the latter one to obtain the documentation sources and release tags.

---

9. Open ReadTheDocs Builds. If the new version hasn't been detected and built, do it manually: use the *Build Version* button to force a build of the *latest* version.

Doing this, ReadTheDocs will "realize" that there is a new tagged release version of the documentation, in the *doc-kurento-readthedocs* repo. After the build is finished, the new release version will be available for selection in the next step.

10. Open ReadTheDocs Advanced Settings and select the new version in the *Default Version* combo box.

Note: We don't set the *Default Version* field to "*stable*", because we want that the actual version number gets shown in the upper part of the side panel (below the Kurento logo, above the search box) when users open the documentation. If "*stable*" was selected here, then users would just see the word "*stable*" in the mentioned panel.

---

11. **AFTER THE WHOLE RELEASE HAS BEEN COMPLETED**: Set *VERSION_RELEASE* to *false*. Now, create a Release Notes document template where to write changes that will accumulate for the next release.

All-In-One script:

```
# Change this
NEW_VERSION="<NextVersion>"          # Eg.: 1.0.1

function do_release {
    local COMMIT_MSG="Prepare for next development iteration"

    # Set [VERSION_RELEASE]="false"
    sed -r -i 's/\[VERSION_RELEASE\]=.*/[VERSION_RELEASE]="false"/' VERSIONS.conf.
↪sh \
    || { echo "ERROR: Command failed: sed"; return 1; }

    # Add a new Release Notes document
```

```
    local RELNOTES_NAME="v${NEW_VERSION//./_}"
    cp source/project/relnotes/v0_TEMPLATE.rst \
        "source/project/relnotes/${RELNOTES_NAME}.rst" \
    && sed -i "s/1.2.3/${NEW_VERSION}/" \
        "source/project/relnotes/${RELNOTES_NAME}.rst" \
    && sed -i "8i\   $RELNOTES_NAME" \
        source/project/relnotes/index.rst \
    || { echo "ERROR: Command failed: sed"; return 1; }

    git add \
        VERSIONS.conf.sh \
        source/project/relnotes/ \
    && git commit -m "$COMMIT_MSG" \
    && git push \
    || { echo "ERROR: Command failed: git"; return 1; }

    echo "Done!"
}

# Run in a subshell where all commands are traced
(set -o xtrace; do_release)
```

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 enabled[1]. Ubuntu has also followed the same policy regarding their own build procedures[2].

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.

31.1 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
```

31.2 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.

31.3 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 (*libSHARED.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: *libSHARED.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` (and maybe linked with the shared library with `-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*.

31.4 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.

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 that 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.

32.1 Building locally

If you are writing documentation for Kurento, there is no need to commit every change and push to see the result only to discover that an image doesn't show up as you expected, or some small mistake breaks a table layout.

- First of all, it's a very good idea to use a text editor that provides spell checking and live-preview visualization of *reST* files; this alone will help catching most grammatical and syntactic mistakes. Visual Studio Code is a great option, it provides extensions for both of these things.

  To install *VS Code* and the two mentioned functions, use their installer, then run these commands:

  ```
  code --install-extension streetsidesoftware.code-spell-checker
  code --install-extension lextudio.restructuredtext
  ```

- Secondly, you can build the documentation locally. Just install the required dependencies:

  ```
  # Ensure that old versions of Sphinx are not installed
  sudo apt-get purge --auto-remove \
      python-sphinx \
      sphinx-common \
      python-sphinx-rtd-theme \
      sphinx-rtd-theme-common
  ```

  ```
  # Ensure that Sphinx is not installed with Python 3
  sudo pip3 uninstall sphinx
  sudo pip3 uninstall sphinx_rtd_theme
  sudo pip3 uninstall sphinxcontrib_websupport
  ```

  ```
  # Install Sphinx with Python 2
  sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
      python-pip \
      python-setuptools

  sudo pip2 install --upgrade sphinx sphinx_rtd_theme
  ```

  And then just run `make html` to build and open your new shiny files:

  ```
  cd doc-kurento/
  make html
  firefox build/html/index.html
  ```

32.2 Kurento documentation Style Guide

32.2.1 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.

32.2.2 Inline markup

- File names, package names, variable names, class and event names, (mostly all kinds of names), acronyms, commit hashes, and in general any kind of identifier which could be broken into different lines 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.
Enable debug by setting the *GST_DEBUG* environment variable.
```

- Paths, URLs, code samples, commands, and in general any machine-oriented keywords are emphasized with double back quotes (as in ``word``). This formatting stands out, and most importantly *it cannot be broken into different lines*. Sample phrases:

```
Use ``apt-get install`` to set up all required packages.
Set ``CMAKE_BUILD_TYPE=Debug`` to build with debug symbols.
The argument ``--gst-debug`` can be used to control the logging level.
```

- There is no difference between using *single asterisks* (*word*), and *single back quotes* (`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.

32.2.3 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
"""""""
```

32.3 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_KMS* keyword when substituted with its final value: 6.15.0.

  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 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.

32.4 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 latter repo, thus triggering a new RTD build.

Testing

Software testing is a broad term within software engineering encompassing a wide spectrum of different concepts. Depending on the size of the System Under Test (SUT) and the scenario in which it is exercised, testing can be carried out at different levels. There is no universal classification for all the different testing levels. Nevertheless, the following levels are broadly accepted:

- Unit: individual program units are tested. Unit tests typically focus on the functionality of individual objects or methods.

- Integration: units are combined to create composite components. Integration tests focus on the interaction of different units.

- System: all of the components are integrated and the system is tested as a whole.

There is a special type of system tests called **end-to-end** (E2E). In E2E tests, the final user is typically impersonated, i.e., simulated using automation techniques. The main benefit of E2E tests is the simulation of real user scenarios in an automated fashion. As described in the rest of this document, a rich variety of E2E has been implemented to assess Kurento.

33.1 E2E Tests

This section introduces the different types of E2E implemented to assess different parts of Kurento, namely **functional**, **stability**, **tutorials**, and **API**.

33.1.1 Functional

Functional tests are aimed to evaluate a given capability provided by Kurento. These tests have created in Java. You can find the source code in the repository kurento-test within kurento-java. In order to run functional tests, Maven should be used as follows:

```
git clone https://github.com/Kurento/kurento-java.git
cd kurento-java
```

475

```
mvn \
    --projects kurento-integration-tests/kurento-test --also-make \
    -Pintegration \
    -Dgroups=org.kurento.commons.testing.SystemFunctionalTests \
    clean verify
```

By default, these tests required a local Kurento Media Server installed in the machine running the tests. In addition, Chrome and Firefox browsers are also required. For further information about running these tests, please read next section.

The main types of functional tests for Kurento are the following:

- WebRTC. Real-time media in the web is one of the core Kurento capabilities, and therefore, a rich test suite to assess the use of WebRTC in Kurento has been implemented. Moreover, two special WebRTC features are also tested:

    - Datachannels. A WebRTC data channel allows to send custom data over an active connection to a peer. Tests using Chrome and Firefox has been implemented to check WebRTC datachannels.

    - ICE. In order to create media communication between peers avoiding *NAT Traversal* problems, ICE (Interactive Connectivity Establishment) negotiation is used in WebRTC. Kurento ICE tests check this connectivity using different network setups (NATs, reflexive, bridge).

- Recorder. Another important capability provided by Kurento is the media archiving. Recorder tests use *RecorderEndpoint* media element by ensuring that the recorded media is as expected.

- Player. KMS's *PlayerEndpoint* allows to inject media from seekable or non-seekable sources to a media pipeline. A suite of tests have been implemented to assess this feature.

- Composite/Dispatcher. KMS allows to mix media using different media elements (*Composite* and *Dispatcher*). These tests are aimed to asses the result of this media mixing.

33.1.2 Stability

Stability tests verifies Kurento capabilities in different scenarios:

- Running media pipelines in large amount of time.

- Using a lot of resources (CPU, memory) of a KMS instance.

Stability tests have been also created using Java, and they are contained in the project kurento-test. Again, we use Maven to execute stability tests against a local KMS and using also local browsers (Chrome, Firefox):

```
git clone https://github.com/Kurento/kurento-java.git
cd kurento-java
mvn \
    --projects kurento-integration-tests/kurento-test --also-make \
    -Pintegration \
    -Dgroups=org.kurento.commons.testing.SystemStabilityTests \
    clean verify
```

33.1.3 Tutorials

The documentation of Kurento includes a number of tutorials tutorials which allows to understand Kurento capabilities using ready to be used simple applications. Kurento tutorials have been developed for three technologies: Java, JavaScript, and Node.js. Moreover, for some of the Java tutorials, different E2E tests have been created. These tests are available in the project kurento-tutorial-test. In order to run these tests, Maven should be used:

```
git clone https://github.com/Kurento/kurento-tutorial-test
cd kurento-tutorial-test
mvn clean verify
```

33.1.4 API

The Kurento API is available in two languages: Java and JavaScript. For both of them, a test suite has been created to verify the correctness of the Kurento API against a running instance of KMS. In you want to run API tests for Java, as usual for Kurento tests, Maven is required, as follows:

```
git clone https://github.com/Kurento/kurento-java.git
cd kurento-java
mvn \
    --projects kurento-integration-tests/kurento-client-test --also-make \
    -Pintegration \
    -Dgroups=org.kurento.commons.testing.KurentoClientTests \
    clean verify
```

In order to run JavaScript API tests against a running instance of local KMS, the command to be used is the following:

```
git clone https://github.com/Kurento/kurento-client-js
cd kurento-client-js
npm install
rm -f node_modules/kurento-client && ln -s .. node_modules/kurento-client
npm test
```

33.2 Running Java tests

Functional, stability, and Java API tests for Kurento have been created using a custom Java library called **Kurento Testing Framework** (KTF). For more details about this framework, please take a look to the next section. If you are interested only in running a group of functional or stability E2E tests in order to assess Kurento, please keep reading this section.

Maven is the the way which E2E Kurento are executed. Therefore, in order to run E2E tests, first we need in have Java and Maven installed. The next step is cloning the GitHub repository which contains the test sources. Most of them are located in the kurento-test project, located inside of kurento-java. Inside this project, we need to invoke Maven to execute tests, for example as follows:

