
Kurento Documentation

Release 7.0.0

Kurento

May 04, 2023

USER DOCUMENTATION

1	Introduction to Kurento	3
1.1	What is Kurento?	3
1.2	Why a WebRTC media server?	5
1.3	Why Kurento Media Server?	6
1.4	Kurento Design Principles	6
2	About OpenVidu	9
3	Getting Started	11
4	Installation Guide	13
4.1	Amazon Web Services	14
4.2	Docker image	15
4.3	Local Installation	16
4.4	STUN/TURN server install	18
4.5	Check your installation	18
5	Installing Nightly Builds	21
5.1	Kurento Media Server	22
5.2	Kurento Java Client	23
5.3	Kurento JavaScript Client	25
6	Configuration	27
6.1	Debug Logging	28
6.2	STUN/TURN Server	28
6.3	Network Interface	29
6.4	WebRTC Bitrate	29
6.5	RTP Ports	29
6.6	Advanced Settings	30
7	Tutorials	33
7.1	Hello World	34
7.2	WebRTC Magic Mirror	61
7.3	RTP Receiver	86
7.4	WebRTC One-To-Many broadcast	89
7.5	WebRTC One-To-One video call	114
7.6	WebRTC One-To-One video call with recording and filtering	141
7.7	WebRTC Many-To-Many video call (Group Call)	161
7.8	Media Elements metadata	169
7.9	WebRTC Media Player	178
7.10	WebRTC outgoing Data Channels	190

7.11	WebRTC incoming Data Channel	199
7.12	WebRTC recording	212
7.13	WebRTC statistics	228
7.14	Chroma Filter	232
7.15	Crowd Detector Filter	250
7.16	Plate Detector Filter	269
7.17	Pointer Detector Filter	283
8	Writing Kurento Applications	303
8.1	Global Architecture	303
8.2	Application Architecture	304
8.3	Media Plane	307
9	Writing Kurento Modules	309
9.1	Scaffolding and development	309
9.2	Installation and usage	312
9.3	Examples	317
10	Frequently Asked Questions	319
10.1	NAT, ICE, STUN, TURN	320
10.2	Kurento in Docker	325
10.3	Media Pipeline	327
11	Troubleshooting Issues	329
11.1	Media Server Crashes	330
11.2	Corrupted Video	331
11.3	Other Media Server issues	334
11.4	Application Server	338
11.5	WebRTC failures	342
11.6	Docker issues	344
11.7	Element-specific info	345
11.8	Browser	348
12	Support	349
12.1	Community Support	349
12.2	Commercial Support	350
13	Client API Reference	351
13.1	Java Client	351
13.2	JavaScript Client	351
13.3	Kurento Js Utils	351
14	Kurento Modules	353
14.1	Media Elements and Media Pipelines	353
14.2	Endpoints	355
14.3	Filters	357
14.4	Hubs	357
14.5	Example Modules	358
15	Kurento Protocol	361
15.1	JSON-RPC message format	362
15.2	Kurento API over JSON-RPC	363
15.3	Network issues	371
15.4	Example: WebRTC in loopback	372
15.5	Creating a custom Kurento Client	374

16 Kurento Utils JS	377
16.1 Overview	378
16.2 How to use it	378
16.3 Examples	378
16.4 Using data channels	380
16.5 Reference documentation	381
16.6 Souce code	384
16.7 Build for browser	385
17 Securing Kurento Applications	387
17.1 Securing Application Servers	387
17.2 Securing Kurento Media Server	389
18 Endpoint Events	393
18.1 MediaObject events	394
18.2 MediaElement events	395
18.3 BaseRtpEndpoint events	396
18.4 WebRtcEndpoint events	397
18.5 Sample sequence of events: WebRtcEndpoint	398
19 NAT Traversal	403
19.1 WebRTC with ICE	403
19.2 RTP without ICE	403
20 WebRTC Statistics	407
20.1 Introduction	407
20.2 API description	407
20.3 Example	409
21 Debug Logging	411
21.1 Default levels	412
21.2 Verbose logging	412
21.3 Logs Location	415
21.4 Log Contents	415
21.5 Logging levels and components	416
22 Kurento Team	419
23 Contribution Guide	421
23.1 Did you find a bug?	422
23.2 Did you fix a bug?	422
23.3 Did you fix whitespace, format code, or make a purely cosmetic patch?	422
23.4 Do you intend to add a new feature or change an existing one?	422
23.5 Thanks for helping	422
24 Code of Conduct	423
25 Release Notes	425
25.1 7.0.1 (UNRELEASED)	425
25.2 7.0.0 (March 2023)	426
25.3 6.18.0 (September 2022)	430
25.4 6.17.0 (March 2022)	436
25.5 6.16.0 (March 2021)	437
25.6 6.15.0 (November 2020)	439
25.7 6.14.0 (June 2020)	443

25.8	6.13.2 (May 2020)	444
25.9	6.13.0 (December 2019)	447
25.10	6.12.0 (October 2019)	449
25.11	6.11.0 (July 2019)	450
25.12	6.10.0 (Apr 2019)	452
25.13	6.9.0 (Dec 2018)	456
25.14	6.8.1 (Oct 2018)	459
25.15	6.7.2 (May 2018)	460
26	Developer Guide	463
26.1	Introduction	464
26.2	Code repositories	464
26.3	Development 101	467
26.4	Build from sources	468
26.5	Install debug symbols	470
26.6	Run and debug with GDB	471
26.7	Work on a forked library	473
26.8	Create Deb packages	474
26.9	Unit Tests	475
26.10	How-To	476
27	Continuous Integration	477
28	Release Procedures	479
28.1	Introduction	480
28.2	Release order	482
28.3	FIRST: Open a new Release Process	484
28.4	Kurento Media Server	484
28.5	Kurento JavaScript client	488
28.6	Kurento Java client	491
28.7	Docker images	494
28.8	Kurento documentation	494
28.9	LAST: Close the Release Process	496
29	Security Hardening	501
29.1	Hardening validation	502
29.2	Hardening in Kurento	502
29.3	PIC/PIE in GCC	502
29.4	PIC/PIE in CMake	503
30	Writing this documentation	505
30.1	Building locally	506
30.2	Kurento documentation Style Guide	507
30.3	Sphinx documentation generator	508
30.4	Read the Docs builds	509
31	Testing	511
31.1	E2E Tests	512
31.2	Running Java tests	514
31.3	Kurento Testing Framework explained	523
32	Browser Details	527
32.1	Firefox	528
32.2	Safari	531
32.3	Chrome	531

32.4	Command-line	532
32.5	WebRTC JavaScript API	533
32.6	Browser MTU	533
32.7	Bandwidth Estimation	534
32.8	Video Encoding	534
33	Congestion Control (RMCAT)	537
33.1	Google Congestion Control	537
33.2	Meaning of REMB	537
34	H.264 video codec	539
34.1	Profiles and Levels	539
34.2	NAL Units (NALU)	541
34.3	SPS, PPS	541
34.4	GStreamer caps	542
34.5	GStreamer “codec_data”	542
35	Memory Fragmentation	545
35.1	Problem background	545
35.2	Solution	546
35.3	Using Jemalloc	546
35.4	Other suggestions	547
36	MP4 recording format	549
36.1	MP4 metadata issues	549
37	NAT Types and NAT Traversal	551
37.1	Basic Concepts	552
37.2	Types of NAT	553
37.3	Types of NAT in the Real World	555
37.4	NAT Traversal	556
38	RTP Streaming Commands	559
38.1	RTP sender examples	560
38.2	RTP receiver examples	564
38.3	SRTP examples	566
38.4	Additional Notes	569
39	Apple Safari	573
39.1	Codec issues	573
39.2	HTML policies for video playback	573
40	Self-Signed Certificates	575
40.1	Using a local domain	576
40.2	Trusting a self-signed certificate	577
41	Glossary	579
42	Indices and tables	587
	Index	589

Kurento Media Server (**KMS**) is a multimedia server package that can be used to develop advanced video applications for *WebRTC* platforms. It is an Open Source project, with source code released under the terms of [Apache License Version 2.0](#) and available on [GitHub](#).

Start here: *Introduction to Kurento* and *Getting Started*, and then learn to write Kurento applications with *Tutorials*.

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*

INTRODUCTION TO KURENTO

1.1 What is Kurento?

Kurento Media Server (**KMS**) is a multimedia server package that can be used to develop advanced video applications for *WebRTC* platforms. It is an Open Source project, with source code released under the terms of [Apache License Version 2.0](#) and available on [GitHub](#).

The most prominent characteristics of Kurento are these:

1.1.1 Modular Pipelines

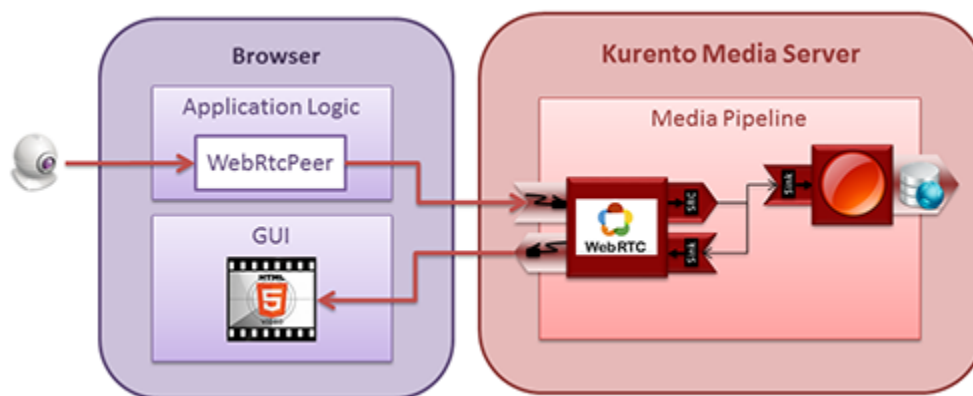


Fig. 1: Simple Example of a Media Pipeline

Kurento provides building blocks such as **WebRTC** and **RTP** senders & receivers, audio/video **mixers**, **media recording**, and more. These *Media Elements* are self-contained objects that hold a specific media capability; they are extremely easy to **compose** by inserting, activating, or deactivating them at any point in time, even when the media is already flowing.

It is also very easy to **extend** Kurento and write your own elements, which can then be integrated with the already existing ones!

Application developers use Kurento to control a so-called *Media Pipeline* with the desired Media Elements, effectively forming a fully customized architecture that is tailored to their needs. Several built-in modules are provided for **group communications**, **transcoding** of media formats, and **routing** of audiovisual flows.

Given the flexible modular approach that Kurento offers, it is possible to achieve both **Selective Forwarding Unit (SFU)** and **Multipoint Conferencing Unit (MCU)** application architectures.

1.1.2 Built-in Modules

Kurento exposes a rich toolbox of media elements as part of its API:

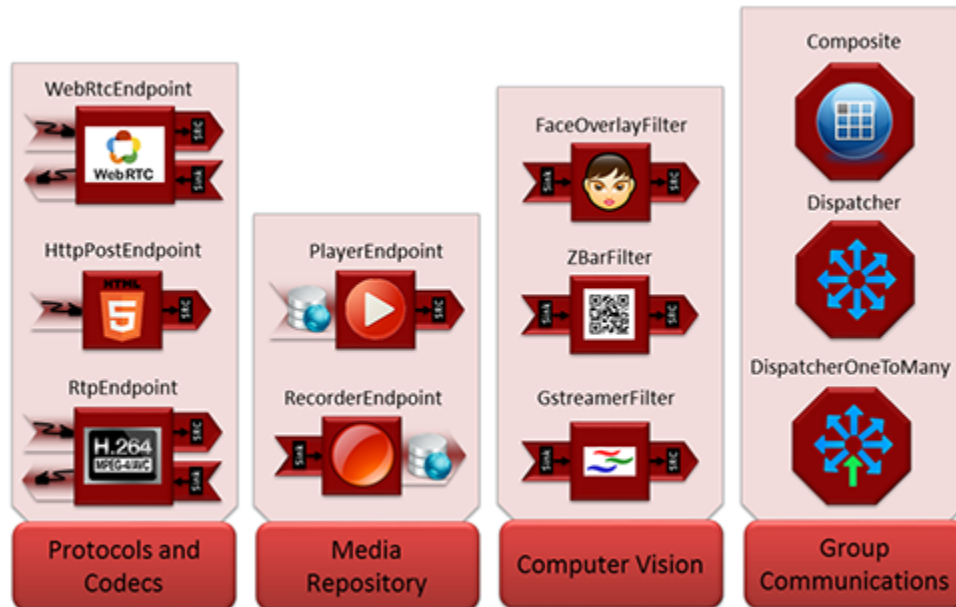


Fig. 2: Some Media Elements provided out of the box by Kurento

For example:

- The **WebRtcEndpoint** is able to send and receive *WebRTC* media streams.
- The **PlayerEndpoint** can be used to consume media from **RTSP**, **HTTP**, or local sources.
- The **RecorderEndpoint** can store media streams into a local or remote file system.
- The **FaceOverlayFilter** is a simple Computer Vision example that detects people's faces on the video streams, to add an overlay image on top of them.

Jump straight into the [Tutorials](#) to see practical examples of all these elements, used in applications built with kurento.

To learn more, read the section about [Kurento Modules](#). Furthermore, remember that Kurento has a plugin API that allows you to *write your own modules*!

1.1.3 JSON-RPC Protocol

KMS exposes all its API features through a JSON-RPC protocol called *Kurento Protocol*, which can be accessed directly through a WebSocket connection. For convenience, Kurento also offers Java and JavaScript SDKs: [Client API Reference](#). But you can use **any programming language**, simply writing your code directly against the protocol.

The picture below shows how to use Kurento in three scenarios:

- Using the Kurento JavaScript SDK directly from a WebRTC browser (only recommended for quick tests and development, not for production services).
- Using the Kurento Java SDK in a standalone Java EE Application Server. The web browser is a client of this application for things like HTML, and *WebRTC* signaling, while the application itself is client of KMS (using the Kurento Protocol to control KMS).

- Using the Kurento JavaScript SDK in a Node.js Application Server. Again, the web browser is a client of this application, while the application is client of KMS.