```
git clone https://github.com/Kurento/kurento-java.git
cd kurento-java
mvn \
    --projects kurento-integration-tests/kurento-test --also-make \
    -Pintegration \
    -Dgroups=org.kurento.commons.testing.IntegrationTests \
    -Dtest=WebRtcOneLoopbackTest \
    clean verify
```

Let's take a closer look to the Maven command:

- `mvn [...] clean verify`: Command to execute the *clean* and *verify* goals in Maven. *clean* will ensure that old build artifacts are deleted, and *verify* involves the execution of the unit and integration tests of a Maven project.

- `--projects kurento-integration-tests/kurento-test --also-make`: Maven options that select a single project for the goal, in this case *kurento-test*, and builds it together with any other dependency it might have.

- `-Pintegration`: Enables the "*integration*" profile ID, as defined in the file *kurento-integration-tests/pom.xml*.

- `-Dgroups=org.kurento.commons.testing.IntegrationTests`: The Kurento E2E test suite is divided into different JUnit 4's categories. This option allows to select different types of IntegrationTests. The most used values for this group are:

  - *IntegrationTests*: Parent category for all Kurento E2E tests.

  - *SystemFunctionalTests*: To run functional tests (as defined in section before).

  - *SystemStabilityTests*: To run stability tests (as defined in section before).

  - *KurentoClientTests*: To run Java API tests (as defined in section before). If this option is used, the project should be also changed using `--projects kurento-integration-tests/kurento-client-test`.

- `-Dtest=WebRtcOneLoopbackTest`: Although not mandatory, it is highly recommended, to select a test or group of test using Maven's *-Dtest* parameter. Using this command we can select a test using the Java class name.

  The wildcard `*` can be used, and Kurento tests follow a fixed notation for their naming, so this can be used to select a group of tests. Note that it's a good idea to quote the string, to prevent unexpected shell expansions. For example:

  - `-Dtest='WebRtc*'`: Used to execute all the functional Kurento tests for WebRTC.

  - `-Dtest='Player*'`: Used to execute all the functional Kurento tests for player.

  - `-Dtest='Recorder*'`: Used to execute all the functional Kurento tests for recorder.

  - `-Dtest='Composite*'`: Used to execute all the functional Kurento tests for composite.

  - `-Dtest='Dispatcher*'`: Used to execute all the functional Kurento tests for dispatcher.

  It's also possible to select multiple test classes with a comma (`,`), such as in `-Dtest=TestClass1,TestClass2`.

  Finally, it is possible to select individual methods *inside* the test classes, separating them with the # symbol:

  - `-Dtest='PlayerOnlyAudioTrackTest#testPlayerOnlyAudioTrackFileOgg*'`: Run the *PlayerOnlyAudioTrackTest.testPlayerOnlyAudioTrackFileOgg* in all its browser configurations (first Chrome, then Firefox).

  Note that the method name is given with a wildcard; this is because for most tests, the actual method name includes information about the browser which is used. Using a wildcard would run this test with both Chrome and Firefox; to choose specifically between those, specify it in the method name:

  - `-Dtest='PlayerOnlyAudioTrackTest#testPlayerOnlyAudioTrackFileOgg[0: chrome]'`: Run *PlayerOnlyAudioTrackTest.testPlayerOnlyAudioTrackFileOgg* exclusively with the Chrome browser. Normally, Chrome is "*[0: chrome]*" and Firefox is "*[1: firefox]*".

  Other combinations are possible:

  - `-Dtest='TestClass#testMethod1*+testMethod2*'`: Run *testMethod1* and *testMethod2* from the given test class.

An HTML report summarizing the results of a test suite executed with KTF is automatically created for Kurento tests. This report is called *report.html* and it is located by default on the *target* folder when tests are executed with Maven. The following picture shows an example of the content of this report.

Fig. 1: *Kurento Test Framework report sample*

Kurento tests are highly configurable. This configuration is done simply adding extra JVM parameters (i.e. `-Dparameter=value`) to the previous Maven command. The following sections summarizes the main test parameters and its default values organized in different categories.

33.2.1 Kurento Media Server

Kurento Media Server (KMS) is the heart of Kurento and therefore it must be properly configured in E2E tests. The following table summarizes the main options to setup KMS in these tests:

Parameter	Description	Default value
test.kms.autostart	Specifies if tests must start Kurento Media Server by themselves (with the method set by *test.kms.scope*), or if an external KMS service should be used instead: • *false*: Test must use an external KMS service, located at the URL provided by property *kms.ws.uri* • *test*: A KMS instance is automatically started before each test execution, and stopped afterwards. • *testsuite*: A KMS instance is started at the beginning of the test suite execution. A "test suite" is the whole group of tests to be executed (e.g. all functional tests). KMS service is stopped after test suite execution.	*test*
test.kms.scope	Specifies how to start KMS when it is internally managed by the test itself (`-Dtest.kms.autostart != false`): • *local*: Try to use local KMS installation. Test will fail is no local KMS is found. • *remote*: KMS is a remote host (use *kms.login* and *kms.passwd*, or *kms.pem*, to access using SSH to the remote machine). • *docker*: Request the docker daemon to start a KMS container based in the image specified by *test.kms.docker.image.name*. Test will fail if daemon is unable to start KMS container. In order to use this scope, a Docker server should be installed in the machine running tests. In addition, the Docker REST should be available for Docker client (used in test).	*local*
test.kms.docker.image.name	KMS docker image used to start a new docker container when KMS service is internally managed by test (`-Dtest.kms.autostart=test` or `testsuite`) with docker scope (`-Dtest.kms.scope=docker`). Ignored if *test.kms.autostart* is *false*. See	*kurento/kurento-media-server-dev:latest*

33.2. Running Java tests 481

For example, in order to run the complete WebRTC functional test suite against a local instance KMS, the Maven command would be as follows:

```
mvn \
    --projects kurento-integration-tests/kurento-test --also-make \
    -Pintegration \
    -Dgroups=org.kurento.commons.testing.SystemFunctionalTests \
    -Dtest=WebRtc* \
    -Dtest.kms.autostart=false \
    clean verify
```

In this case, an instance of KMS should be available in the machine running the tests, on the URL `ws://localhost:8888/kurento` (which is the default value for *kms.ws.uri*).

33.2.2 Browsers

In order to test automatically the web application under test using Kurento, web browsers (typically Chrome or Firefox, which allow to use WebRTC) are required. The options to configure these browsers are summarized in the following table:

Parameter	Description	Default value
test.selenium.scope	Specifies the scope used for browsers in Selenium test scenarios: • *local*: browser installed in the local machine. • *docker*: browser in Docker container (Chrome or Firefox). • *saucelabs*: browser in Sauce-Labs cloud.	*local*
docker.node.chrome.image	Docker image identifier for Chrome when browser scope is *docker*.	*elastestbrowsers/chrome:latest*
docker.node.firefox.image	Docker image identifier for Firefox when browser scope is *docker*.	*elastestbrowsers/firefox:latest*
test.selenium.record	Allow recording the browser while executing a test, and generate a video with the completely test. This feature can be activated (*true*) only if the scope for browsers is *docker*.	*false*
test.config.file	Path to a JSON-based file with configuration keys (test scenario, see "KTF explained" section for further details). Its content is transparently managed by test infrastructure and passed to tests for configuration purposes.	*test.conf.json*
test.timezone	Time zone for dates in browser log traces. This feature is interesting when using Saucelabs browsers, in order to match dates from browsers with KMS. Accepted values are *GMT*, *CET*, etc.	none
saucelab.user	User for SauceLabs	none
saucelab.key	Key path for SauceLabs	none
saucelab.idle.timeout	Idle time in seconds for SauceLabs requests	*120*
saucelab.command.timeout	Command timeout for SauceLabs requests	*300*
saucelab.max.duration	Maximum duration for a given SauceLabs session (in seconds)	1800

For example, in order to run the complete WebRTC functional test suite using *dockerized* browsers and recordings, the command would be as follows:

```
mvn \
    --projects kurento-integration-tests/kurento-test --also-make \
    -Pintegration \
    -Dgroups=org.kurento.commons.testing.SystemFunctionalTests \
    -Dtest=WebRtc* \
    -Dtest.selenium.scope=docker \
    -Dtest.selenium.record=true \
```