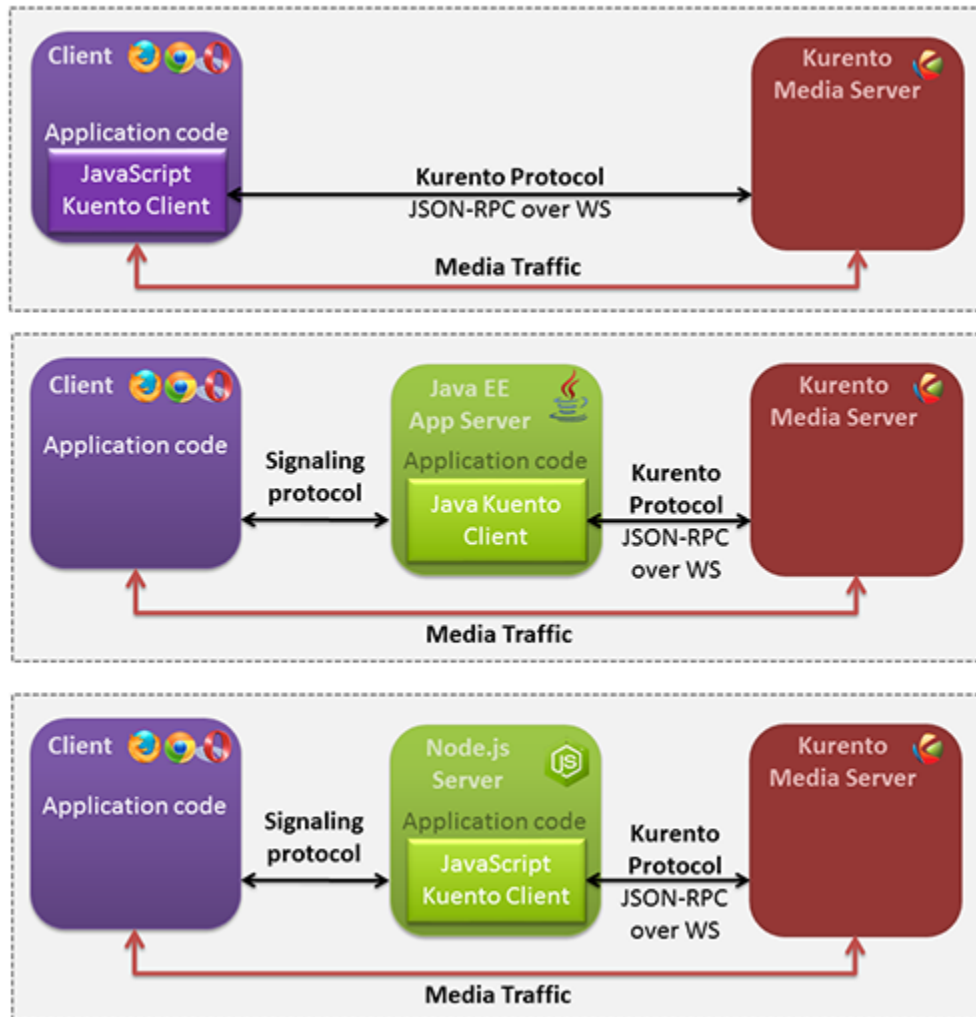


Fig. 3: Connection of Kurento Java and JavaScript SDKs to Kurento Media Server

Complete examples for the supported SDK technologies are described in [Tutorials](#).

1.2 Why a WebRTC media server?

WebRTC is a set of protocols and APIs that provide web browsers and mobile applications with Real-Time Communications (RTC) capabilities over peer-to-peer connections. It was conceived to allow connecting browsers without intermediate helpers or services, but in practice this P2P model falls short when trying to create more complex applications. For this reason, in most cases a central media server is required.

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

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").

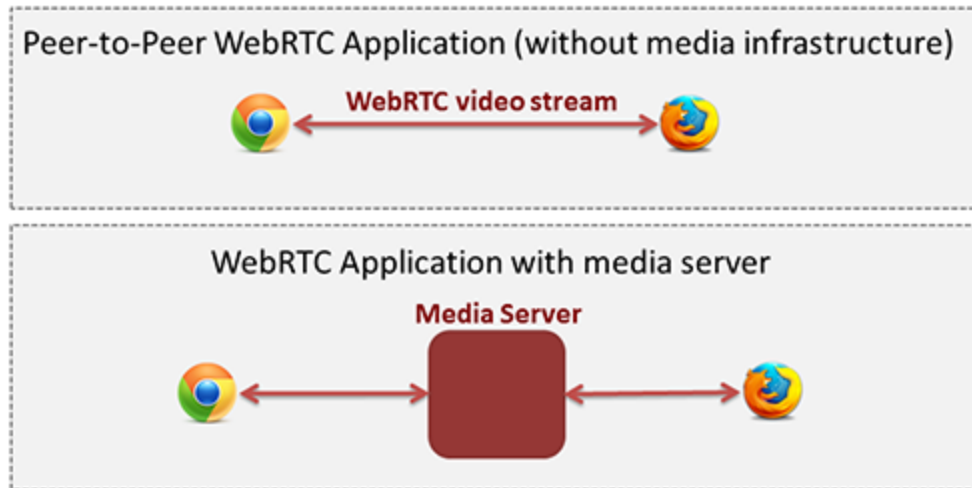


Fig. 4: *Peer-to-peer WebRTC approach vs. WebRTC through a media server*

- **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.

1.3 Why Kurento Media Server?

Kurento Media Server (KMS) can be used in the *WebRTC Media Server* model, to allow 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 (*both MCU and SFU functionality*) supporting media mixing and media routing/dispatching.
- Generic support for filters implementing **Computer Vision** and **Augmented Reality** algorithms.
- Media storage that supports writing operations for *WebM* and *MP4* and playing in all formats supported by *GStreamer*.
- Automatic media transcoding between any of the codecs supported by GStreamer, including VP8, H.264, H.263, AMR, OPUS, Speex, G.711, and more.

1.4 Kurento Design Principles

Kurento is designed based on the following main principles:

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

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

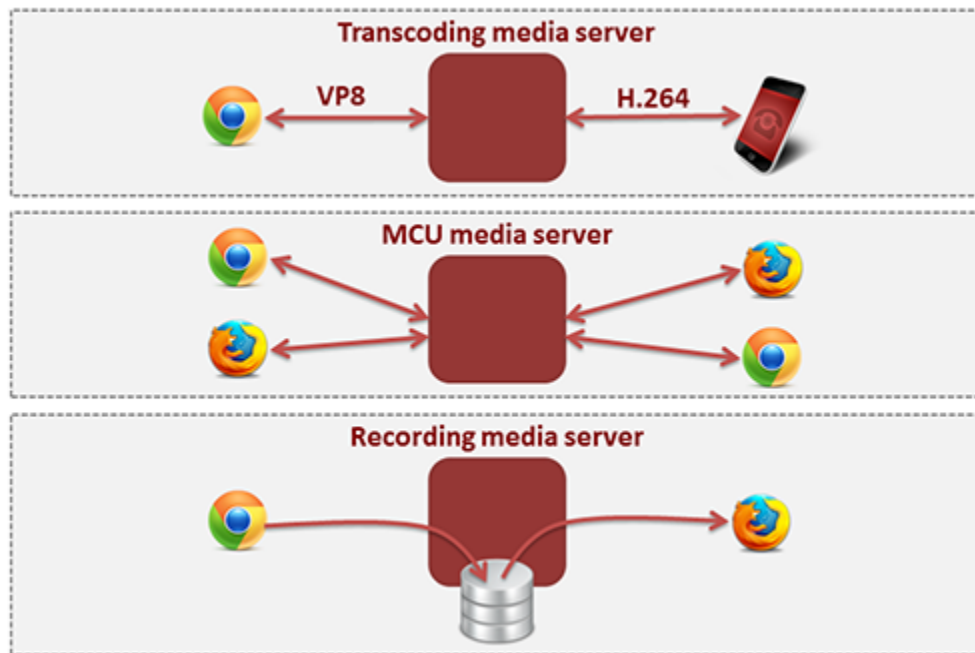


Fig. 5: Typical WebRTC Media Server capabilities

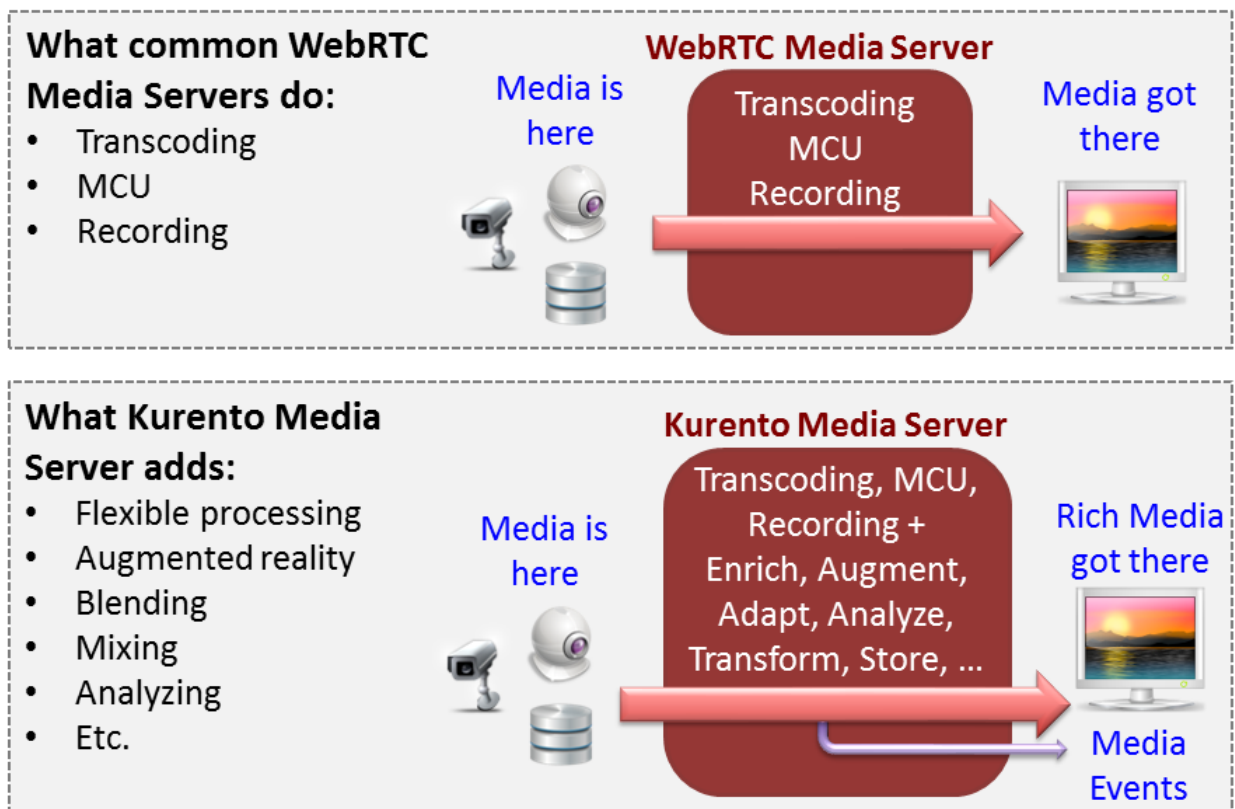


Fig. 6: Kurento Media Server capabilities

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

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

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

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

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

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

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

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

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

ABOUT OPENVIDU

OpenVidu is a framework that is based on Kurento, and encapsulates most of its features in order to **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 simpler API: just include the OpenVidu client-side library, and use 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.

OpenVidu is developed by the same team that created Kurento. Check it out: <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 [OpenVidu](#) project builds on top of Kurento to offer a simpler and easier to use solution that will save you time and development effort.

2. **Install KMS**

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

3. **Configure KMS**

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

4. **Write an Application**

Write an application that uses the *Kurento Protocol* to communicate with KMS and use its capabilities. The easiest way of doing this is to build on one of the provided *Kurento Clients*.

Have a look at 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

Table of Contents

- *Installation Guide*
 - *Amazon Web Services*
 - *Docker image*
 - * *Running*
 - *Why host networking?*
 - * *Docker Upgrade*
 - *Local Installation*
 - * *Running*
 - * *Local Upgrade*
 - *STUN/TURN server install*
 - *Check your installation*
 - * *Checking RTP port connectivity*

Kurento Media Server (KMS) is compiled and provided for installation by the Kurento team members, in a variety of forms. The only officially supported processor architecture is **64-bit x86**, so for other platforms (such as ARM) you will have to build from sources.

- *Using an EC2 instance* in the *Amazon Web Services* (AWS) cloud service is suggested to users who don't want to worry about properly configuring a server and all software packages, because the provided template does all this automatically.
- *The Kurento Docker image* allows to run KMS on top of any host machine, for example Fedora or CentOS. 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` allows to have total control of the installation process. It also means that it's easier to make mistakes, so we don't recommend this installation method. Do this only if you are a seasoned System Administrator.

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

The AWS *CloudFormation* template file for [Amazon Web Services](#) (AWS) 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. Now provide this Amazon S3 URL for the template:

```
https://s3-eu-west-1.amazonaws.com/aws.kurento.org/KMS-Coturn-cfn-7.0.0.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

The `kurento-media-server` Docker image is a nice *all-in-one* package for an easy quick start. It comes with all the default settings, which is enough to let you try the *Tutorials*.

If you need to insert or extract files from a Docker container, there is a variety of methods: You could use a `bind mount`; a `volume`; `cp` some files from an already existing container; change your `ENTRYPOINT` to generate or copy the files at startup; or `base FROM` this Docker image and build a new one with your own customizations. Check *Kurento in Docker* for an example of how to use `bind-mounts` to provide your own configuration files.

These are the exact contents of the image:

- A local `apt-get` installation of KMS, as described in *Local Installation*, plus all its extra plugins (chroma, platedetector, etc).
- Debug symbols installed, as described in *Install debug symbols*. This allows getting useful stack traces in case the KMS process crashes. If this happens, please [report a bug](#).
- All **default settings** from the local installation, as found in `/etc/kurento/`. For details, see *Configuration*.

4.2.1 Running

Docker allows to fine-tune how a container runs, so you'll want to read the *Docker run reference* and find out the command options that are needed for your project.

This is a good starting point, which runs the latest Kurento Media Server image with default options:

```
docker pull kurento/kurento-media-server:7.0.0

docker run -d --name kurento --network host \
    kurento/kurento-media-server:7.0.0
```

By default, KMS listens on the port **8888**. Clients wanting to control the media server using the *Kurento Protocol* should open a WebSocket connection to that port, either directly or by means of one of the provided *Client API Reference* SDKs.

The *health checker script* inside this Docker image does something very similar in order to check if the container is healthy.

Once the container is running, you can get its log output with the `docker logs` command:

```
docker logs --follow kms >"kms-$(date '+%Y%m%dT%H%M%S')'.log" 2>&1
```

For more details about KMS logs, check *Debug Logging*.

Why host networking?

Notice how our suggested `docker run` command uses `--network host`? Using *Host Networking* is recommended for software like proxies and media servers, because otherwise publishing large ranges of container ports would consume a lot of memory. You can read more about this issue in our *Troubleshooting Guide*.

The Host Networking driver **only works on Linux hosts**, so if you are using Docker for Mac or Windows then you'll need to understand that the Docker network gateway acts as a NAT between your host and your container. To use KMS without STUN (e.g. if you are just testing some of the *Tutorials*) you'll need to publish all required ports where KMS will listen for incoming data.

For example, if you use Docker for Mac and want to have KMS listening on the UDP port range **[5000, 5050]** (thus allowing incoming data on those ports), plus the TCP port **8888** for the *Client API Reference*, run:

```
docker run --rm \
  -p 8888:8888/tcp \
  -p 5000-5050:5000-5050/udp \
  -e KMS_MIN_PORT=5000 \
  -e KMS_MAX_PORT=5050 \
  kurento/kurento-media-server:7.0.0
```

4.2.2 Docker Upgrade

One of the nicest things about the Docker deployment method is that changing versions, or upgrading, is almost trivially easy. Just *pull* the new image version and use it to run your new container:

```
# Download the new image version:
docker pull kurento/kurento-media-server:7.0.0

# Create a new container based on the new version of KMS:
docker run [...] kurento/kurento-media-server:7.0.0
```

4.3 Local Installation

With this method, you will install Kurento Media Server from the native Ubuntu packages built by us.

Officially supported platforms: **Ubuntu 20.04 (Focal)** (64-bits).

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 \
  gnupg
```

2. Add the Kurento repository to your system configuration.

Run these commands:

```
# Get DISTRIB_* env vars.
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add Kurento repository key for apt-get.
sudo apt-key adv \
  --keyserver hkp://keyserver.ubuntu.com:80 \
  --recv-keys 234821A61B67740F89BFD669FC8A16625AFA7A83

# Add Kurento repository line for apt-get.
sudo tee "/etc/apt/sources.list.d/kurento.list" >/dev/null <<EOF
# Kurento Media Server - Release packages
deb [arch=amd64] http://ubuntu.openvidu.io/7.0.0 $DISTRIB_CODENAME main
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 \
kurento-media-server
```

This will install the release version of Kurento Media Server.