```
    clean verify
```

In order to avoid wasting too much disk space, recordings of successful tests are deleted at the end of the test. For failed tests, however, recordings will be available by default on the path `target/surefire-reports/` (which can be changed using the property `-Dtest.project.path`).

33.2.3 Web server

Kurento is typically consumed using a web application. E2E tests follow this architecture, and so, a web application up and running in a web server is required. Kurento-test provides a sample web application out-of-the-box aimed to assess main Kurento features. Also, a custom web application for tests can be specified using its URL. The following table summarizes the configuration options for the test web applications.

Parameter	Description	Default value
test.app.autostart	Specifies whether test application where Selenium browsers connect must be started by test or if it is externally managed: • *false* : Test application is externally managed and not started by test. This is required when the web under test is already online. In this case, the URL where Selenium browsers connects is specified by the properties: *test.host*, *test.port*, *test.path* and *test.protocol*. • *test* : test application is started before each test execution. • *testsuite*: Test application is started at the beginning of test execution.	*testsuite*
test.host	IP address or host name of the URL where Selenium browsers will connect when test application is externally managed (`-Dtest.app.autostart=false`). Notice this address must be reachable by Selenium browsers and hence network topology between browser and test application must be taken into account.	*127.0.0.1*
test.port	Specifies port number where test application must bind in order to listen for browser requests.	*7779*
test.path	Path of the URL where Selenium connects when test application is externally managed (`-Dtest.app.autostart=false`).	*/*
test.protocol	Protocol of the URL where Selenium browsers will connect when test application is externally managed (`-Dtest.app.autostart=false`).	*http*
test.url.timeout	Timeout (in seconds) to wait that web under test is available.	*500*

33.2.4 Fake clients

In some tests (typically in performance or stability tests), another instance of KMS is used to generate what we call *fake clients*, which are WebRTC peers which are connected in a WebRTC one to many communication. The KMS used for this features (referred as *fake KMS*) is controlled with the parameters summarized in the following table:

Parameter	Description	Default value
fake.kms.scope	This property is similar to *test.kms.scope*, except that it affects the KMS used by fake client sessions.	*local*
fake.kms.ws.uri	URL of a KMS service used by WebRTC clients. This property is used when service is externally managed (`-Dfake.kms.autostart=false`) and ignored otherwise. If not specified, *kms.ws.uri* is first looked at before using the default value.	`ws://localhost:8888/kurento`
fake.kms.autostart	Specifies if tests must start KMS or an external KMS service must be used for fake clients (sessions that use KMS media pipelines instead of the WebRTC stack provided by a web browser): *false*: Test must use an external KMS service whose URL is provided by the property *fake.kms.ws.uri* (with *kms.ws.uri* as fallback). Test will fail if neither properties are provided. *test*: KMS instance is started for before each test execution. KMS is destroyed after test execution. *testsuite*: KMS service is started at the beginning of test suite execution. KMS service is stopped after test suite execution. Following properties are honored when KMS is managed by test: *fake.kms.scope*, *test.kms.docker.image.name*, *test.kms.debug*	*false*

Although available in KTF, the fake clients feature is not very used in the current tests. You can see an example in the stability test LongStabilityCheckMemoryTest.

33.2.5 Other test features

Kurento tests can be configured in many different ways. The following table summarizes these miscellaneous features for tests.

Param-eter	Description	Default value
test.num.retries	Number of retries for failed tests	*1*
test.report	Path for HTML report	`target/ report. html`
test.project.path	Path for test file output (e.g. log files, screen captures, and video recordings).	`target/ surefire-reports/`
test.workspace	Absolute path of working directory used by tests as temporary storage. Make sure test user has full access to this folder.	`/tmp`
test.workspace.host	Absolute path, seen by docker agent, where directory *test.workspace* is mounted. Mandatory when scope is set to docker, as it is used by test infrastructure to share config files. This property is ignored when scope is different from docker.	*none*
test.docker.forcepulling	Force running *docker pull* to always get the latest Docker images.	*true*
test.files.disk	Absolute path where test files (videos) are located.	`/var/lib/ jenkins/ test-files`
test.files.http	Hostname (without "http://") of a web server where test files (videos) are located.	*files.openvidu.io*
test.player.url	URL used for playback tests. It can be anything supported by PlayerEndpoint: *file://...*, *http://...*, *rtsp://...*, etc.	*http://{test.files.http}*
project.path	In Maven reactor projects this is the absolute path of the module where tests are located. This parameter is used by test infrastructure to place test attachments. Notice this parameter must not include a trailing /.	*.*
kms.generate.rtp.pcap	Path where rtp/pst statistics will be stored	`file:// WORKSPACE/ testClassName`
bower.kurentoclient.tag	Tag used by Bower to download kurento-client	none
bower.kurentoutils.tag	Tag used by Bower to download kurento-utils.	none
bower.release.url	URL from where JavaScript binaries (kurento-client and kurento-utils) will be downloaded. Dependencies will be gathered from Bower if this parameter is not provided.	none
test.seek.repetitions	Number of times the tests with seek feature will be executed	*100*
test.num.sessions	Number of total sessions executed in stability tests	*50*
test.screenshare.title	Title of the window to be shared automatically from tests	*Screen 1*

33.3 Kurento Testing Framework explained

In order to assess properly Kurento from a final user perspective, a rich suite of E2E tests has been designed and implemented. To that aim, the **Kurento Testing Framework** (KTF) has been created. KTF is a part of the Kurento project aimed to carry out end-to-end (E2E) tests for Kurento. KTF has been implemented on the top of two well-known open-source testing frameworks: JUnit and Selenium.

KTF provides high level capabilities to perform advanced automated testing for Kurento-based applications. KTF has been implemented in Java, and as usual it is hosted on GitHub, in the project kurento-test. KTF has been designed on the top of **JUnit 4**, providing a rich hierarchy of classes which are going to act as parent for JUnit 4 tests cases. This hierarchy is the following:

The most important classes of this diagram are the following:

- KurentoTest: Top class of the KTF. It provides different features out-of-the-box for tests extending this class, namely:
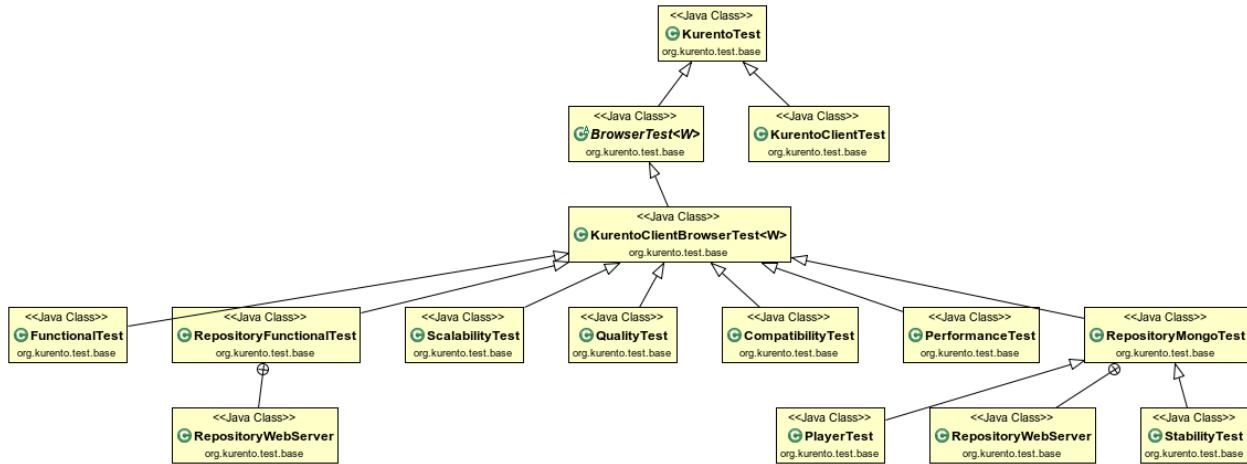
Fig. 2: *Kurento Testing Framework class hierarchy*

– Improved test lifecycle: KTF enhances the lyfecycle of JUnit 4 test cases, watching the result of tests (passed, failed). Moreover, KTF provides extra annotations to be used in different parts of the test lifecycle, such as FailedTest, FinishedTest, FinishedTestClass, StartedTest, StartedTestClass, or SucceededTest.

– Reporting: As introduced before, an HTML report summarizing the results of a test suite executed with KTF is automatically created for Kurento tests (*report.html*, located by default on the *target* folder when tests are executed with Maven).

– Retries mechanism: In order to detect flaky tests, a retries mechanism is present in KTF. This mechanism allows to repeat a failed test a configurable number of times.

• KurentoClientTest: It provides an instance of **Kurento Media Server** (KMS) together with a instance of a **Kurento Java Client** to control KMS. There are three options to run this KMS (see parameter *test.kms.scope*):

– Local KMS. To use this option, it is a pre-requisite to have KMS installed in the machine running this type of tests.

– Remote KMS. To use this option, it is a pre-requisite that KMS is installed in a remote host. If this KMS is going to be started by tests, then it is also required to have SSH access to the remote host in which KMS is installed (using parameters *kms.login* and *kms.passwd*, or providing a certificate with *kms.pem*).

– KMS in a **Docker** container. To use this option, it is a pre-requisite to have Docker installed in the machine running this type of tests.

• BrowserTest: This class provides wrappers of Selenium WebDriver instances aimed to control a group of web browsers for tests. By default, KTF allows to use **Chrome** or **Firefox** as browsers. The scope of these browsers can be configured to use:

– Local browser, i.e. installed in the local machine.

– Remote browser, i.e. installed in the remote machines (using Selenium Grid).

– Docker browsers, i.e. executed in Docker containers.

– Saucelabs browsers. Saucelabs is a cloud solution for web testing. It provides a big number of browsers to be used in Selenium tests. KTF provides seamless integration with Saucelabs.

Test scenario can be configured in *BrowserTest* tests in two different ways:

– Programmatically using Java. Test scenario uses JUnit 4's parameterized feature. The Java class TestScenario is used by KTF to configure the scenario, for example as follows:

```
@Parameters(name = "{index}: {0}")
public static Collection<Object[]> data() {
    TestScenario test = new TestScenario();
    test.addBrowser(BrowserConfig.BROWSER, new Browser.Builder().
→browserType(BrowserType.CHROME)
        .scope(BrowserScope.LOCAL).webPageType(webPageType).build());

    return Arrays.asList(new Object[][] { { test } });
}
```

– Using a JSON file. KTF allows to describe tests scenarios based on JSON notation. For each execution defined in these JSON files, the browser scope can be chosen. For example, the following example shows a test scenario in which two executions are defined. First execution defines two local browsers (identified as peer1 and peer2), Chrome and Firefox respectively. The second execution defines also two browsers, but this time browsers are located in the cloud infrastructure provided by Saucelabs.

```
{
    "executions":[
        {
            "peer1":{
                "scope":"local",
                "browser":"chrome"
            },
            "peer2":{
                "scope":"local",
                "browser":"firefox"
            }
        },
        {
            "peer1":{
                "scope":"saucelabs",
                "browser":"explorer",
                "version":"11"
            },
            "peer2":{
                "scope":"saucelabs",
                "browser":"safari",
                "version":"36"
            }
        }
    ]
}
```

- KurentoClientBrowserTest: This class can be seen as a mixed of the previous ones, since it provides the capability to use KMS (local or *dockerized*) together with a group of browser test using a *test scenario*. Moreover, it provides a web server started with each test for testing purposed, with a custom web page available to test **WebRTC** in Kurento in a easy manner. As can be seen in the diagram before, this class is the parent of a rich variety of different classes. In short, these classes are used to distinguish among different types of tests. See next section for more information.

33.3.1 Test scenario in JSON

Test scenario consist of a list of executions, where each execution describes how many browsers must be available and their characteristics. Each browser has an unique identifier (can be any string) meaningful for the test. The following keys can be specified in a JSON test scenario in order to customize individual instances:

- *scope*: Specifies what type of browser infrastructure has to be used by the test execution. This value can be overridden by command line property *test.selenium.scope*.

    - *local*: Start the browser as a local process in the same CPU where test is executed.

    - *docker*: Start browser as a docker container.

    - *saucelabs*: Start browser in SauceLabs.

- *host*: IP address or host name of URL used by the browser to execute tests. This value can be overridden by command line property *test.host*.

- *port*: Port number of the URL used by the browser to execute the test. This value can be overridden by command line property *test.port*.

- *path*: Path of the URL used by browser to execute the test. This value can be overridden by command line property *test.path*.

- *protocol*: Protocol of the URL used by browser to execute the test. This value can be overridden by command line property *test.protocol*.

- *browser*: Specifies the browser platform to be used by the test execution. Test will fail if required browser is not found.

- *saucelabsUser*: SauceLabs user. This property is mandatory for SauceLabs scope and ignored otherwise. Its value can be overridden by command line property *saucelab.user*.

- *saucelabsKey*: SauceLabs key. This property is mandatory for SauceLabs scope and ignored otherwise. Its value can be overridden by command line property *saucelab.key*.

- *version*: Version of browser to be used when test is executed in SauceLabs infrastructure. Test will fail if requested version is not found.

33.3.2 TO-DO

Rename:

- test.kms.docker.image.name -> test.docker.image.kms

- docker.node.chrome.image -> test.docker.image.chrome

- docker.node.firefox.image -> test.docker.image.firefox

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*.

34.1 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 actual 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). These messages are appended to normal RTCP packets.

34.2 Meaning of 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)
>   - ssrcs: 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.
```

H.264 video codec

This page is all about H.264, also known as **Advanced Video Coding (AVC)**, **MPEG-4 Part 10**, or **MPEG-4 AVC**.

Main resources:

- H.264 Specification

-

Table of Contents

35.1 Profiles and Levels

Profiles and Levels specify restrictions on bitstreams and hence limits on the capabilities needed to **decode** them. They may also be used to indicate interoperability points between individual decoder implementations. These parameters are defined in the **Annex A** of the H.264 Specification:

- Each **Profile** specifies a subset of features that shall be supported by all *decoders* conforming to that Profile.

- Each **Level** specifies a set of limits on the values that may be taken by the parameters of the bitstream.

The same set of Level definitions is used with all Profiles, but individual implementations may support a different Level for each supported Profile. For any given Profile, Levels generally correspond to decoder processing load and memory capability.

While common knowledge states that bandwidth usage decreases as H.264 moves from Baseline to Main to High Profile, unfortunately this is simply not true. Users seeking to reduce bandwidth usage should test their sources (such as cameras) in place with multiple Profiles to determine which is best for their application, as manufacturer implementation and data savings vary widely. For example, switching to High Profile in some cameras may increase bitrate, making network congestion issues worse.

The Profile and Level of an H.264 stream is usually given by a 3-byte hexadecimal value called **Sequence Parameter Set** (*SPS*):

1. **profile_idc**

2. **profile-iop**

    - *constraint_set0_flag*: Full compliance with Baseline Profile

    - *constraint_set1_flag*: Full compliance with Main Profile

    - *constraint_set2_flag*: Full compliance with Extended Profile

    - *constraint_set3_flag*

    - *constraint_set4_flag*

    - *constraint_set5_flag*

    - *reserved_zero_2bits* (2 bits, always 0)

3. **level_idc**

The *profile-iop* is a set of binary flags that change the meaning of the other two bytes. Their meaning are defined together with the different Profiles, and also in the H.264 Specification: *Section 7.4.2.1.1 Sequence parameter set data semantics*.

35.1.1 Profiles

These are the first five, most commonly used Profiles:

Profile name	Parameters
Baseline	*profile_idc* = 66 = 0x42, *constraint_set1_flag* = 0
Constrained Baseline	*profile_idc* = 66 = 0x42, *constraint_set1_flag* = 1
Main	*profile_idc* = 77 = 0x4D
Extended	*profile_idc* = 88 = 0x58
High	*profile_idc* = 100 = 0x64

There are more Profiles which specify higher capabilities, such as *High 10*, *High 4:2:2*, *High 4:4:4* or *CAVLC 4:4:4*. These are properly defined in the H.264 Specification.

35.1.2 Levels

The *Baseline*, *Constrained Baseline*, *Main*, and *Extended* Profiles share a set of Levels. These specify some numeric parameters related to the decoding bitrates, timings, motion vectors, and other algorithmic constrains that define the capabilities required by the decoder. The Specification contains a table where multiple Levels are defined: *1*, *1b*, *1.1*, *1.2*, *1.3*, *2*, *2.1*, . . . , up to *6.2*.

Profiles which are higher than the ones mentioned also have their own defined set of shared Levels.

35.2 NAL Units (NALU)

Sources:

- https://en.wikipedia.org/wiki/Network_Abstraction_Layer

- https://stackoverflow.com/questions/24884827/possible-locations-for-sequence-picture-parameter-sets-for-h-264-stream/24890903#24890903

An H.264 video is organized into Network Abstraction Layer Units ("NAL units" or "NALU") that help transporting it with optimal performance depending on whether the transport is stream-oriented or packet-oriented:

- For stream-oriented transports: the **Byte-Stream Format**. The NAL units are delivered as a continuous stream of bytes, which means that some boundary mechanism will be needed for the receiver to detect when one unit ends and the next one starts. This is done with the typical method of adding a unique byte pattern before each unit: The receiver will scan the incoming stream, searching for this **3-byte start code prefix**, and finding one will mark the beginning of a new unit.

- For packet-oriented transports: the **Packet-Transport Format**. This is the preferred method for the **Advanced Video Coding (AVC)** standard, and it builds upon the fact that all data is carried in packets that are already framed by the system transport protocol (such as RTP/UDP), so start code prefix patterns are not needed for identification of the boundaries of NAL units.

The NAL units can contain either actual data video (*VCL units*) or codec metadata (*non-VCL units*). Some of the most important types of NAL units are:

- **Sequence Parameter Set (SPS)**: This non-VCL NALU contains information required to configure the decoder such as profile, level, resolution, frame rate.

- **Picture Parameter Set (PPS)**: Similar to the SPS, this non-VCL NALU contains information on entropy coding mode, slice groups, motion prediction, quantization parameters (QP), and deblocking filters.

- **Instantaneous Decoder Refresh (IDR)**: This VCL NALU is a self contained image slice. That is, an IDR can be decoded and displayed without referencing any other NALU save SPS and PPS.

- **Access Unit Delimiter (AUD)**: An AUD is an optional NALU that can be use to delimit frames in an elementary stream. It is not required (unless otherwise stated by the container/protocol, like TS), and is often not included in order to save space, but it can be useful to finds the start of a frame without having to fully parse each NALU.

35.3 SPS, PPS

A large number of NAL units are combined to form a single video frame; the metadata of such frame would be transmitted in a **Picture Parameter Set (PPS)**. Likewise, a set of PPS would form an actual video sequence, and the metadata for it would be transmitted in a **Sequence Parameter Set (SPS)**. Both PPS and SPS can be sent well ahead of the actual units that will refer to them; then, each individual unit will just contain an index pointing to the corresponding parameter set, so the receiver is able to successfully decode the video.

Details about the exact contents of PPS and SPS packets can be found in the H.264 Specification sections "*Sequence parameter set data syntax*" and "*Picture parameter set RBSP syntax*".

Parameter sets can be sent in-band with the actual video, or sent out-of-band via some channel which might be more reliable than the transport of the video itself. This second option makes sense for transports where it is possible that some corruption or information loss might happen; losing a *PPS* could prevent decoding of a whole frame, and losing an *SPS* would be worse as it could render a whole chunk of the video impossible to decode, so it is important to transmit these parameter sets via the most reliable channel.

35.4 GStreamer caps

Whenever using H.264 as the video codec in Kurento, we'll see log messages such as this one:

```
caps: video/x-h264, stream-format=(string)avc, alignment=(string)au,
codec_data=(buffer)0142c01fffe1000e6742c01f8c8d40501e900f08846a01000468ce3c80,
level=(string)3.1, profile=(string)constrained-baseline, width=(int)640,
height=(int)480, framerate=(fraction)0/1, interlace-mode=(string)progressive,
chroma-format=(string)4:2:0, bit-depth-luma=(uint)8, bit-depth-chroma=(uint)8,
parsed=(boolean)true
```

This describes in detail all aspects of the encoded video; some of them are universal properties of any video (such as the width, height and framerate), while others are highly specific to the H.264 encoding.

- **stream-format**: Indicates if the H.264 video is stream-oriented (*stream-format = byte-stream*) or packet-oriented (*stream-format = avc*). For *byte-stream* videos the required parameter sets will be sent in-band with the video, but for *avc* the video metadata is conveyed via an additional *caps* field named *codec_data*.

- **codec_data**: Only present when the video is packet oriented (*stream-format = avc*), this value represents an **AVCDecoderConfigurationRecord** struct.

- Other information such as *level*, *profile*, *width*, *height*, *framerate*, *interlace-mode*, and the various *chroma* and *luma* settings, are just duplicated values that were extracted from the *codec_data* by an H.264 parser (namely the **h264parse** GStreamer element). This is also indicated by means of setting the field *parsed=true*.

35.5 GStreamer "codec_data"

GStreamer passes a *codec_data* field in its *caps* when the H.264 video is using the *avc* stream format. This field is printed in debug logs as a long hexadecimal sequence, but in reality it is an instance of an *AVCDecoderConfigurationRecord*, defined in the standard ISO/IEC 14496-15 (aka. *MPEG-4*) as follows:

```
aligned(8) class AVCDecoderConfigurationRecord {
    unsigned int(8) configurationVersion = 1;
    unsigned int(8) AVCProfileIndication;
    unsigned int(8) profile_compatibility;
    unsigned int(8) AVCLevelIndication;
    bit(6) reserved = '111111'b;
    unsigned int(2) lengthSizeMinusOne;
    bit(3) reserved = '111'b;
    unsigned int(5) numOfSequenceParameterSets;
    for (i=0; i< numOfSequenceParameterSets; i++) {
        unsigned int(16) sequenceParameterSetLength ;
        bit(8*sequenceParameterSetLength) sequenceParameterSetNALUnit;
    }
    unsigned int(8) numOfPictureParameterSets;
    for (i=0; i< numOfPictureParameterSets; i++) {
        unsigned int(16) pictureParameterSetLength;
        bit(8*pictureParameterSetLength) pictureParameterSetNALUnit;
    }
    if( profile_idc  ==  100  ||  profile_idc  ==  110  ||
        profile_idc  ==  122  ||  profile_idc  ==  144 )
    {
        bit(6) reserved = '111111'b;
        unsigned int(2) chroma_format;
        bit(5) reserved = '11111'b;
```

```
        unsigned int(3) bit_depth_luma_minus8;
        bit(5) reserved = '11111'b;
        unsigned int(3) bit_depth_chroma_minus8;
        unsigned int(8) numOfSequenceParameterSetExt;
        for (i=0; i< numOfSequenceParameterSetExt; i++) {
            unsigned int(16) sequenceParameterSetExtLength;
            bit(8*sequenceParameterSetExtLength) sequenceParameterSetExtNALUnit;
        }
    }
}
```

- **AVCProfileIndication**: profile code as defined in H.264 Specification (*profile_idc*).

- **profile_compatibility**: byte which occurs between the *profile_idc* and *level_idc* in a sequence parameter set (SPS), as defined in H.264 Specification. (*constraint_setx_flag*)

- **AVCLevelIndication**: level code as defined in H.264 Specification (*level_idc*).

- **lengthSizeMinusOne**: length in bytes of the *NALUnitLength* field in an AVC video sample or AVC parameter set sample of the associated stream minus one. For example, a size of one byte is indicated with a value of 0. The value of this field shall be one of 0, 1, or 3 corresponding to a length encoded with 1, 2, or 4 bytes, respectively.

- **numOfSequenceParameterSets**: number of SPSs that are used as the initial set of SPSs for decoding the AVC elementary stream.

- **sequenceParameterSetLength**: length in bytes of the SPS NAL unit as defined in H.264 Specification.

- **sequenceParameterSetNALUnit**: a SPS NAL unit, as specified in H.264 Specification. SPSs shall occur in order of ascending parameter set identifier with gaps being allowed.

- **numOfPictureParameterSets**: number of picture parameter sets (PPSs) that are used as the initial set of PPSs for decoding the AVC elementary stream.

- **pictureParameterSetLength**: length in bytes of the PPS NAL unit as defined in H.264 Specification.

- **pictureParameterSetNALUnit**: a PPS NAL unit, as specified in H.264 Specification. PPSs shall occur in order of ascending parameter set identifier with gaps being allowed.

- **chroma_format**: *chroma_format* indicator as defined by the *chroma_format_idc* parameter in H.264 Specification.

- **bit_depth_luma_minus8**: bit depth of the samples in the Luma arrays. For example, a bit depth of 8 is indicated with a value of zero (bit depth = 8 + *bit_depth_luma_minus8*). The value of this field shall be in the range of 0 to 4, inclusive.

- **bit_depth_chroma_minus8**: bit depth of the samples in the Chroma arrays. For example, a bit depth of 8 is indicated with a value of zero (bit depth = 8 + *bit_depth_chroma_minus8*). The value of this field shall be in the range of 0 to 4, inclusive.

- **numOfSequenceParameterSetExt**: number of Sequence Parameter Set Extensions that are used for decoding the AVC elementary stream.

- **sequenceParameterSetExtLength**: length in bytes of the SPS Extension NAL unit as defined in H.264 Specification.

- **sequenceParameterSetExtNALUnit**: a SPS Extension NAL unit, as specified in H.264 Specification.

35.5.1 Example mapping

Let's "translate" a sample *codec_data* into its components, to show the meaning of each field:

```
codec_data=(buffer)0142c01fffe1000e6742c01f8c8d40501e900f08846a01000468ce3c80
```

This would map to an *AVCDecoderConfigurationRecord* struct as follows:

```
0142c01fffe1000e6742c01f8c8d40501e900f08846a01000468ce3c80
01                                                    -> configurationVersion =␣
↪0x01 = 1
  42                                                  -> AVCProfileIndication =␣
↪0x42 = 66
    c0                                                -> profile_compatibility =␣
↪0xC0
      1f                                              -> AVCLevelIndication =␣
↪0x1F = 31
        ff                                            -> lengthSizeMinusOne =␣
↪0b11 = 3
          e1                                          ->␣
↪numOfSequenceParameterSets = 0b00001 = 1
            000e                                      ->␣
↪sequenceParameterSetLength = 0x000E = 14
                6742c01f8c8d40501e900f08846a          -> 1x14 bytes␣
↪sequenceParameterSetNALUnit
                                           01         ->␣
↪numOfPictureParameterSets = 0x01 = 1
                                             0004     ->␣
↪pictureParameterSetLength = 0x0004 = 4
                                                 68ce3c80 -> 1x4 bytes␣
↪pictureParameterSetNALUnit
```

This is the mapping for the first bytes of the Sequence Parameter Set NAL Unit:

```
6742c01f8c8d40501e900f08846a
67                            -> Header = 0x67 = 0b0110_0111
                                 forbidden_zero_bit = 0b0 = 0
                                 nal_ref_idc = 0b11 = 3
                                 nal_unit_type = 0b00111 = 7
  42                          -> profile_idc = 0x42 = 66
    c0                        -> constraint_setx_flag = 0xC0 = 0b1100_0000
                                 constraint_set0_flag = 1
                                 constraint_set1_flag = 1
      1f                      -> level_idc = 0x1F = 31
        ...                   -> Chroma, luma, scaling and more information
```

Note how the fields *profile_idc*, *constraint_setx_flag*, and *level_idc* get duplicated outside of this structure, in the *codec_data*'s *AVCProfileIndication*, *profile_compatibility*, and *AVCLevelIndication*, respectively.