4.3.1 Running

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](#).

4.3.2 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 \
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`

- *kurento-module-core*
- *kurento-module-elements*
- *kurento-module-filters*
- *libnice10*
- *openh264*
- And more

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

4.4 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: [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:

- [Configuration](#)
- [How to install Coturn?](#)
- [How to test my STUN/TURN server?](#)

4.5 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](#).

4.5.1 Checking RTP port connectivity

This section explains how you can verify that Kurento Media Server can be reached from a remote client machine, in scenarios where **the media server is not behind a NAT**.

You will take the role of an end user application, such as a web browser, wanting to send audio and video to the media server. For that, we’ll use *Netcat* in the server, and either *Netcat* or *Ncat* in the client (because Ncat has more installation choices for Linux, Windows, and Mac clients).

The check proposed here will not work if the media server sits behind a NAT, because we are not punching holes in it (e.g. with STUN, see [When are STUN and TURN needed?](#)); doing so is outside of the scope for this section, but you could also do it by hand if needed (like shown in [Do-It-Yourself hole punching](#)).

First part: Server

Follow these steps on the machine where Kurento Media Server is running.

- First, install Netcat, which is available for most Linux distributions. For example:

```
# For Debian/Ubuntu:
sudo apt-get update ; sudo apt-get install netcat-openbsd
```

- Then, start a Netcat server, listening on any port of your choice:

```
# To test a TCP port:
nc -vnl <server_port>

# To test an UDP port:
nc -vnul <server_port>
```

Second part: Client

Now move to a client machine, and follow the next steps.

- Install either of Netcat or Ncat. On Linux, Netcat is probably available as a package. On MacOS and Windows, it might be easier to download a prebuilt installer from the [Ncat downloads page](#).
- Now, run Netcat or Ncat to connect with the server and send some test data. These examples use ncat, but the options are the same if you use nc:

```
# Linux, MacOS:
ncat -vn -p <client_port> <server_ip> <server_port> # TCP
ncat -vnu -p <client_port> <server_ip> <server_port> # UDP

# Windows:
ncat.exe -vn -p <client_port> <server_ip> <server_port> # TCP
ncat.exe -vnu -p <client_port> <server_ip> <server_port> # UDP
```

Note: The `-p <client_port>` is optional. We're using it here so the source port is well known, allowing us to expect it on the server's Ncat output, or in the IP packet headers if packet analysis is being done (e.g. with *Wireshark* or *tcpdump*). Otherwise, the O.S. would assign a random source port for our client.

- When the connection has been established, try typing some words and press Return or Enter. If you see the text appearing on the server side of the connection, **the test has been successful**.
- If the test is successful, you will see the client's source port in the server output. If this number is *different* than the `<client_port>` you used, this means that the client is behind a *Symmetric NAT*, and **a TURN relay will be required for WebRTC**.
- If the test data is not reaching the server, or the client command fails with a message such as `Ncat: Connection refused`, it means the connection has failed. You should review the network configuration to make sure that a firewall or some other filtering device is not blocking the connection. This is an indication that there are some issues in the network, which gives you a head start to troubleshoot missing media in your application.

For example: Assume you want to connect from the port `3000` of a client whose public IP is `198.51.100.2`, to the port `55000` of your server at `203.0.113.2`. This is what both client and server terminals could look like:

```
# CLIENT

$ ncat -vn -p 3000 203.0.113.2 55000
Ncat: Connected to 203.0.113.2:55000
(input) THIS IS SOME TEST DATA
```

```
# SERVER

$ nc -vnl 55000
Listening on 0.0.0.0 55000
Connection received on 198.51.100.2 3000
(output) THIS IS SOME TEST DATA
```

Notice how the server claims to have received a connection from the client's IP (`198.51.100.2`) and port (`3000`). This means that the client's NAT, if any, does not alter the source port of its outbound packets. If we saw here a different port, it would mean that the client's NAT is Symmetric, which usually requires using a TURN relay for WebRTC.

INSTALLING NIGHTLY BUILDS

Table of Contents

- *Installing Nightly Builds*
 - *Kurento Media Server*
 - * *Docker image*
 - * *Local Installation*
 - *Kurento Java Client*
 - * *Per-User config*
 - * *Per-Project config*
 - *Kurento JavaScript Client*

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

While official Kurento releases are published as Docker images and tagged with a release number, the latest development progress is tagged with *dev*: `kurento/kurento-media-server` (notice the `dev-*` tags). Other than that, these images behave exactly like the release ones. For usage instructions check out this section: [Docker image](#).

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 \
gnupg
```

2. Add the Kurento repository to your system configuration.

Run these commands:

```
# Get DISTRIB_* env vars.
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add Kurento repository key for apt-get.
sudo apt-key adv \
    --keyserver hkp://keyserver.ubuntu.com:80 \
    --recv-keys 234821A61B67740F89BFD669FC8A16625AFA7A83

# Add Kurento repository line for apt-get.
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 main
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 \
kurento-media-server
```

This will install the nightly version of Kurento Media Server.

5.2 Kurento Java Client

Development builds of Kurento Java packages are uploaded to the [GitHub Maven Repository](#).

This repo can be configured once per-User (by editing Maven's global `settings.xml`), or it can be added per-Project, to every `pom.xml`. We recommend using the first method.

For more information about adding a snapshots repository to Maven, check the official documentation: [Guide to Testing Development Versions of Plugins](#).

5.2.1 Per-User config

Add the snapshots repository to your Maven settings file: `$HOME/.m2/settings.xml`. If this file doesn't exist yet, you can copy it from `/etc/maven/settings.xml`, which offers a nice default template to get you started.

Edit the settings file to include this:

```
<settings>
  ...
  <profiles>
    <profile>
      <id>snapshot</id>
      <repositories>
        <repository>
          <id>kurento-github-download</id>
          <name>Kurento GitHub Maven packages (public access)</name>
          <url>https://public:103;hp_
→ fW4yqnUBB4LZvk8DE6VEbsu6XdnSBZ466WEJ@maven.pkg.github.com/kurento/*</url>
          <releases>
            <enabled>>false</enabled>
          </releases>
          <snapshots>
            <enabled>>true</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>kurento-github-download</id>
          <name>Kurento GitHub Maven packages (public access)</name>
          <url>https://public:103;hp_
→ fW4yqnUBB4LZvk8DE6VEbsu6XdnSBZ466WEJ@maven.pkg.github.com/kurento/*</url>
          <releases>
            <enabled>>false</enabled>
          </releases>
          <snapshots>
            <enabled>>true</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  ...
</settings>
```

Then use the `-Psnapshot` argument in your Maven commands, to enable the new profile. For example:

```
mvn -Psnapshot clean package
```

```
mvn dependency:get -Psnapshot -Dartifact='org.kurento:kurento-client:7.0.0-SNAPSHOT'
```

If you don't want to change all your Maven commands, it is possible to mark the profile as active by default. This way, a `-Psnapshot` argument will always be implicitly added, so all calls to Maven will already use the profile:

```
<settings>
...
<profiles>
  <profile>
    <id>snapshot</id>
    ...
  </profile>
</profiles>
<activeProfiles>
  <activeProfile>snapshot</activeProfile>
</activeProfiles>
...
</settings>
```

5.2.2 Per-Project config

This method consists on explicitly adding access to the snapshots repository, for a specific project. Open the project's `pom.xml` and include this:

```
<project>
...
<repositories>
  <repository>
    <id>kurento-github-download</id>
    <name>Kurento GitHub Maven packages (public access)</name>
    <url>https://public:&#103;hp_fw4yqnUBB4LZvk8DE6VEbsu6XdnSBZ466WEJ@maven.pkg.
    ↪github.com/kurento/*</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>kurento-github-download</id>
    <name>Kurento GitHub Maven packages (public access)</name>
    <url>https://public:&#103;hp_fw4yqnUBB4LZvk8DE6VEbsu6XdnSBZ466WEJ@maven.pkg.
    ↪github.com/kurento/*</url>
    <releases>
      <enabled>>false</enabled>
```

(continues on next page)

(continued from previous page)

```
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
    ...
  </project>
```

Afterwards, in the same `pom.xml`, look for the desired dependency and change its version to a snapshot one. For example:

```
<dependency>
  <groupId>org.kurento</groupId>
  <artifactId>kurento-client</artifactId>
  <version>7.0.0-SNAPSHOT</version>
</dependency>
```

5.3 Kurento JavaScript Client

Change the *dependencies* section in the application's *package.json*, to point directly to the development repository:

```
"dependencies": {
  "kurento-client": "git+https://github.com/Kurento/kurento-client-js.git#master"
}
```


CONFIGURATION

Table of Contents

- *Configuration*
 - *Debug Logging*
 - *STUN/TURN Server*
 - *Network Interface*
 - *WebRTC Bitrate*
 - *RTP Ports*
 - *Advanced Settings*
 - * *ICE-TCP*
 - * *External IP Address*
 - * *Maximum Transmission Unit*
 - * *WebRTC DTLS certificates*

Most (but not all, see below) of the settings in Kurento can be set statically in 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 *SdpEndpoints* (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.

For other settings not directly available in configuration files, make sure to read the Client API SDK docs, where all exposed methods are documented:

- [Kurento Client JavaDoc](#).
- [Kurento Client JsDoc](#).

The *Kurento Docker* images also accept some environment variables that map directly to settings in the above files. If this is not flexible enough, you can always use a [bind-mount](#) or [volume](#) with a different set of configuration files in `/etc/kurento/`. For some tips about these techniques, go to *Kurento in Docker*.

6.1 Debug Logging

KMS uses the environment variable `GST_DEBUG` to define the debug level of all underlying modules. Check *Debug Logging* for more information about this and other environment variables.

Set this variable to change the verbosity level of the log messages generated by KMS.

Local install

- Set environment variable `GST_DEBUG` in `/etc/default/kurento-media-server`.

Docker

- Pass environment variable `GST_DEBUG`:

```
docker run [...] \
  -e GST_DEBUG="Kurento*:5" \
  kurento/kurento-media-server:7.0.0
```

6.2 STUN/TURN Server

Read *When are STUN and TURN needed?* to learn about when you might need to use these, and *STUN/TURN server install* for guidance on how to install your own STUN/TURN server.

Local install

- Set values `stunServerAddress` and `stunServerPort` to use a STUN server, or set `turnURL` to use a TURN server; in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Docker

- Pass environment variables `KMS_STUN_IP` and `KMS_STUN_PORT` for STUN, or `KMS_TURN_URL` for TURN.

Client API

- Java: `setStunServerAddress` and `setStunServerPort` for STUN, or `setTurnUrl` for TURN.
- JavaScript: `setStunServerAddress` and `setStunServerPort` for STUN, or `setTurnUrl` for TURN.

6.3 Network Interface

To specify the network interface name(s) that KMS should use to communicate from the environment where it is running (either a physical machine, a virtual machine, a Docker container, etc.)

Local install

- Set value `networkInterfaces` in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Docker

- Pass environment variable `KMS_NETWORK_INTERFACES`.

Client API

- Java: `setNetworkInterfaces`.
- JavaScript: `setNetworkInterfaces`.

6.4 WebRTC Bitrate

The default **MaxVideoSendBandwidth** range of the `WebRtcEndpoint` is a VERY conservative one, and leads to a low maximum video quality. Most applications will probably want to increase this to higher values such as 2000 kbps (2 mbps): [Java](#), [JavaScript](#).

There are several ways to override the default settings for variable bitrate and network bandwidth estimation:

- `setMinVideoRecvBandwidth` / `setMaxVideoRecvBandwidth`
- `setMinVideoSendBandwidth` / `setMaxVideoSendBandwidth`
- `setEncoderBitrate` / `setMinEncoderBitrate` / `setMaxEncoderBitrate`
 - This setting is also configurable in `/etc/kurento/modules/kurento/MediaElement.conf.ini`.

6.5 RTP Ports

These two parameters define the minimum and maximum ports that Kurento Media Server will bind to (listen) in order to receive remote RTP packets. This affects the operation of both `RtpEndpoint` and `WebRtcEndpoint`.

Plain RTP (`RtpEndpoint`) needs 2 ports for each media kind: an even port is used for RTP, and the next odd port is used for RTCP. WebRTC (`WebRtcEndpoint`) uses RTCP Multiplexing (`rtcp-mux`) when possible, so it only uses 1 port for each media kind.

Local install

- Set values `minPort`, `maxPort` in `/etc/kurento/modules/kurento/BaseRtpEndpoint.conf.ini`.

Docker

- Pass environment variables `KMS_MIN_PORT`, `KMS_MAX_PORT`.

6.6 Advanced Settings

These settings are only provided for advanced users who know what they are doing and why they need them. For most cases, the default values are good enough for most users.

6.6.1 ICE-TCP

ICE-TCP is what allows WebRTC endpoints to exchange ICE candidates that use the TCP protocol; in other words, the feature of using TCP instead of UDP for WebRTC communications.

If you have a well known scenario and you are 100% sure that UDP will work, then disabling TCP provides slightly faster times when establishing WebRTC sessions. I.e., with ICE-TCP disabled, the time between joining a call and actually seeing the video will be shorter.

Of course, if you cannot guarantee that UDP will work in your network, then **you should leave this setting enabled**, which is the default. Otherwise, UDP might fail and there would be no TCP fallback for WebRTC to work.

Local install

- Set value `iceTcp` to 1 (ON) or 0 (OFF) in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Docker

- Set environment variable `KMS_ICE_TCP` to 1 (ON) or 0 (OFF).

Client API

- Java: `setIceTcp`.
- JavaScript: `setIceTcp`.

6.6.2 External IP Address

When this feature is used, all of the Kurento IPv4 and/or IPv6 ICE candidates are mangled to contain the given address. This can speed up WebRTC connection establishment in scenarios where the external or public IP is already well known, also having the benefit that STUN won't be needed *for the media server*.

Local install

- Set values `externalIPv4`, `externalIPv6` in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Docker

- Pass environment variables `KMS_EXTERNAL_IPV4`, `KMS_EXTERNAL_IPV6`.
- If the special value `auto` is used, then the container will auto-discover its own public IP address by performing a DNS query to some of the well established providers (OpenDNS, Google, Cloudflare). You can review the script here: [getmyip.sh](https://github.com/kurento/kurento/blob/master/scripts/getmyip.sh). In cases where these services are not reachable, the external IP parameters are left unset.

Client API

- Java: `setExternalIPv4`.
- JavaScript: `setExternalIPv4`.

6.6.3 Maximum Transmission Unit

The MTU is a hard limit on the size that outbound packets will have. For some users it is important being able to lower the packet size in order to prevent fragmentation.

For the vast majority of use cases it is better to use the default MTU value of 1200 Bytes, which is also the default value in most popular implementations of WebRTC (see [Browser MTU](#)).

Local install

- Set value `mtu` in `/etc/kurento/modules/kurento/BaseRtpEndpoint.conf.ini`.

Docker

- Pass environment variable `KMS_MTU`.

Client API

- Java: `setMtu`.
- JavaScript: `setMtu`.

6.6.4 WebRTC DTLS certificates

By default, Kurento uses a different self-signed certificate for every `WebRtcEndpoint` (see [Media Plane security \(DTLS\)](#)). If you want or need to use the same cert for every endpoint, first join both your certificate (chain) file(s) and the private key with a command such as this one:

```
# Make a single file to be used with Kurento Media Server.
cat cert.pem key.pem >cert+key.pem
```

Then, configure the path to `cert+key.pem`:

Local install

- Set either of `pemCertificateRSA` or `pemCertificateECDSA` with the path to your certificate file in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Docker

- Pass environment variables `KMS_PEM_CERTIFICATE_RSA` or `KMS_PEM_CERTIFICATE_ECDSA` with the path *inside the container*. Also, make sure the file is actually found in that path; normally you would do that with a `bind-mount`, a Docker volume, or a custom Docker image. For more information and examples, check [Kurento in Docker](#).

TUTORIALS

Table of Contents

- *Tutorials*
 - *Hello World*
 - *WebRTC Magic Mirror*
 - *RTP Receiver*
 - *WebRTC One-To-Many broadcast*
 - *WebRTC One-To-One video call*
 - *WebRTC One-To-One video call with recording and filtering*
 - *WebRTC Many-To-Many video call (Group Call)*
 - *Media Elements metadata*
 - *WebRTC Media Player*
 - *WebRTC outgoing Data Channels*
 - *WebRTC incoming Data Channel*
 - *WebRTC recording*
 - *WebRTC statistics*
 - *Chroma Filter*
 - *Crowd Detector Filter*
 - *Plate Detector Filter*
 - *Pointer Detector Filter*

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 \
    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.

7.1 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)

7.1.1 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 [Introduction to Kurento](#) 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.git
cd kurento/tutorials/java/hello-world/
git checkout 7.0.0
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:

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 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.

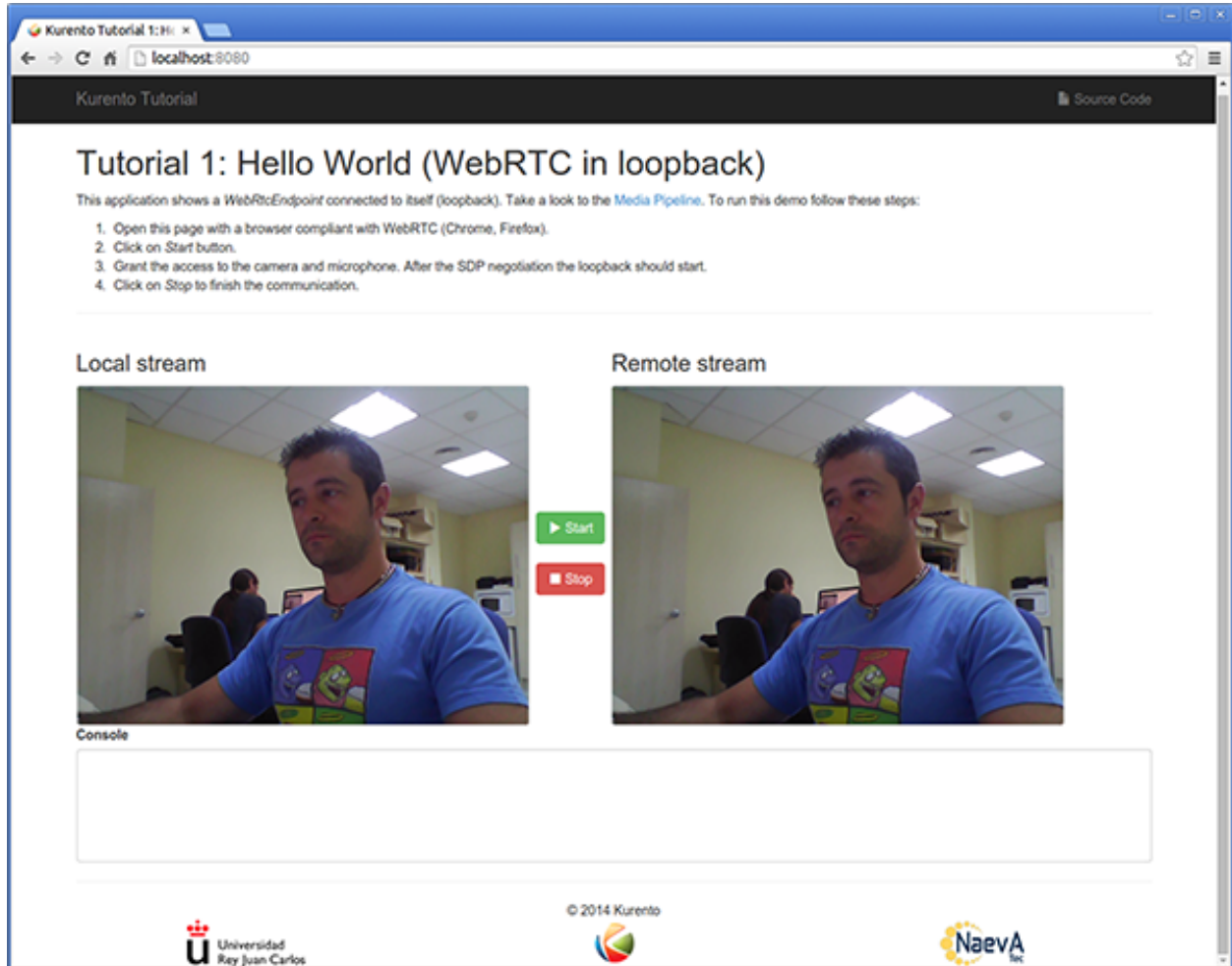


Fig. 1: Kurento Hello World Screenshot: WebRTC in loopback

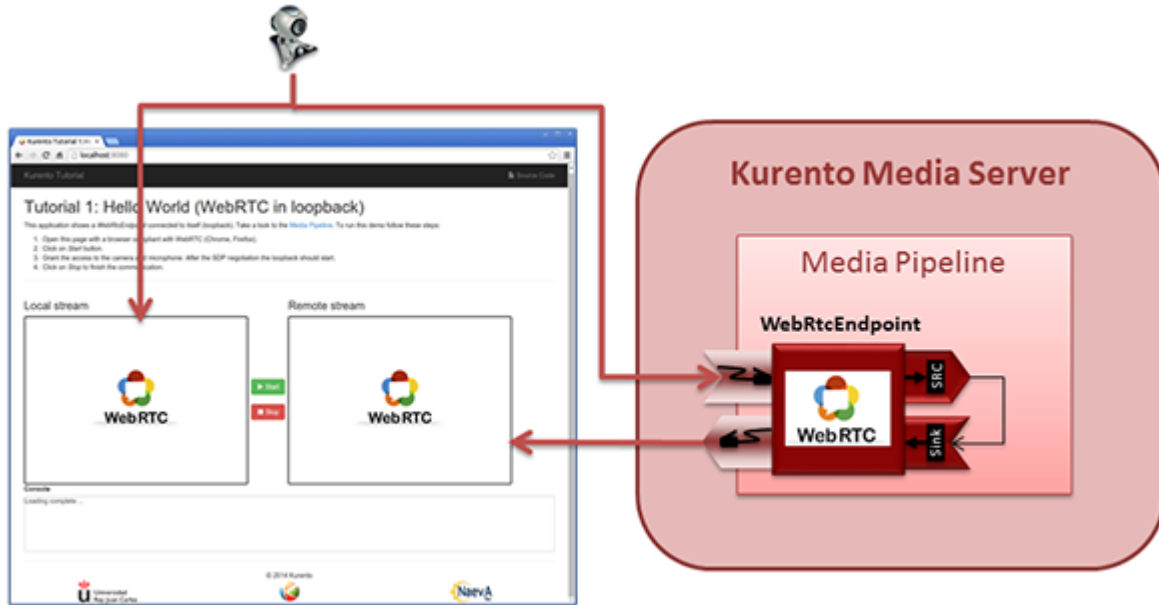


Fig. 2: Kurento Hello World Media Pipeline in context

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 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 HelloWorldApp implements WebSocketConfigurer {
    @Bean
    public HelloWorldHandler handler() {
```

(continues on next page)

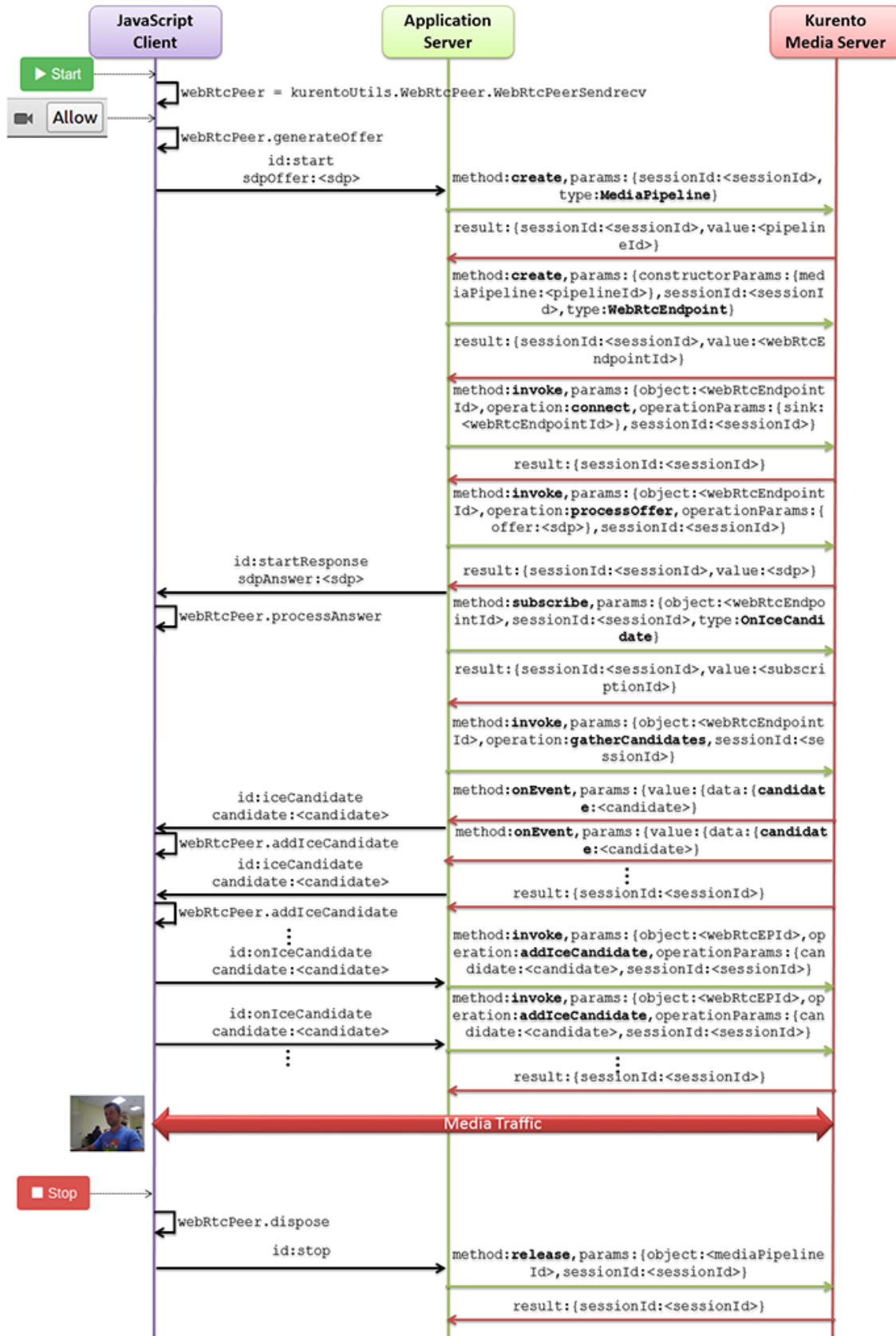


Fig. 3: Complete sequence diagram of Kurento Hello World (WebRTC in loopback) demo

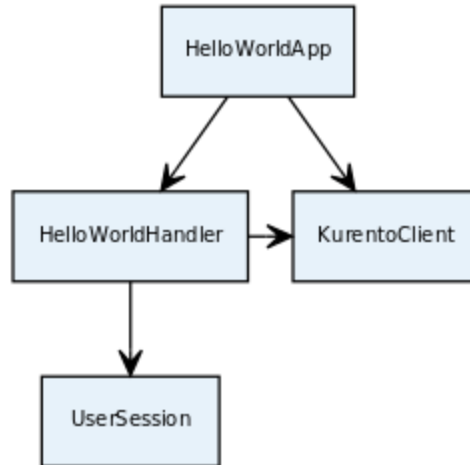


Fig. 4: Server-side class diagram of the HelloWorld app

(continued from previous page)

```

    return new HelloWorldHandler();
}

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

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

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

```

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

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

```

public void handleTextMessage(WebSocketSession session, TextMessage message)
    throws Exception {
    [...]
    switch (messageId) {
        case "start":
            start(session, jsonMessage);
    }
}

```

(continues on next page)

(continued from previous page)

```

        break;
    case "stop": {
        stop(session);
        break;
    }
    case "onIceCandidate":
        onRemoteIceCandidate(session, jsonMessage);
        break;
    default:
        sendError(session, "Invalid message, ID: " + messageId);
        break;
    }
    [...]
}

```

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

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

```

final MediaPipeline pipeline = kurento.createMediaPipeline();

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

webRtcEp.connect(webRtcEp);

```

- **Create event listeners.** All objects managed by Kurento have the ability to emit several types of events, as explained in *Endpoint Events*. 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.

```

// Common events for all objects that inherit from BaseRtpEndpoint
addErrorListener(
    new EventListener<ErrorEvent>() { ... });
addMediaFlowInStateChangedListener(
    new EventListener<MediaFlowInStateChangedEvent>() { ... });
addMediaFlowOutStateChangedListener(
    new EventListener<MediaFlowOutStateChangedEvent>() { ... });
addConnectionStateChangedListener(
    new EventListener<ConnectionStateChangedEvent>() { ... });
addMediaStateChangedListener(
    new EventListener<MediaStateChangedEvent>() { ... });
addMediaTranscodingStateChangedListener(
    new EventListener<MediaTranscodingStateChangedEvent>() { ... });

// Events specific to objects of class WebRtcEndpoint
addIceCandidateFoundListener(
    new EventListener<IceCandidateFoundEvent>() { ... });
addIceComponentStateChangedListener(

```

(continues on next page)

(continued from previous page)

```

    new EventListener<IceComponentStateChangedEvent>() { ... });
addIceGatheringDoneListener(
    new EventListener<IceGatheringDoneEvent>() { ... });
addNewCandidatePairSelectedListener(
    new EventListener<NewCandidatePairSelectedEvent>() { ... });

```

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

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

```

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

```

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

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

Client-Side Logic

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

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

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

```

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

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

    switch (parsedMessage.id) {
    case 'startResponse':
        startResponse(parsedMessage);
        break;

```

(continues on next page)

(continued from previous page)

```

    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);
}

```

(continues on next page)

(continued from previous page)

```
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](#).

7.1.2 JavaScript - Hello world

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 [Introduction to Kurento](#) 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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/hello-world/
git checkout 7.0.0
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 a screenshot of this demo running:

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

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

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](#).

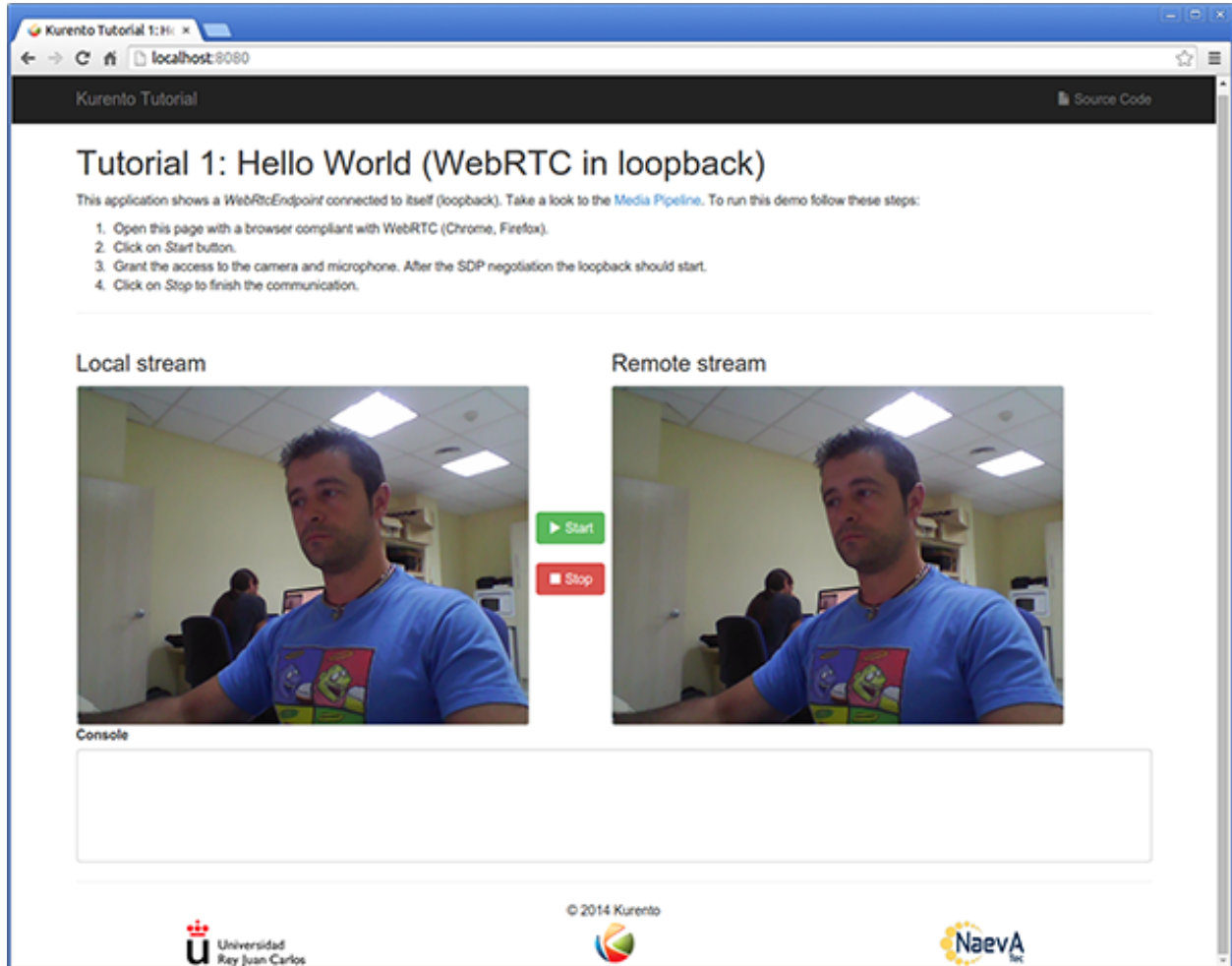


Fig. 5: Kurento Hello World Screenshot: WebRTC in loopback

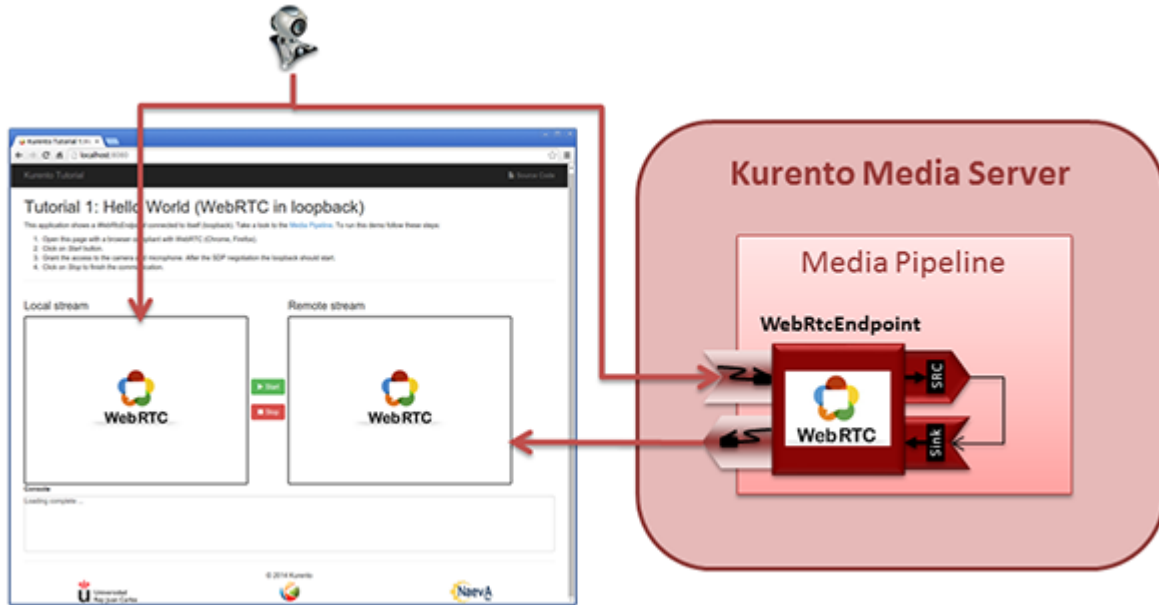


Fig. 6: Kurento Hello World Media Pipeline in context

JavaScript Logic

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

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

In addition, these two JavaScript libraries are also required:

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

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

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

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

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

(continues on next page)

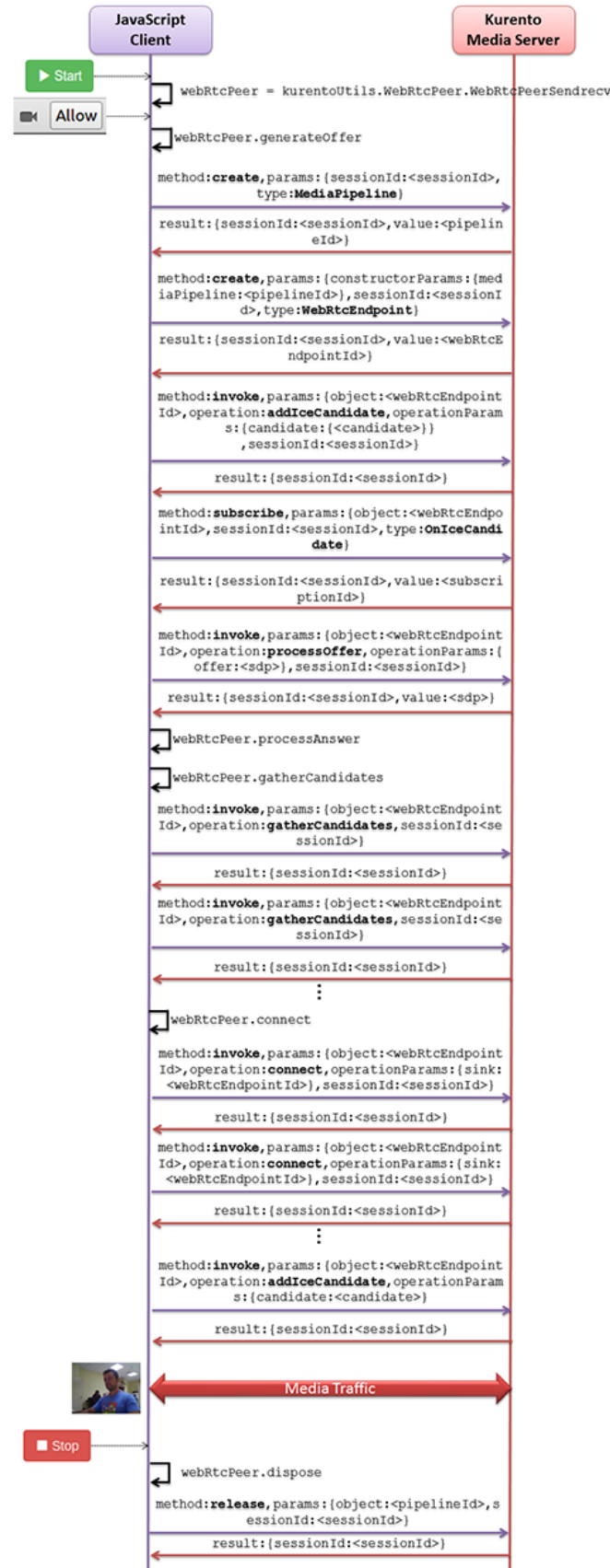


Fig. 7: Complete sequence diagram of Kurento Hello World (WebRTC in loopback) demo

(continued from previous page)

```

    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);

```

(continues on next page)

(continued from previous page)

```
        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": "7.0.0",
  "kurento-utils": "7.0.0"
}
```

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

```
bower install
```

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

7.1.3 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 [Introduction to Kurento](#) 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.js 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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-node/hello-world/
git checkout 7.0.0
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

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 a screenshot of this demo running:

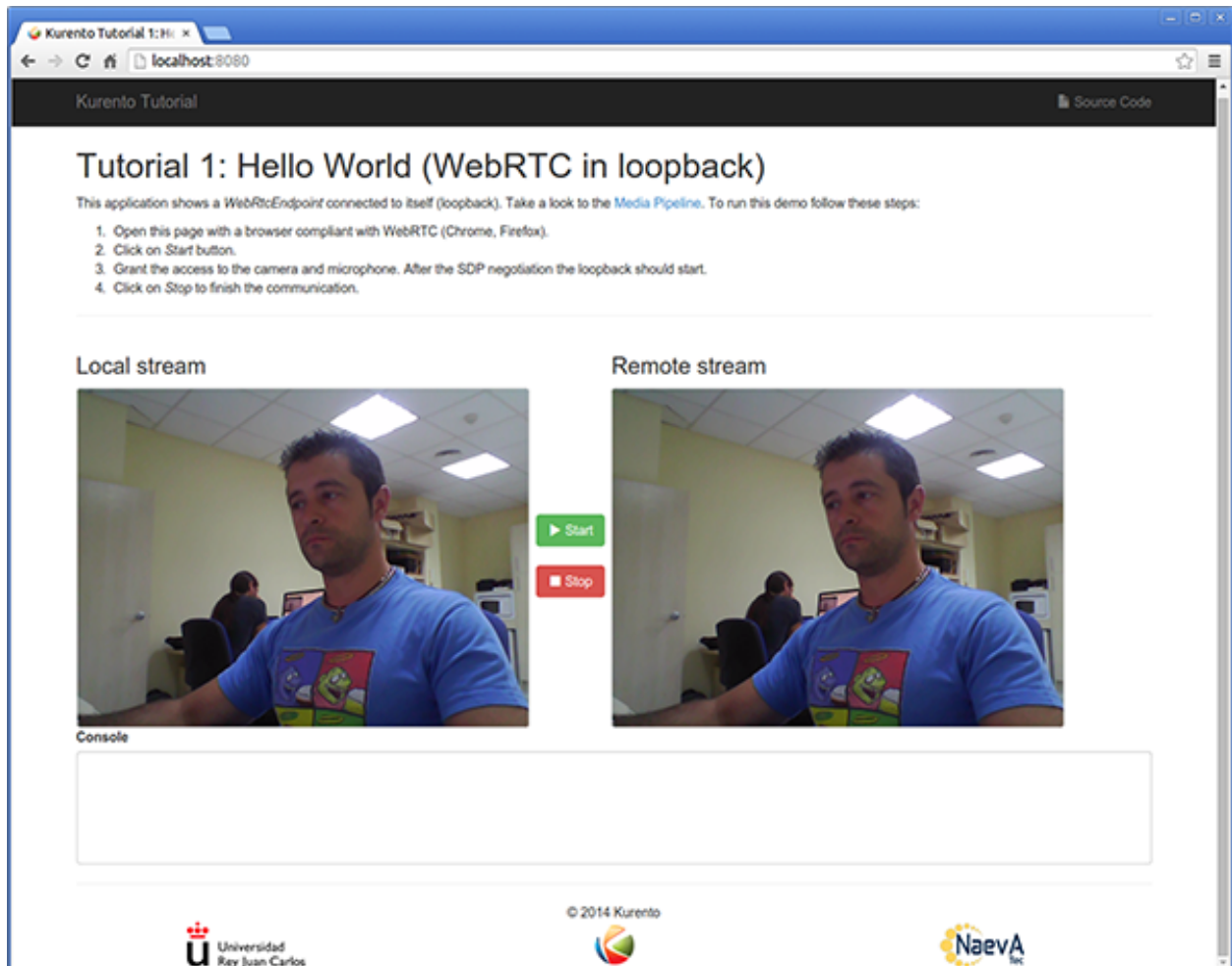


Fig. 8: Kurento Hello World Screenshot: WebRTC in loopback

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

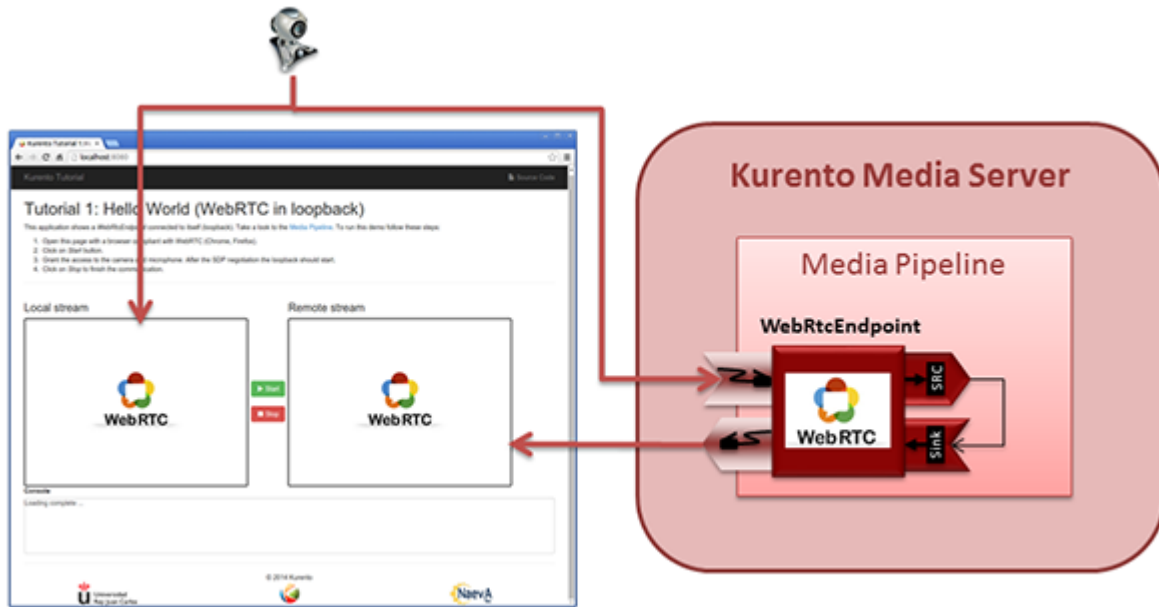


Fig. 9: Kurento Hello World Media Pipeline in context

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`).

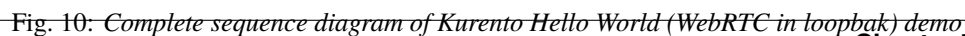
```
var ws = require('ws');

[...]
```

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

/*
 * Management of WebSocket messages
```

(continues on next page)



(continued from previous page)

```

*/
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;
        }
    });
}

```

(continues on next page)

(continued from previous page)

```

        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 just need a single *WebRtcEndpoint* connected to itself (i.e. in loopback). These functions are called in the *start* function, which is fired when the *start* message is received:

```

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

```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

                webRtcEndpoint.on('IceCandidateFound', 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);
                });
            });
        });
    });
}

```

(continues on next page)

(continued from previous page)

```

        });

        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 `IceCandidateFound` 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]) {

```

(continues on next page)

(continued from previous page)

```

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

```

Client-Side Logic

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

```

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

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

[...]

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

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

```

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

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

[...]