In this example case, according to the definitions from H.264 Specification (Annex A.2.1 Baseline profile), a *profile_idc* of 66 with *constraint_set0_flag* and *constraint_set1_flag* = 1 corresponds to the H.264 **Constrained Baseline profile**; and *level_idc* = 31 which means **Level 3.1**.

MP4 recording format

While **AAC** and **H.264** are the most common audio and video encodings used in the industry, **MP4** is the *de-facto* standard file format to store them. MP4 acts as a "*container*", as its job is to contain one or more audio or video streams, together with any additional data such as subtitles or chapter markers.

When you configure a WebRTC stream over the network, the actual information being sent or received are the individual audio and video streams, possibly encoded as the mentioned AAC and H.264 codecs. However, if you use Kurento's **RecorderEndpoint**, then you need to choose a container format in order to store these streams into an actual file. Here is when the most important decision must be done, as different container formats have different advantages and drawbacks.

36.1 MP4 metadata issues

An MP4 file breaks down into several units of data, called **atoms**. Some of these atoms will contain actual audio samples and video frames; some contain accompanying data such as captions, chapter index, title, poster, etc; and lastly some contain so-called *metadata* which is information *about the video itself*, required by the player or receiving decoder to do its job adequately.

In an MP4 container, every movie contains a video metadata atom or "**moov atom**", describing the timescale, duration, and display characteristics of the movie. It acts as an index of the video data, and the file will not start to play until the player can access this index.

By default, MP4 stores the *moov* atom **at the end of the file**. This is fine for local playback, since the entire file is available for playback right away. However, writing the *moov* atom only at the end is a very bad idea for cases such as progressive download or streaming as it would force the download of the entire file first before it will start playback. Similarly, **live recording** in this format can be problematic if the recorder is abruptly stopped for whatever reason (think an unexpected crash), rendering an **unreadable file**.

The solution for these special cases is to move the *moov* atom to the beginning of the file. This ensures that the required movie information is downloaded (or recorded) first, enabling playback to start right away. The placement of the *moov* atom is specified in various software packages through settings such as "progressive download," "fast start," "streaming mode," or similar options.

36.1.1 MP4 Fast-Start in Kurento

An easy solution to the interrupted recording problem is to move the *moov* atom metadata to the beginning of the file. This means that if the recording crashes, all required metadata will have already been written out to the file, so the file will still be readable (at least to some point). Using this technique is called **MP4 Fast-Start**.

MP4 Fast-Start has been tested for recording in Kurento, however the results were not very satisfying. In case of a crash, the files were effectively playable, however this would come at a great cost in terms of resulting file size. MP4 videos with Fast-Start mode enabled would grow in size much faster than those recorded *without* Fast-Start enabled.

Due to this observation, MP4 container format is just considered not the best choice for live recording with Kurento, with WEBM being the better option. Also the popular **MKV** (Matroska container format) has been made available to use for recording since Kurento Media Server release **6.10**. This format is much more robust than MP4 and provides great results for live recording. However, note that most browsers don't have native support for directly opening local MKV files (e.g. with `<video src="myVideo.mkv">`), while they are able to open WEBM, so we still recommend WEBM as the most compatible container format for recordings.

NAT Types and NAT Traversal

Sources:

-
- Symmetric NAT and It's Problems (archive)
- Peer-to-Peer Communication Across Network Address Translators (archive)
- The hole trick - How Skype & Co. get round firewalls (archive)

37.1 Basic Concepts

37.1.1 Transport Address

A Transport Address is the combination of a host's IP address and a port. When talking about transmission of packets between hosts, we'll refer to them in terms of the transport addresses of both the source and the destination.

37.1.2 Packet transmission

In order to talk about traffic flows in this document, we'll refer to packet transmissions as the fact of sending data packets from a source to a destination host. The transmission is represented by the transport addresses of each host, plus a direction:

```
(SRC_ADDR, SRC_PORT) -> (DST_ADDR, DST_PORT)
```

Note:

- `->` denotes the direction of the transmission.

- `(ADDR, PORT)` denotes a transport address.

- `(SRC_ADDR, SRC_PORT)` is the transport address of the host that sends packets.

- `(DST_ADDR, DST_PORT)` is the transport address of the host that receives packets.

37.1.3 Inbound NAT transmission

A *NAT* maintains **inbound** rules which are used to translate transport addresses between the NAT's *external* and *internal* sides. Usually, the **external** side of the NAT is facing the public internet (**WAN**), while the **internal** side of the NAT is facing a private local network (**LAN**).

The inbound rules have the form of a *hash table* (or *map*), which stores a direct relationship between a pair of external transport addresses (a *quadruplet*) and a uniquely corresponding internal transport address. In other words:

- Given the quadruplet formed by the NAT external transport address, and a remote host's transport address ...

- ... there is an inbound rule for an internal transport address.

Typically, these NAT rules are created automatically during an outbound transmission that originated from within the LAN towards some remote host: it is at that moment when the NAT creates a new entry into its table (this is **step 1** in the following visualizations). Later, this entry in the NAT table is used to decide which local host needs to receive the response that the remote host might send (this is **step 2** in the visualizations). Rules created in this way are called "*dynamic rules*"; on the other hand, "*static rules*" can be explicitly created by an administrator, in order to set up a fixed NAT table.

Visualization:

```
   {NAT internal side}  |    {NAT external side}  |  {Remote host}
                        |                         |
1. (INT_ADDR, INT_PORT) => [ (EXT_ADDR, EXT_PORT) -> (REM_ADDR, REM_PORT) ]
2. (INT_ADDR, INT_PORT) <= [ (EXT_ADDR, EXT_PORT) <- (REM_ADDR, REM_PORT) ]
```

Meaning: Some host initiated an outbound packet transmission from the IP address INT_ADDR and port INT_PORT, towards the remote host at REM_ADDR and port REM_PORT. When the first packet crossed the NAT, it automatically created a dynamic rule, translating the internal transport address (INT_ADDR, INT_PORT) into the external transport address (EXT_ADDR, EXT_PORT). EXT_ADDR is the external IP address of the NAT, while EXT_PORT might be the same as INT_PORT, or it might be different (that is up to the NAT to decide).

Note:

- `->` and `<-` denote the direction of the transmission in each step.

- `=>` denotes the creation of a new rule in the NAT table.

- `[ (ADDR, PORT), (ADDR, PORT) ]`, with square brackets, denotes the transport address quadruplet used to access the NAT mapping table.

- `<=` denotes the resolution of the NAT mapping.

- `(INT_ADDR, INT_PORT)` is the **source** transport address on the *internal side* of the NAT for a local host making a transmission during step 1, and it is the **destination** transport address for the same host receiving the transmission during step 2.

- `(EXT_ADDR, EXT_PORT)` is the **source** transport address on the *external side* of the NAT from where a transmission is originated during step 1, and it is the **destination** transport address for the transmission being received during step 2.

- `(REM_ADDR, REM_PORT)` is the **destination** transport address of some remote host receiving a transmission during step 1, and it is the **source** transport address of a remote host that makes a transmission during step 2.

37.2 Types of NAT

There are two categories of NAT behavior, namely **Cone** and **Symmetric** NAT. The crucial difference between them is that the former will use the same port numbers for internal and external transport addresses, while the latter will always use different numbers for each side of the NAT. This will be explained later in more detail.

Besides, there are 3 types of Cone NATs, with varying degrees of restrictions regarding the allowed sources of inbound transmissions. To connect with a local host which is behind a Cone NAT, it's first required that the local host performs an outbound transmission to a remote one. This way, a dynamic rule will be created for the destination transport address, allowing the remote host to connect back. The only exception is the Full Cone NAT, where a static rule can be created beforehand by an administrator, thanks to the fact that this kind of NAT ignores what is the source transport address of the remote host that is connecting.

37.2.1 Full Cone NAT

This type of NAT allows inbound transmissions from *any source IP address* and *any source port*, as long as the destination tuple exists in a 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-grade routers. Of course, as it is the case for all NAT types, it is also possible to create dynamic rules by first performing an outbound transmission.

Visualization:

```
    {NAT internal side}  |    {NAT external side}  |  {Remote host}
                         |                         |
1. (INT_ADDR, INT_PORT) => [ (EXT_ADDR, INT_PORT) -> (REM_ADDR, REM_PORT) ]
2. (INT_ADDR, INT_PORT) <= [ (EXT_ADDR, INT_PORT) <- (   *    ,    *    ) ]
```
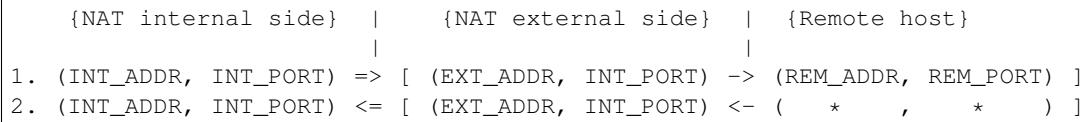
Note:

- ⋆ means that any value could be used: a remote host can connect from *any* IP address and port.

- The **source** IP address (*REM_ADDR*) in step 2 can be different from the **destination** IP address that was used in step 1.

- The **source** IP port (*REM_PORT*) in step 2 can be different from the **destination** IP port that was used in step 1.

- The *same* port (*INT_PORT*) is used in the internal and the external sides of the NAT. This is the most common case for all Cone NATs, only being different for Symmetric NATs.

37.2.2 (Address-)Restricted Cone NAT

This type of NAT allows inbound transmissions from a *specific source IP address* but allows for *any source port*. Typically, an inbound rule of this type was previously created dynamically, when the local host initiated an outbound transmission to a remote one.

Visualization:

```
    {NAT internal side}  |    {NAT external side}  |  {Remote host}
                         |                         |
1. (INT_ADDR, INT_PORT) => [ (EXT_ADDR, INT_PORT) -> (REM_ADDR, REM_PORT) ]
2. (INT_ADDR, INT_PORT) <= [ (EXT_ADDR, INT_PORT) <- (REM_ADDR,    *    ) ]
```
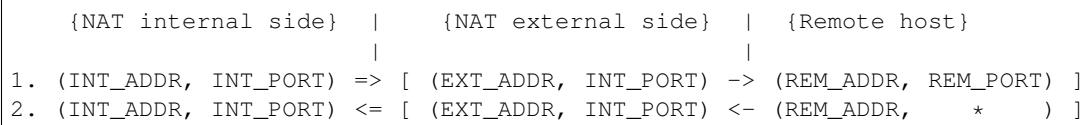
Note:

- The **source** IP address (*REM_ADDR*) in step 2 must be the same as the **destination** IP address that was used in step 1.

- The **source** IP port (*REM_PORT*) in step 2 can be different from the **destination** IP port that was used in step 1.

- The *same* port (*INT_PORT*) is used in the internal and the external sides of the NAT.

37.2.3 Port-Restricted Cone NAT

This is the most restrictive type of Cone NAT: it only allows inbound transmissions from a *specific source IP address* and a *specific source port*. Again, an inbound rule of this type was previously created dynamically, when the local host initiated an outbound transmission to a remote one.