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

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

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

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

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

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

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

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

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

Dependencies

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

```
"dependencies": {  
  [...]  
  "kurento-client" : "7.0.0"  
}
```

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" : "7.0.0"  
}
```

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

7.2 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.

7.2.1 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.git  
cd kurento/tutorials/java/magic-mirror/  
git checkout 7.0.0  
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:

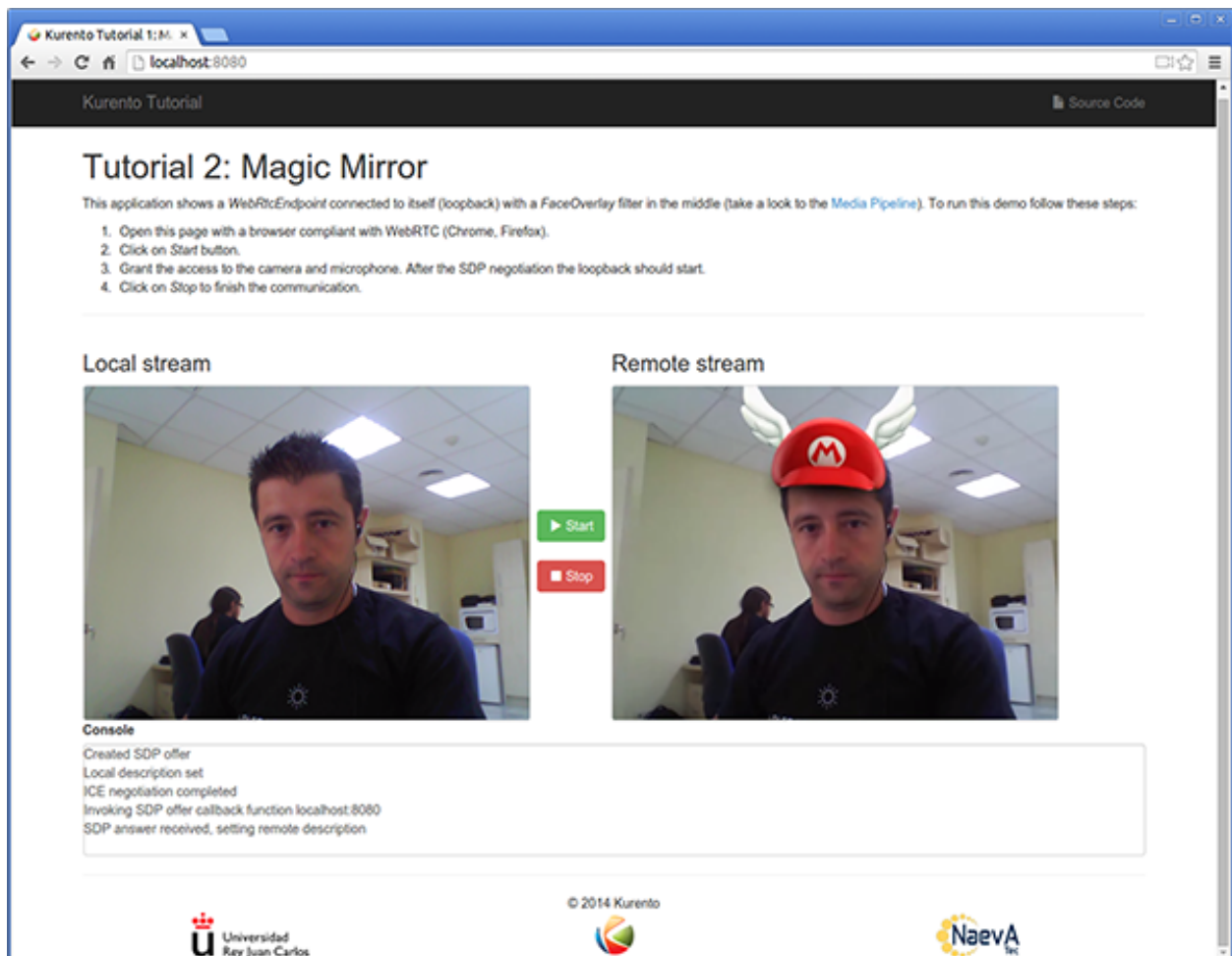


Fig. 11: Kurento Magic Mirror Screenshot: WebRTC with filter in loopback

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*).

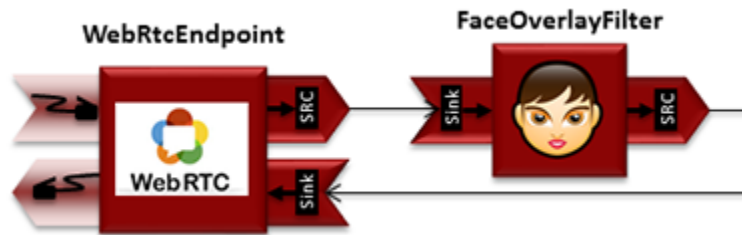


Fig. 12: *WebRTC with filter in loopback Media Pipeline*

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 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.

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:

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:

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.

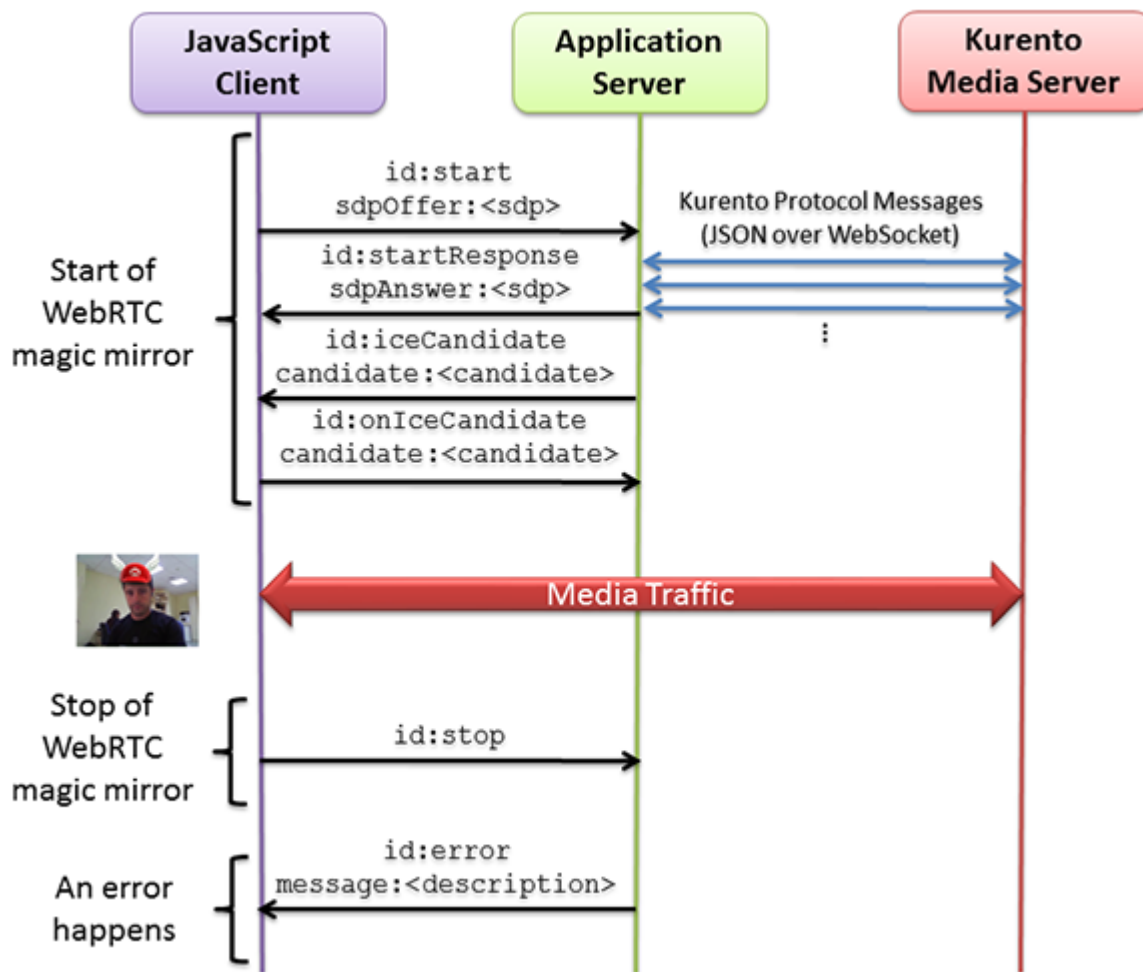


Fig. 13: One to one video call signaling protocol

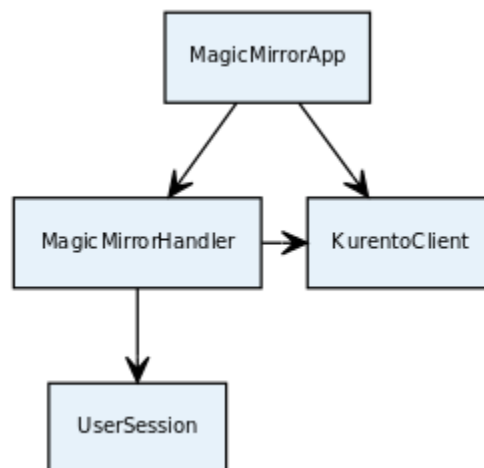


Fig. 14: Server-side class diagram of the MagicMirror app


```

@EnableWebSocket
@SpringBootApplication
public class MagicMirrorApp implements WebSocketConfigurer {

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

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

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

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

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

```

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

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

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

```

public class MagicMirrorHandler extends TextWebSocketHandler {

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

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

    @Autowired
    private KurentoClient kurento;

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

```

(continues on next page)

(continued from previous page)

```

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

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

    UserSession user = users.get(session.getId());
    if (user != null) {
        IceCandidate candidate = new IceCandidate(jsonCandidate.get("candidate").
↪getAsString(),
        jsonCandidate.get("sdpMid").getAsString(), jsonCandidate.get(
↪"sdpMLineIndex").getAsInt());
        user.addCandidate(candidate);
    }
    break;
}
default:
    sendError(session, "Invalid message with id " + jsonMessage.get("id").
↪getAsString());
    break;
}
}

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

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

```

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

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // User session
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);

```

(continues on next page)

(continued from previous page)

```

WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
user.setWebRtcEndpoint(webRtcEndpoint);
users.put(session.getId(), user);

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

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

String appServerUrl = System.getProperty("app.server.url", MagicMirrorApp.DEFAULT_
↳APP_SERVER_URL);
faceOverlayFilter.setOverlaidImage(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.

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

Client-Side

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

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

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

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

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
            onError("Error message from server: " + parsedMessage.message);
            break;
    }
}
```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

    var message = {
        id: 'onIceCandidate',

```

(continues on next page)

(continued from previous page)

```
        candidate: candidate
    };
    sendMessage(message);
}
```

Dependencies

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

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

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

7.2.2 JavaScript - Magic Mirror

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/magic-mirror/
git checkout 7.0.0
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](#).

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.

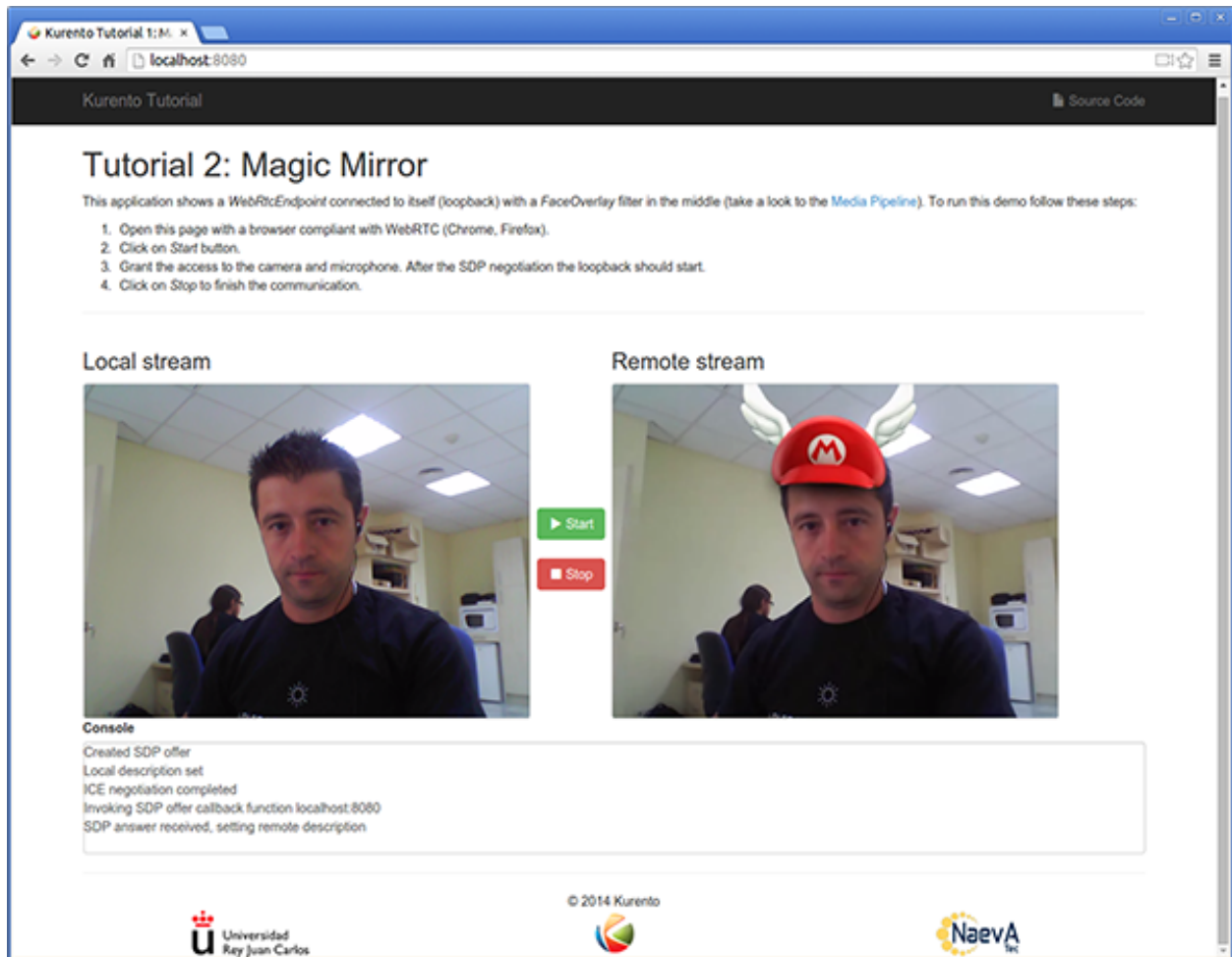


Fig. 15: Kurento Magic Mirror Screenshot: WebRTC with filter in loopback

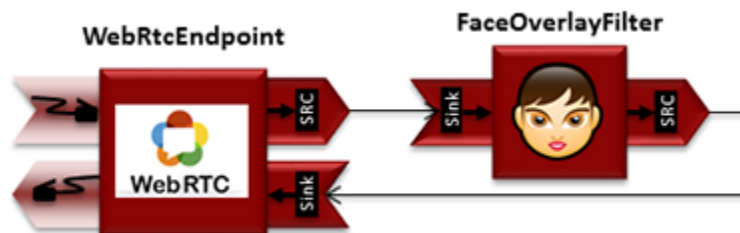


Fig. 16: WebRTC with filter in loopback Media Pipeline

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

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

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

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

    [...]
})
```

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

Inside this function, a call to *generateOffer* is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the *KurentoClient* class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port 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*, 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 *FaceOverlay-Filter*. Then, these media elements are interconnected:

```

pipeline.create('WebRtcEndpoint', function(error, webRtcEp) {
  if (error) return onError(error);

  setIceCandidateCallbacks(webRtcPeer, webRtcEp, onError)

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

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

  pipeline.create('FaceOverlayFilter', function(error, filter) {
    if (error) return onError(error);

    filter.setOverlaidImage(args.hat_uri, -0.35, -1.2, 1.6, 1.6,
    function(error) {
      if (error) return onError(error);
    });

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

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

```

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

```

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

```

Dependencies

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

```

"dependencies": {
  "kurento-client": "7.0.0",
  "kurento-utils": "7.0.0"
}

```

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

7.2.3 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.js 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* in your system. In an Ubuntu machine, you can install it as follows:

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

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.git  
cd kurento/tutorials/javascript-node/magic-mirror/  
git checkout 7.0.0  
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:

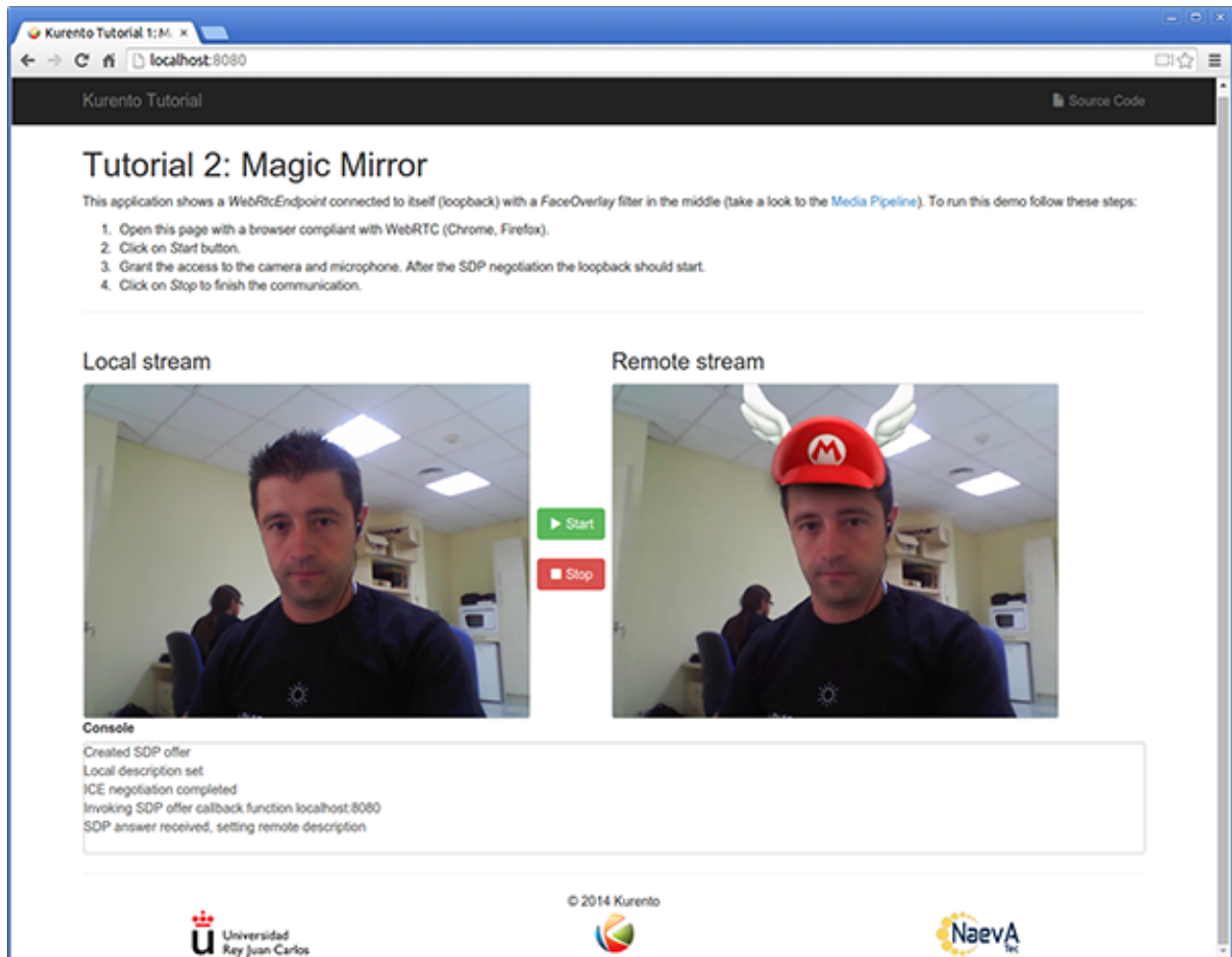


Fig. 17: Kurento Magic Mirror Screenshot: WebRTC with filter in loopback

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*).

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

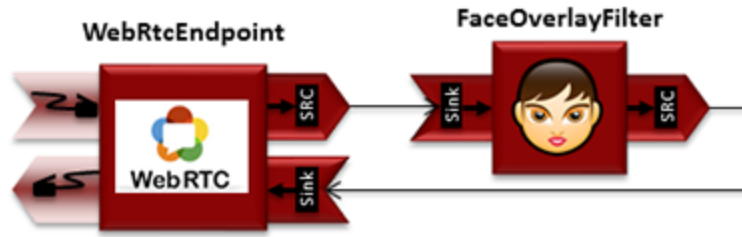


Fig. 18: WebRTC with filter in loopback Media Pipeline

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

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

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

Application Server Logic

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

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

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

[...]
```

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

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

(continues on next page)

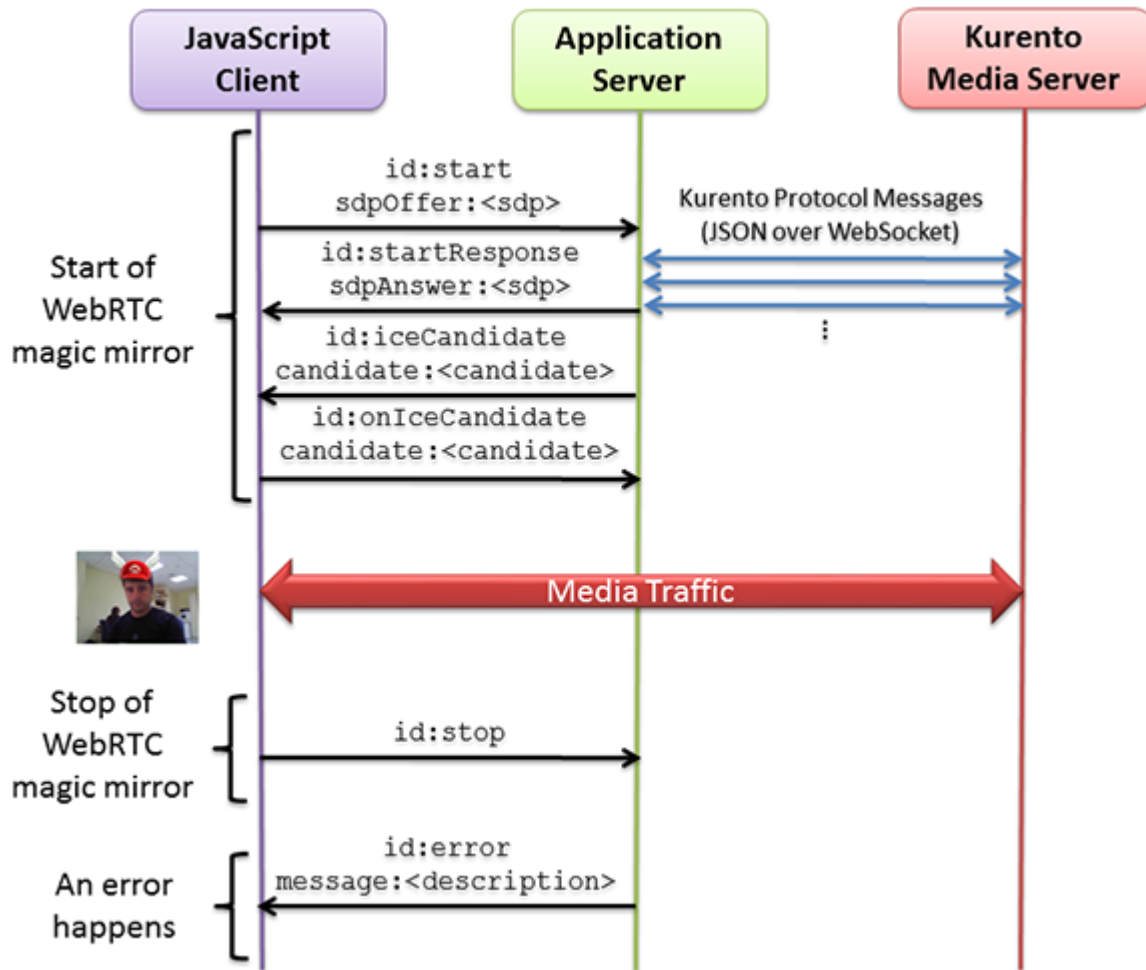


Fig. 19: One to one video call signaling protocol

(continued from previous page)

```

});

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 *Kuren-*

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

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

      createMediaElements(pipeline, ws, function(error, webRtcEndpoint) {
```

(continues on next page)

(continued from previous page)

```

        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('IceCandidateFound', 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) {

```

(continues on next page)

(continued from previous page)

```

    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 `IceCandidateFound` function. These candidates are stored in a queue when the `webRtcEndpoint` is not available yet. Then these candidates are added to the media element by calling to the `addIceCandidate` method.

```

var candidatesQueue = {};

[...]

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

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

```

Client-Side Logic

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

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

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

[...]

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

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

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

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

[...]
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

Dependencies

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

```
"dependencies": {  
  [...]  
  "kurento-client" : "7.0.0"  
}
```