Visualization:

```
    {NAT internal side}  |    {NAT external side}  |  {Remote host}
                         |                         |
1. (INT_ADDR, INT_PORT) => [ (EXT_ADDR, INT_PORT) -> (REM_ADDR, REM_PORT) ]
2. (INT_ADDR, INT_PORT) <= [ (EXT_ADDR, INT_PORT) <- (REM_ADDR, REM_PORT) ]
```
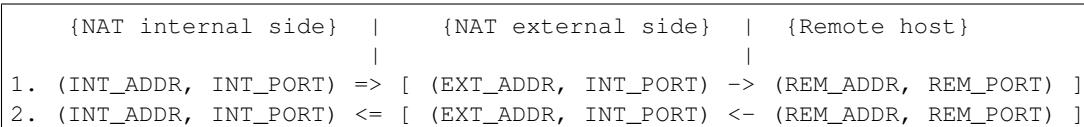
Note:

- The **source** IP address (*REM_ADDR*) in step 2 must be the same as the **destination** IP address that was used in step 1.

- The **source** IP port (*REM_PORT*) in step 2 must be the same as the **destination** IP port that was used in step 1.

- The *same* port (*INT_PORT*) is used in the internal and the external sides of the NAT.

37.2.4 Symmetric NAT

This type of NAT behaves in the same way of a Port-Restricted Cone NAT, with an important difference: for each outbound transmission to a different remote transport address (i.e. to a different remote host), the NAT assigns a **new random source port** on the external side. This means that two consecutive transmissions from the same local port to two different remote hosts will have two different external source ports, even if the internal source transport address 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 transmission to the *STUN* server and the direct transmission between peers.

Visualization:

```
    {NAT internal side}  |    {NAT external side}  |  {Remote host}
                         |                         |
1. (INT_ADDR, INT_PORT) => [ (EXT_ADDR, EXT_PORT1) -> (REM_ADDR, REM_PORT1) ]
2. (INT_ADDR, INT_PORT) <= [ (EXT_ADDR, EXT_PORT1) <- (REM_ADDR, REM_PORT1) ]
...
3. (INT_ADDR, INT_PORT) => [ (EXT_ADDR, EXT_PORT2) -> (REM_ADDR, REM_PORT2) ]
4. (INT_ADDR, INT_PORT) <= [ (EXT_ADDR, EXT_PORT2) <- (REM_ADDR, REM_PORT2) ]
```

Note:

- When the outbound transmission is done in step 1, *EXT_PORT1* gets defined as a new random port number, assigned for the new remote transport address *(REM_ADDR, REM_PORT1)*.

- Later, another outbound transmission is done in step 3, from the same local address and port to the same remote host but at a different port. *EXT_PORT2* is a new random port number, assigned for the new remote transport address *(REM_ADDR, REM_PORT2)*.

37.3 Types of NAT in the Real World

Quoting from :

> This terminology has been the source of much confusion, as it has proven inadequate at describing real-life NAT behavior. Many NAT implementations combine these types, and it is, therefore, better to refer to specific individual NAT behaviors instead of using the Cone/Symmetric terminology. **RFC 4787** attempts to alleviate this issue by introducing standardized terminology for observed behaviors. For the first bullet in each row of the above table, the RFC would characterize Full-Cone, Restricted-Cone, and Port-Restricted Cone NATs as having an *Endpoint-Independent Mapping*, whereas it would characterize a Symmetric NAT as having an *Address-* and *Port-Dependent Mapping*. For the second bullet in each row of the above table, **RFC 4787** would also label Full-Cone NAT as having an *Endpoint-Independent*

Filtering, Restricted-Cone NAT as having an *Address-Dependent Filtering*, Port-Restricted Cone NAT as having an *Address and Port-Dependent Filtering*, and Symmetric NAT as having either an *Address-Dependent Filtering* or *Address and Port-Dependent Filtering*. There are other classifications of NAT behavior mentioned, such as whether they preserve ports, when and how mappings are refreshed, whether external mappings can be used by internal hosts (i.e., its behavior), and the level of determinism NATs exhibit when applying all these rules.[2]

Especially, most NATs combine *symmetric NAT* for outbound transmissions with *static port mapping*, where inbound packets addressed to the external address and port are redirected to a specific internal address and port. Some products can redirect packets to several internal hosts, e.g., to divide the load between a few servers. However, this introduces problems with more sophisticated communications that have many interconnected packets, and thus is rarely used.

37.4 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 host that sits behind these kinds of devices. 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 host which is behind any type of NAT, it's first required that the local host performs an outbound transmission to the remote one. This way, a dynamic rule will be created for the destination transport address, allowing the remote host to connect back.

In order to tell one host when it has to perform an outbound transmission to another one, and the destination transport address it must use, the typical solution is to use a helper service such as *STUN*. This is usually managed by a third host, 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.

STUN / *TURN* requirements:

- 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*.

37.4.1 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 hosts:

A. One sitting behind a NAT. We'll call this the host **A**.

B. 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="198.51.100.1" # Public IP address of the NAT which hides the host A
A_PORT="1111"       # Listening port on the host A
B_IP="203.0.113.2"  # 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 at **A**.

37.4.2 PySTUN

PySTUN is a tool that uses STUN servers in order to try and detect what is the type of the NAT, when run from a host 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
```

RTP Streaming Commands

∗ *About the SRTP Master Key*

∗ *Using FFmpeg*

∗ *SDP Offer examples*

In this document you will find several examples of command-line programs that can be used to generate RTP and SRTP streams. These streams can then be used to feed any general (S)RTP receiver, although the intention here is to use them to connect an *RtpEndpoint* from a Kurento Media Server pipeline.

The tool used for all these programs is gst-launch, part of the GStreamer multimedia library.

These examples start from the simplest and then build on each other to end up with a full featured RTP generator. Of course, as more features are added, the command grows in complexity. A very good understanding of *gst-launch* and of GStreamer is recommended.

To run these examples, follow these initial steps:

1. Install required packages:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    gstreamer1.0-{tools,libav} \
    gstreamer1.0-plugins-{base,good,bad,ugly}
```

2. [Optional] Enable debug logging:

```
export GST_DEBUG=3
```

3. Copy & paste these commands into a console.

4. Write a Kurento application that does this:

```
sdpAnswer = rtpEndpoint.processOffer(sdpOffer);
log.info("SDP Answer: {}", sdpAnswer);
```

5. Start pipeline (e.g. in the *RTP Receiver tutorial*, push "Start")

6. From the logs: get the KMS port from the SDP Answer (in the RTP Receiver tutorial, this appears in the web page)

7. Set *PEER_V* in the *gst-launch* commands to the KMS port.

38.1 RTP sender examples

38.1.1 Simplest RTP sender

Features:

- Video RTP sender

```
PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
bash -c 'gst-launch-1.0 -e \
    uridecodebin uri="file://$SELF_PATH" \
        ! videoconvert ! x264enc tune=zerolatency \
        ! rtph264pay \
        ! "application/x-rtp,payload=(int)103,clock-rate=(int)90000" \
        ! udpsink host=$PEER_IP port=$PEER_V'
```

38.1.2 Example 2

Features:

- Video RTP sender

- Video RTCP receiver

```
PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
SELF_V=5004 SELF_VSSRC=112233 \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
↪"user\@example.com\"" \
    uridecodebin uri="file://$SELF_PATH" \
        ! videoconvert ! x264enc tune=zerolatency \
        ! rtph264pay \
        ! "application/x-rtp,payload=(int)103,clock-rate=(int)90000,ssrc=(uint)$SELF_
↪VSSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! udpsink host=$PEER_IP port=$PEER_V \
    udpsrc port=$((SELF_V+1)) \
        ! r.recv_rtcp_sink'
```

38.1.3 Example 3

Features:

- Video RTP sender

- Video RTCP receiver console dump

```
PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
SELF_V=5004 SELF_VSSRC=112233 \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
↪"user\@example.com\"" \
    uridecodebin uri="file://$SELF_PATH" \
        ! videoconvert ! x264enc tune=zerolatency \
        ! rtph264pay \
        ! "application/x-rtp,payload=(int)103,clock-rate=(int)90000,ssrc=(uint)$SELF_
↪VSSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! udpsink host=$PEER_IP port=$PEER_V \
    udpsrc port=$((SELF_V+1)) \
        ! tee name=t \
        t. ! queue ! r.recv_rtcp_sink \
        t. ! queue ! fakesink dump=true async=false'
```

38.1.4 Example 4

Features:

- Video RTP & RTCP sender

- Video RTCP receiver console dump

```
PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
SELF_V=5004 SELF_VSSRC=112233 \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
↪"user\@example.com\"" \
    uridecodebin uri="file://$SELF_PATH" \
        ! videoconvert ! x264enc tune=zerolatency \
        ! rtph264pay \
        ! "application/x-rtp,payload=(int)103,clock-rate=(int)90000,ssrc=(uint)$SELF_
↪VSSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! udpsink host=$PEER_IP port=$PEER_V \
    r.send_rtcp_src \
        ! udpsink host=$PEER_IP port=$((PEER_V+1)) sync=false async=false \
    udpsrc port=$((SELF_V+1)) \
        ! tee name=t \
        t. ! queue ! r.recv_rtcp_sink \
        t. ! queue ! fakesink dump=true async=false'
```

38.1.5 Example 5

Features:

- Video RTP & RTCP sender

- Video RTCP receiver console dump

- Symmetrical ports (for autodiscovery)

```
PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
SELF_V=5004 SELF_VSSRC=112233 \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
↪"user\@example.com\"" \
    uridecodebin uri="file://$SELF_PATH" \
        ! videoconvert ! x264enc tune=zerolatency \
        ! rtph264pay \
        ! "application/x-rtp,payload=(int)103,clock-rate=(int)90000,ssrc=(uint)$SELF_
↪VSSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! udpsink host=$PEER_IP port=$PEER_V bind-port=$SELF_V \
    r.send_rtcp_src \
        ! udpsink host=$PEER_IP port=$((PEER_V+1)) bind-port=$((SELF_V+1)) sync=false␣
↪async=false \
    udpsrc port=$((SELF_V+1)) \
        ! tee name=t \
        t. ! queue ! r.recv_rtcp_sink \
        t. ! queue ! fakesink dump=true async=false'
```

38.1.6 Example 6

Features:

- Audio RTP & RTCP sender

- Video RTCP receiver console dump

- Symmetrical ports (for autodiscovery)

```
PEER_A=9006 PEER_IP=127.0.0.1 \
SELF_A=5006 SELF_ASSRC=445566 \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
↪"user\@example.com\"" \
    audiotestsrc volume=0.5 \
        ! audioconvert ! audioresample ! opusenc \
        ! rtpopuspay \
        ! "application/x-rtp,payload=(int)96,clock-rate=(int)48000,ssrc=(uint)$SELF_
↪ASSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! udpsink host=$PEER_IP port=$PEER_A bind-port=$SELF_A \
    r.send_rtcp_src \
        ! udpsink host=$PEER_IP port=$((PEER_A+1)) bind-port=$((SELF_A+1)) sync=false␣
↪async=false \
    udpsrc port=$((SELF_A+1)) \
        ! tee name=t \
        t. ! queue ! r.recv_rtcp_sink \
        t. ! queue ! fakesink dump=true async=false'
```

38.1.7 Full-featured RTP sender

Features:

- Audio & Video RTP & RTCP sender

- Audio & Video RTCP receiver

- Video RTCP receiver console dump

- Symmetrical ports (for autodiscovery)

```
PEER_A=9006 PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
SELF_A=5006 SELF_ASSRC=445566 \
SELF_V=5004 SELF_VSSRC=112233 \
bash -c 'gst-launch-1.0 -e \
    rtpbin name=r sdes="application/x-rtp-source-sdes,cname=(string)\"user\@example.
↪com\"" \
    uridecodebin uri="file://$SELF_PATH" name=d \
    d. ! queue \
        ! audioconvert ! audioresample ! opusenc \
        ! rtpopuspay \
        ! "application/x-rtp,payload=(int)96,clock-rate=(int)48000,ssrc=(uint)$SELF_
↪ASSRC" \
        ! r.send_rtp_sink_0 \
    d. ! queue \
        ! videoconvert ! x264enc tune=zerolatency \
```

(continues on next page)

```
    ! rtph264pay \
    ! "application/x-rtp,payload=(int)103,clock-rate=(int)90000,ssrc=(uint)$SELF_
→VSSRC" \
    ! r.send_rtp_sink_1 \
r.send_rtp_src_0 \
    ! udpsink host=$PEER_IP port=$PEER_A bind-port=$SELF_A \
r.send_rtcp_src_0 \
    ! udpsink host=$PEER_IP port=$((PEER_A+1)) bind-port=$((SELF_A+1)) sync=false␣
→async=false \
udpsrc port=$((SELF_A+1)) \
    ! r.recv_rtcp_sink_0 \
r.send_rtp_src_1 \
    ! udpsink host=$PEER_IP port=$PEER_V bind-port=$SELF_V \
r.send_rtcp_src_1 \
    ! udpsink host=$PEER_IP port=$((PEER_V+1)) bind-port=$((SELF_V+1)) sync=false␣
→async=false \
udpsrc port=$((SELF_V+1)) \
    ! tee name=t \
    t. ! queue ! r.recv_rtcp_sink_1 \
    t. ! queue ! fakesink dump=true async=false'
```

38.2 RTP receiver examples

38.2.1 Example 1

Features:

- Video RTP & RTCP receiver

- RTCP sender

```
PEER_V=5004 PEER_IP=127.0.0.1 \
SELF_V=9004 \
CAPS_V="media=(string)video,clock-rate=(int)90000,encoding-name=(string)H264,
→payload=(int)103" \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
→"user\@example.com\"" \
udpsrc port=$SELF_V \
    ! "application/x-rtp,$CAPS_V" \
    ! r.recv_rtp_sink \
r.recv_rtp_src \
    ! rtph264depay \
    ! decodebin \
    ! autovideosink \
udpsrc port=$((SELF_V+1)) \
    ! r.recv_rtcp_sink \
r.send_rtcp_src \
    ! udpsink host=$PEER_IP port=$((PEER_V+1)) sync=false async=false'
```

Note: RtpSession is used to handle RTCP, and it needs explicit video caps.

38.2.2 Example 2

Features:

- Audio & Video RTP & RTCP receiver

- Video RTCP receiver console dump

- Audio & Video RTCP sender

- Symmetrical ports (for autodiscovery)

```
PEER_A=5006 PEER_ASSRC=445566 PEER_V=5004 PEER_VSSRC=112233 PEER_IP=127.0.0.1 \
SELF_A=9006 SELF_V=9004 \
CAPS_A="media=(string)audio,clock-rate=(int)48000,encoding-name=(string)OPUS,
↪payload=(int)96" \
CAPS_V="media=(string)video,clock-rate=(int)90000,encoding-name=(string)H264,
↪payload=(int)103" \
bash -c 'gst-launch-1.0 -e \
    rtpbin name=r sdes="application/x-rtp-source-sdes,cname=(string)\"user\@example.
↪com\"" \
    udpsrc port=$SELF_A \
        ! "application/x-rtp,$CAPS_A" \
        ! r.recv_rtp_sink_0 \
    r.recv_rtp_src_0_${PEER_ASSRC}_96 \
        ! rtpopusdepay \
        ! decodebin \
        ! autoaudiosink \
    udpsrc port=$((SELF_A+1)) \
        ! r.recv_rtcp_sink_0 \
    r.send_rtcp_src_0 \
        ! udpsink host=$PEER_IP port=$((PEER_A+1)) bind-port=$((SELF_A+1)) sync=false␣
↪async=false \
    udpsrc port=$SELF_V \
        ! "application/x-rtp,$CAPS_V" \
        ! r.recv_rtp_sink_1 \
    r.recv_rtp_src_1_${PEER_VSSRC}_103 \
        ! rtph264depay \
        ! decodebin \
        ! autovideosink \
    udpsrc port=$((SELF_V+1)) \
        ! tee name=t \
        t. ! queue ! r.recv_rtcp_sink_1 \
        t. ! queue ! fakesink dump=true async=false \
    r.send_rtcp_src_1 \
        ! udpsink host=$PEER_IP port=$((PEER_V+1)) bind-port=$((SELF_V+1)) sync=false␣
↪async=false'
```

38.3 SRTP examples

For the SRTP examples, you need to install the Kurento's fork of GStreamer:

```
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
    gstreamer1.5-{tools,libav} \
    gstreamer1.5-plugins-{base,good,bad,ugly}
```

38.3.1 SRTP simple sender

Features:

- Video SRTP sender

```
PEER_V=9004 PEER_IP=127.0.0.1 \
SELF_PATH="$PWD/video.mp4" \
SELF_VSSRC=112233 \
SELF_KEY="4142434445464748494A4B4C4D4E4F50515253545556575859 5A31323334" \
bash -c 'gst-launch-1.5 -e \
    uridecodebin uri="file://$SELF_PATH" \
    ! videoconvert \
    ! x264enc tune=zerolatency \
    ! rtph264pay \
    ! "application/x-rtp,payload=(int)103,ssrc=(uint)$SELF_VSSRC" \
    ! srtpenc key="$SELF_KEY" \
        rtp-cipher="aes-128-icm" rtp-auth="hmac-sha1-80" \
        rtcp-cipher="aes-128-icm" rtcp-auth="hmac-sha1-80" \
    ! udpsink host=$PEER_IP port=$PEER_V'
```

38.3.2 SRTP simple receiver

Features:

- Video SRTP receiver

```
PEER_VSSRC=112233 \
PEER_KEY="4142434445464748494A4B4C4D4E4F50515253545556575859 5A31323334" \
SELF_V=9004 \
SRTP_CAPS="payload=(int)103,ssrc=(uint)$PEER_VSSRC,roc=(uint)0, \
    srtp-key=(buffer)$PEER_KEY, \
    srtp-cipher=(string)aes-128-icm,srtp-auth=(string)hmac-sha1-80, \
    srtcp-cipher=(string)aes-128-icm,srtcp-auth=(string)hmac-sha1-80" \
bash -c 'gst-launch-1.5 -e \
    udpsrc port=$SELF_V \
    ! "application/x-srtp,$SRTP_CAPS" \
    ! srtpdec \
    ! rtph264depay \
    ! decodebin \
    ! autovideosink'
```

Note: No RtpSession is used to handle RTCP, so no need for explicit video caps.

38.3.3 SRTP complete sender

Features:

- Video SRTP & SRTCP sender
- SRTCP receiver console dump

```
PEER_V=9004 PEER_VSSRC=332211 PEER_IP=127.0.0.1 \
PEER_KEY="343332315A595857565554535251504F4E4D4C4B4A494847464544434241" \
SELF_PATH="$PWD/video.mp4" \
SELF_V=5004 SELF_VSSRC=112233 \
SELF_KEY="4142434445464748494A4B4C4D4E4F50515253545556575859545A31323334" \
SRTP_CAPS="payload=(int)103,ssrc=(uint)$PEER_VSSRC,roc=(uint)0, \
    srtp-key=(buffer)$PEER_KEY, \
    srtp-cipher=(string)aes-128-icm,srtp-auth=(string)hmac-sha1-80, \
    srtcp-cipher=(string)aes-128-icm,srtcp-auth=(string)hmac-sha1-80" \
bash -c 'gst-launch-1.5 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
→"user\@example.com\"" \
    srtpenc name=e key="$SELF_KEY" \
        rtp-cipher="aes-128-icm" rtp-auth="hmac-sha1-80" \
        rtcp-cipher="aes-128-icm" rtcp-auth="hmac-sha1-80" \
    srtpdec name=d \
    uridecodebin uri="file://$SELF_PATH" \
        ! videoconvert ! x264enc tune=zerolatency \
        ! rtph264pay \
        ! "application/x-rtp,payload=(int)103,ssrc=(uint)$SELF_VSSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! e.rtp_sink_0 \
    e.rtp_src_0 \
        ! udpsink host=$PEER_IP port=$PEER_V \
    r.send_rtcp_src \
        ! e.rtcp_sink_0 \
    e.rtcp_src_0 \
        ! udpsink host=$PEER_IP port=$((PEER_V+1)) sync=false async=false \
    udpsrc port=$((SELF_V+1)) \
        ! "application/x-srtcp,$SRTP_CAPS" \
        ! d.rtcp_sink \
    d.rtcp_src \
        ! tee name=t \
        t. ! queue ! r.recv_rtcp_sink \
        t. ! queue ! fakesink dump=true async=false'
```

38.3.4 SRTP complete receiver

Features:

- Video SRTP & SRTCP receiver

- SRTCP sender

```
PEER_V=5004 PEER_VSSRC=112233 PEER_IP=127.0.0.1 \
PEER_KEY="4142434445464748494A4B4C4D4E4F50515253545556575859545A31323334" \
SELF_V=9004 SELF_VSSRC=332211 \
SELF_KEY="343332315A595857565554535251504F4E4D4C4B4A494847464544434241" \
SRTP_CAPS="payload=(int)103,ssrc=(uint)$PEER_VSSRC,roc=(uint)0, \
    srtp-key=(buffer)$PEER_KEY, \
    srtp-cipher=(string)aes-128-icm,srtp-auth=(string)hmac-sha1-80, \
    srtcp-cipher=(string)aes-128-icm,srtcp-auth=(string)hmac-sha1-80" \
CAPS_V="media=(string)video,clock-rate=(int)90000,encoding-name=(string)H264,
→payload=(int)103" \
bash -c 'gst-launch-1.5 -e \
```

(continues on next page)

```
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\
→"recv\@example.com\"" \
    srtpenc name=e key="$SELF_KEY" \
        rtp-cipher="aes-128-icm" rtp-auth="hmac-sha1-80" \
        rtcp-cipher="aes-128-icm" rtcp-auth="hmac-sha1-80" \
    srtpdec name=d \
    udpsrc port=$SELF_V \
        ! "application/x-srtp,$SRTP_CAPS" \
        ! d.rtp_sink \
    d.rtp_src \
        ! "application/x-rtp,$CAPS_V" \
        ! r.recv_rtp_sink \
    r.recv_rtp_src \
        ! rtph264depay \
        ! decodebin \
        ! autovideosink \
    udpsrc port=$((SELF_V+1)) \
        ! "application/x-srtcp,$SRTP_CAPS" \
        ! d.rtcp_sink \
    d.rtcp_src \
        ! r.recv_rtcp_sink \
    fakesrc num-buffers=-1 sizetype=2 \
        ! "application/x-rtp,payload=(int)103,ssrc=(uint)$SELF_VSSRC" \
        ! r.send_rtp_sink \
    r.send_rtp_src \
        ! fakesink async=false \
    r.send_rtcp_src \
        ! e.rtcp_sink_0 \
    e.rtcp_src_0 \
        ! udpsink host=$PEER_IP port=$((PEER_V+1)) sync=false async=false'
```

Note: *fakesrc* is used to force *rtpsession* to use the desired SSRC.

38.4 Additional Notes

These are some random and unstructured notes that don't have the same level of detail as the previous section. They are here just as a way of taking note of alternative methods or useful bits of information, but don't expect that any command from this section works at all.

38.4.1 About 'sync=false'

Pipeline initialization is done with 3 state changes:

1. NULL -> READY: Underlying devices are probed to ensure they can be accessed.

2. READY -> PAUSED: Preroll is done, which means that an initial frame is brought from the sources and set into the sinks of the pipeline.

3. PAUSED -> PLAYING: Sources start generating frames, and sinks start receiving and processing them.

The **sync** property indicates whether the element is Live (`sync=true`) or Non-Live (`sync=false`):

- Live elements are synchronized against the clock, and only process data according to the established rate. The timestamps of the incoming buffers will be used to schedule the exact render time of its contents.

- Non-Live elements do not synchronize with any clock, and process data as fast as possible. The pipeline will ignore the timestamps of the video frames and it will play them as fast as they arrive, ignoring all timing information. Note that setting "sync=false" is almost never a solution when timing-related problems occur.

For example, a video camera or an output window/screen would be Live elements; a local file would be a Non-Live element.

The **async** property enables (`async=true`) or disables (`async=false`) the Preroll feature:

- Live sources cannot produce an initial frame until they are set to PLAYING state, so Preroll cannot be done with them on PAUSE state. If Prerolling is enabled in a Live sink, it will be set on hold waiting for that initial frame to arrive, and only then they will be able to complete the Preroll and start playing.

- Non-Live sources should be able to produce an initial frame before reaching the PLAYING state, allowing their downstream sinks to Preroll as soon as the PAUSED state is set.

Since RTCP packets from the sender should be sent as soon as possible and do not participate in preroll, `sync=false` and `async=false` are configured on *udpsink*.

See:

- https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-rtpbin.html

- https://gstreamer.freedesktop.org/documentation/design/latency.html

38.4.2 About the SRTP Master Key

The SRTP Master Key is the concatenation of (key, salt). With *AES_CM_128* + *HMAC_SHA1_80*, Master Key is 30 bytes: 16 bytes key + 14 bytes salt.

Key formats:

- GStreamer (*gst-launch*): Hexadecimal.

- Kurento (*RtpEndpoint*): ASCII.

- SDP Offer/Answer: Base64.

Use this website to convert between formats: http://tomeko.net/online_tools/hex_to_base64.php

Encryption key used by the **sender** examples:

- ASCII: `ABCDEFGHIJKLMNOPQRSTUVWXYZ1234`.

- In Hex: `4142434445464748494A4B4C4D4E4F505152535455565758595A31323334`.

- In Base64: `QUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVoxMjM0`.

Encryption key used by the **receiver** examples:

- ASCII: `4321ZYXWVUTSRQPONMLKJIHGFEDCBA`.

- In Hex: `343332315A595857565554535251504F4E4D4C4B4A494847464544434241`.

- In Base64: `NDMyMVpZWFdWVVRTUlFQT05NTEtKSUhHRkVEQ0JB`.

38.4.3 Using FFmpeg

It should be possible to use FFmpeg to send or receive RTP streams; just make sure that all stream details match between the SDP negotiation and the actual encoded stream. For example: reception ports, Payload Type, encoding settings, etc.

This command is a good starting point to send RTP:

```
ffmpeg -re -i "video.mp4" -c:v libx264 -tune zerolatency -payload_type 103 \
    -an -f rtp rtp://IP:PORT
```

Note that Payload Type is **103** in these and all other examples, because that's the number used in the SDP Offer sent to the *RtpEndpoint* in Kurento. You could use any other number, just make sure that it gets used consistently in both SDP Offer and RTP sender program.

38.4.4 SDP Offer examples

Some examples of the SDP Offer that should be sent to Kurento's *RtpEndpoint* to configure it with needed parameters for the RTP sender examples shown in this page:

Audio & Video RTP & RTCP sender

A basic SDP message that describes a simple Audio + Video RTP stream.

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
m=audio 5006 RTP/AVP 96
a=rtpmap:96 opus/48000/2
a=sendonly
a=ssrc:445566 cname:user@example.com
m=video 5004 RTP/AVP 103
a=rtpmap:103 H264/90000
a=sendonly
a=ssrc:112233 cname:user@example.com
```

Some modifications that would be done for KMS:

- Add support for *REMB Congestion Control*.

- Add symmetrical ports (for *Port Autodiscovery*).

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
m=audio 5006 RTP/AVP 96
a=rtpmap:96 opus/48000/2
a=sendonly
a=direction:active
a=ssrc:445566 cname:user@example.com
m=video 5004 RTP/AVPF 103
a=rtpmap:103 H264/90000
a=rtcp-fb:103 goog-remb
a=sendonly
```

(continues on next page)

```
a=direction:active
a=ssrc:112233 cname:user@example.com
```

Apple Safari

There are two main implementations of the Safari browser: the Desktop edition which can be found in Mac OS workstations and laptops, and the iOS edition which comes installed as part of the iOS Operating System in mobile devices such as iPhone or iPad.

39.1 Codec issues

Safari (both Desktop and iOS editions) included a half-baked implementation of the WebRTC standard, at the least with regards to the codecs compatibility. The WebRTC specs state that both VP8 and H.264 video codecs MUST be implemented in all WebRTC endpoints*[0], but Apple only added VP8 support starting from Safari Release 68. Older versions of the browser won't be able to decode VP8 video, so if the source video isn't already in H.264 format, Kurento Media Server will need to transcode the input video so they can be received by Safari.

In order to ensure compatibility with Safari browsers, also caring to not trigger on-the-fly transcoding between video codecs, it is important to make sure that Kurento has been configured with support for H.264, and it is also important to check that the SDP negotiations are actually choosing this as the preferred codec.

If you are targeting Safari version 68+, then this won't pose any problem, as now both H.264 and VP8 can be used for WebRTC.

39.2 HTML policies for video playback

Until recently, this has been the recommended way of inserting a video element in any HTML document:

```
<video id="myVideo" autoplay></video>
```

---

[0] RFC 7742, Section 5. Mandatory-to-Implement Video Codec: | WebRTC Browsers MUST implement the VP8 video codec as described in **RFC 6386** | and H.264 Constrained Baseline as described in H264. | | WebRTC Non-Browsers that support transmitting and/or receiving video | MUST implement the VP8 video codec as described in **RFC 6386** and | H.264 Constrained Baseline as described in H264.

All Kurento tutorials are written to follow this example. As a general rule, most browsers honor the *autoplay* attribute, *Desktop Safari* included; however, *iOS Safari* is an exception to this rule, because it implements a more restrictive set of rules that must be followed in order to allow playing video from a `<video>` HTML tag.

You should make a couple changes in order to follow with all the latest changes in browser's policies for automatically playing videos:

1. Start automatic video playback without audio, using the *muted* attribute together with *autoplay*.

2. Add the *playsinline* attribute if you want to avoid fullscreen videos in *iOS Safari*.

A video tag that includes all these suggestions would be like this:

```
<video id="myVideo" playsinline autoplay muted></video>
```

Sources for this section:

- https://webkit.org/blog/6784/new-video-policies-for-ios/

- https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video#Browser_compatibility

- https://developer.apple.com/library/content/releasenotes/General/WhatsNewInSafari/Articles/Safari_10_0.html

39.2.1 autoplay muted

The *autoplay* attribute is honored by all browsers, and it makes the `<video>` tag to automatically start playing as soon as the source stream is available. In other words: the method `video.play()` gets implicitly called as soon as a source video stream becomes available and is set with `video.srcObject = stream`.

However, in *iOS Safari* (version >= 10), the *autoplay* attribute is only available for videos that **have no sound**, are **muted**, or have a **disabled audio track**. In any other case, the *autoplay* attribute will be ignored, and the video won't start playing automatically when a new stream is set.

The solution that is most intuitive for the user is that a muted video is presented, and then the user is asked to click somewhere in order to enable the audio:

```
<video id="myVideo" autoplay muted></video>
```

This will allow the user interface to at least automatically start playing a video, so the user will see some movement and acknowledge that the media playback has started. Then, an optional label might ask the user to press to unmute, an action that would comply with the browser's *autoplay* policies.

Another alternative is to avoid using the *autoplay* attribute altogether. Instead, manually call the `play()` method as a result of some user interaction. The safest way is to call the `myVideo.play()` method from inside a button's *onclick* event handler.

39.2.2 playsinline

Most browsers assume that a video should be played from inside the specific area that the `<video>` element occupies. So, for example, a tag such as this one:

```
<video id="myVideo" width="480px" height="360px"></video>
```

will play the video in an area that is 480x360 pixels.

That is not the case for *iOS Safari*, where all videos play full screen by default: whenever a video starts playing, the browser will maximize its area to fill all the available space in the screen. This can be avoided by adding the *playsinline* attribute to the `<video>` tag:

```
<video id="myVideo" width="480px" height="360px" playsinline></video>
```

With this, videos will play in *iOS Safari* as they do in any other browser, as inline videos inside their corresponding area.

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 protocol used for *NAT Traversal*. It defines a technique that allows communication between two endpoints when one is inside a NAT and the other is outside of it. The net effect of the ICE process is that the NAT will be left with all needed ports open for communication, and both endpoints will have complete information about the IP address and ports where the other endpoint can be contacted.

ICE doesn't work standalone: it uses a couple of helper protocols called *STUN* and *TURN*.

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 and 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 a mechanism that hides from the public access the private IP addresses of machines inside a network. The NAT mechanism is typically found in all types of network devices, ranging from home routers to full-fledged corporate firewalls. In all cases the effect is the same: machines inside the NAT cannot be freely accessed from outside.

NAT introduces a lot of problems for WebRTC communications: machines inside the network will be able to send data to the outside, but they won't be able to receive data from remote participants that are outside the network. In order to allow for this, NAT devices typically allow to configure **NAT bindings** to let data come in from the outside part of the network; creating these NAT bindings is what is called *NAT Traversal*, also commonly referred as "opening ports".

See also:

NAT Types and NAT Traversal Entry in our Knowledge Base.

How Network Address Translation Works (archive) A comprehensive description of NAT and its mechanics.

NAT Traversal NAT Traversal 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.

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*.

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 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.

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? (archive)

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 Media Plane. Functions such as media negotiation, QoS parametrization, call establishment, user registration, user presence, etc. as managed in this plane.

See also:

Media Plane

WebRTC in the real world: STUN, TURN and signaling (archive)

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 (archive)

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 *Session Traversal Utilities for NAT* (STUN) is a protocol that complements *ICE* in the task of solving the *NAT Traversal* issue. It can be used by any endpoints to determine the IP address and port allocated to it by a *NAT*. It can also be used to check connectivity between two endpoints, and as a keep-alive protocol to maintain NAT bindings. STUN works with many existing types of NAT, and does not require any special behavior from them.

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 *Traversal Using Relays around NAT* (TURN) is an extension of *STUN*, used where the *NAT* security policies are too strict and the needed NAT bindings cannot be successfully created to achieve *NAT Traversal*. In these situations, it is necessary for the host to use the services of a TURN server that acts as a communication relay.

Note: You don't need to configure both STUN and TURN, because TURN already includes STUN functionality.

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 41

Indices and tables

- genindex

- search

Index