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" : "7.0.0"  
}
```

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

7.3 RTP Receiver

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

7.3.1 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.git
cd kurento/tutorials/java/rtp-receiver/
git checkout 7.0.0
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:

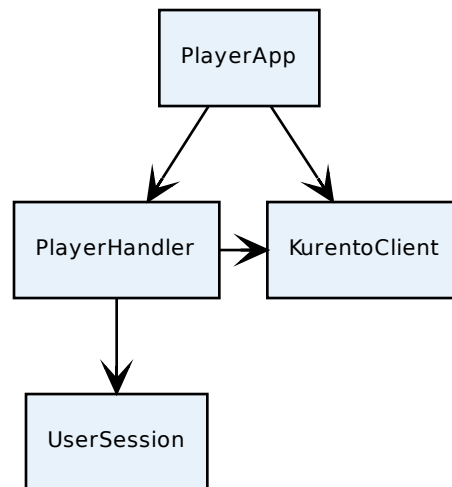


Fig. 20: Server-side class diagram of the Application Server

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.

7.4 WebRTC One-To-Many broadcast

Video broadcasting for *WebRTC*. One peer transmits a video stream and N peers receive it.

7.4.1 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.git
cd kurento/tutorials/java/one2many-call/
git checkout 7.0.0
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 a 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

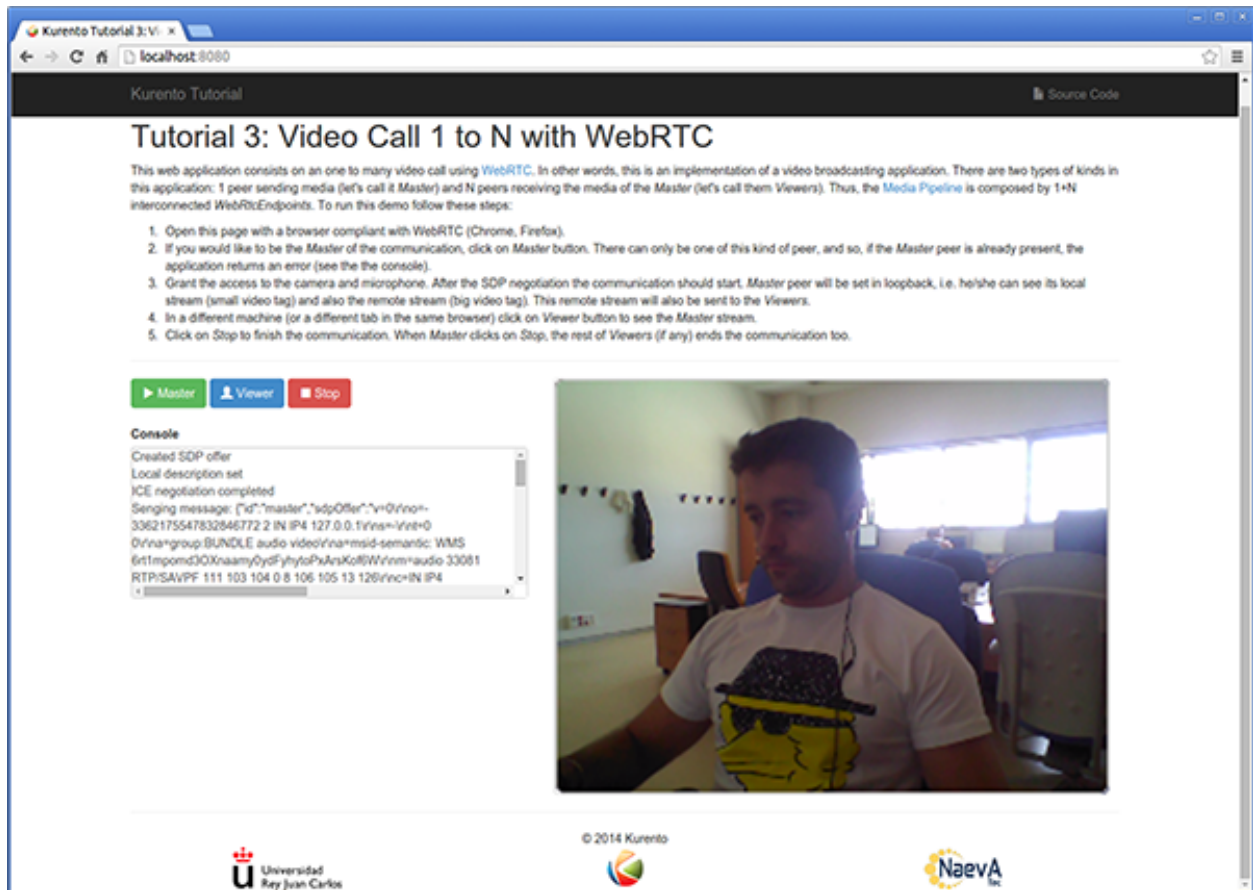


Fig. 21: One to many video call screenshot

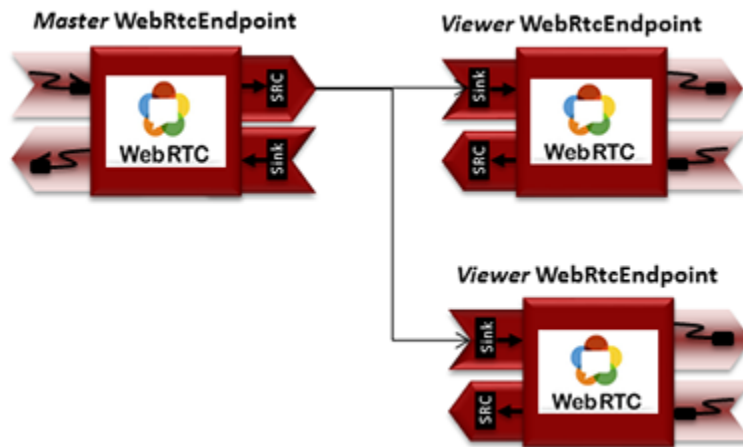


Fig. 22: One to many video call Media Pipeline

is three-tier. To communicate these entities two WebSockets are used. First, a WebSocket is created between client and server-side to implement a custom signaling protocol. Second, another WebSocket is used to perform the communication between the Kurento Java Client and the Kurento Media Server. This communication is implemented by the **Kurento Protocol**. For further information, please see this [page](#).

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

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

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

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

Application Server Logic

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

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

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

(continues on next page)

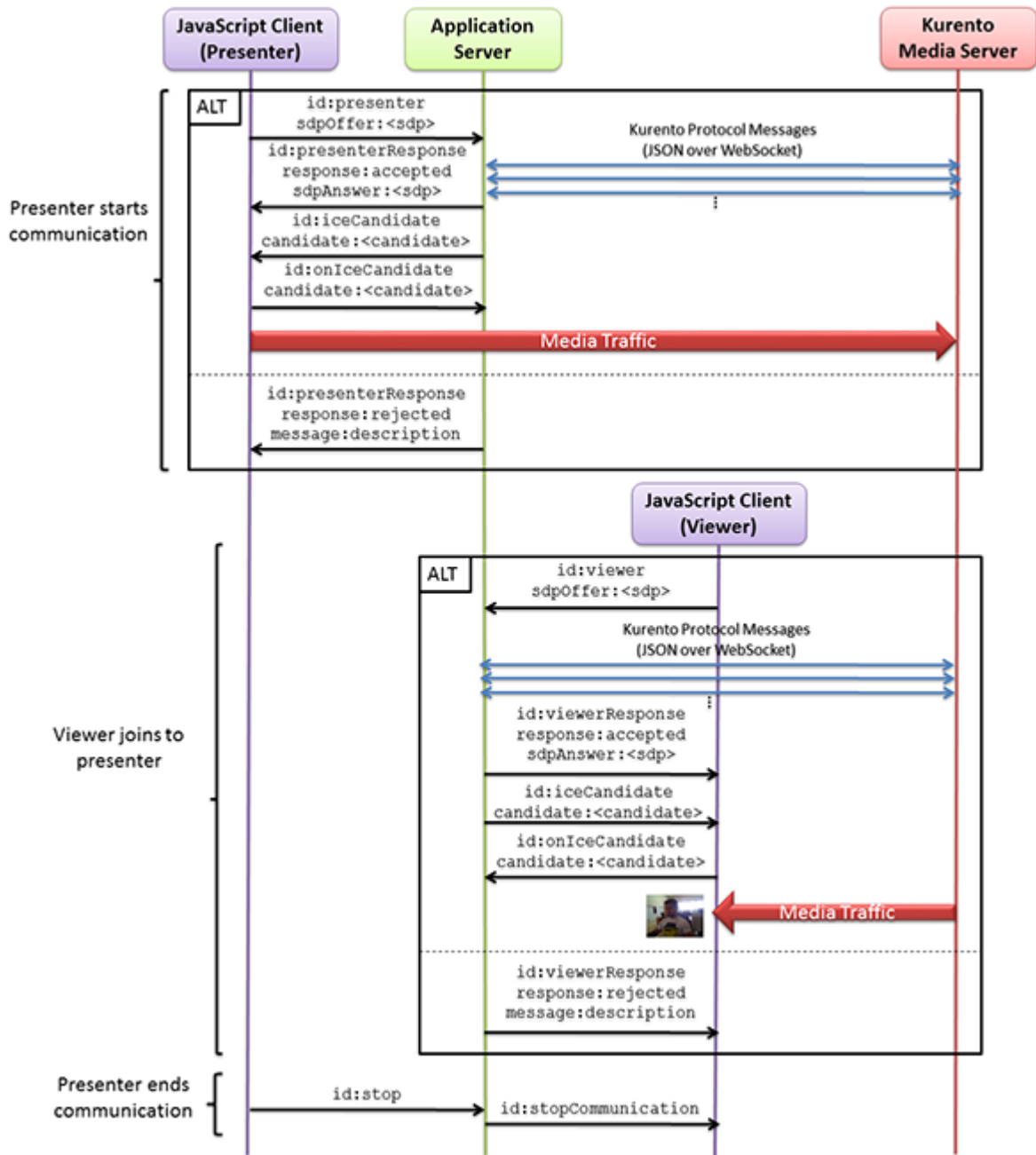


Fig. 23: One to many video call signaling protocol

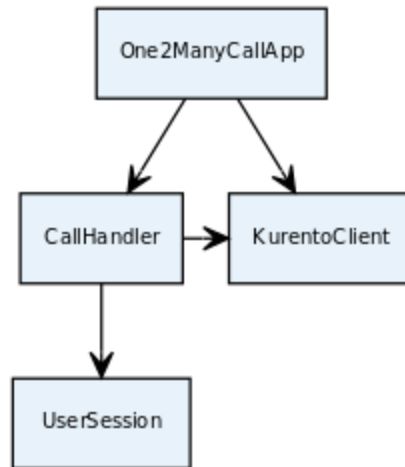


Fig. 24: Server-side class diagram of the One2Many app

(continued from previous page)

```

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

```

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

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

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

```

public class CallHandler extends TextWebSocketHandler {

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

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

    @Autowired
    private KurentoClient kurento;

    private MediaPipeline pipeline;
    private UserSession presenterUserSession;

    @Override

```

(continues on next page)

(continued from previous page)

```

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

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

            UserSession user = null;
            if (presenterUserSession != null) {
                if (presenterUserSession.getSession() == session) {
                    user = presenterUserSession;
                } else {
                    user = viewers.get(session.getId());
                }
            }
            if (user != null) {
                IceCandidate cand = new IceCandidate(candidate.get("candidate").
↳getAsString(),
                candidate.get("sdpMid").getAsString(), candidate.get("sdpMLineIndex").
↳getAsInt());
                user.addCandidate(cand);
            }
            break;
        }
        case "stop":
            stop(session);
            break;
        default:
            break;
        }
    }

    private void handleErrorResponse(Throwable t, WebSocketSession session,
        String responseId) throws IOException {
        stop(session);
        log.error(t.getMessage(), t);
    }

```

(continues on next page)

(continued from previous page)

```

        JsonObject response = new JsonObject();
        response.addProperty("id", responseId);
        response.addProperty("response", "rejected");
        response.addProperty("message", t.getMessage());
        session.sendMessage(new TextMessage(response.toString()));
    }

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

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

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

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

```

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

```

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

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

        WebRtcEndpoint presenterWebRtc = presenterUserSession.getWebRtcEndpoint();

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

            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {

```

(continues on next page)

(continued from previous page)

```

        synchronized (session) {
            session.sendMessage(new TextMessage(response.toString()));
        }
    } catch (IOException e) {
        log.debug(e.getMessage());
    }
}

});

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

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

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

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

```

The viewer method is similar, but not the *Presenter* WebRtcEndpoint is connected to each of the viewers WebRtcEndpoints, otherwise an error is sent back to the client.

```

private synchronized void viewer(final WebSocketSession session, JsonObject jsonMessage)
↪throws IOException {
    if (presenterUserSession == null || presenterUserSession.getWebRtcEndpoint() == null)
↪{
        JsonObject response = new JsonObject();
        response.addProperty("id", "viewerResponse");
        response.addProperty("response", "rejected");
        response.addProperty("message", "No active sender now. Become sender or . Try
↪again later ...");
        session.sendMessage(new TextMessage(response.toString()));
    } else {
        if (viewers.containsKey(session.getId())) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "viewerResponse");
            response.addProperty("response", "rejected");
            response.addProperty("message",
                "You are already viewing in this session. Use a different browser to add
↪additional viewers.");

```

(continues on next page)

(continued from previous page)

```

        session.sendMessage(new TextMessage(response.toString()));
        return;
    }
    UserSession viewer = new UserSession(session);
    viewers.put(session.getId(), viewer);

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

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

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

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

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

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

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

```

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

```

private synchronized void stop(WebsocketSession session) throws IOException {
    String sessionId = session.getId();
    if (presenterUserSession != null && presenterUserSession.getSession().getId().
    ↪ equals(sessionId)) {
        for (UserSession viewer : viewers.values()) {

```

(continues on next page)

(continued from previous page)

```

        JsonObject response = new JsonObject();
        response.addProperty("id", "stopCommunication");
        viewer.sendMessage(response);
    }

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

```

Client-Side

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

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

```

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

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

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

```

(continues on next page)

(continued from previous page)

```

    });
    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);
            });
    }
}

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*:

```
<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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.4.2 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.js 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* in your system. In an Ubuntu machine, you can install it as follows:

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install -y nodejs
```

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.git
cd kurento/tutorials/javascript-node/one2many-call/
git checkout 7.0.0
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 a screenshot of the Presenter's web GUI:

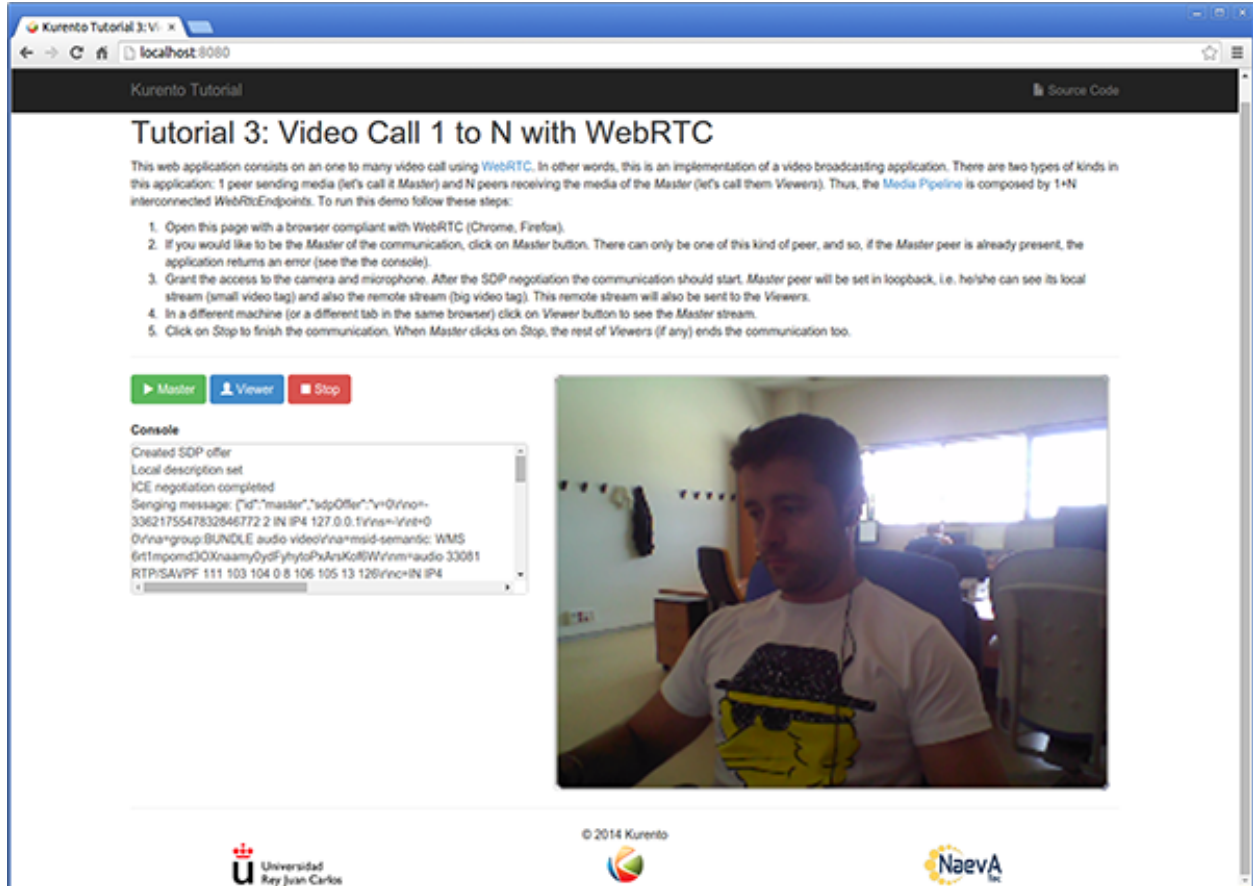


Fig. 25: One to many video call screenshot

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

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.

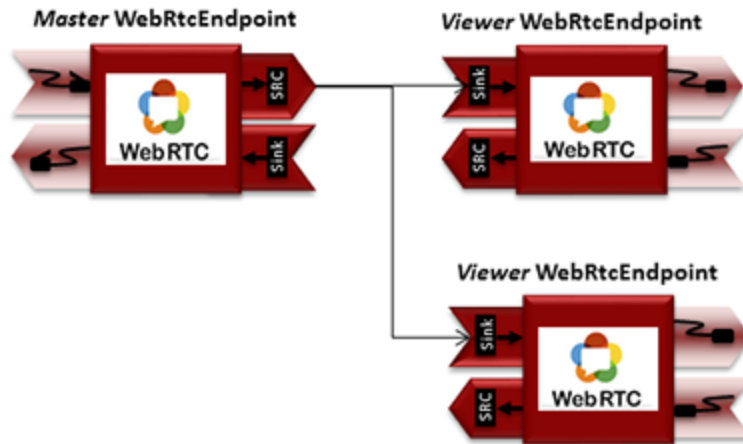


Fig. 26: One to many video call Media Pipeline

4. When the *Presenter* finishes the session each connected *Viewer* receives an *stopCommunication* message and also terminates its session.

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

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

Application Server Logic

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

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

```

var ws = require('ws');

[...]

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

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

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

```

(continues on next page)

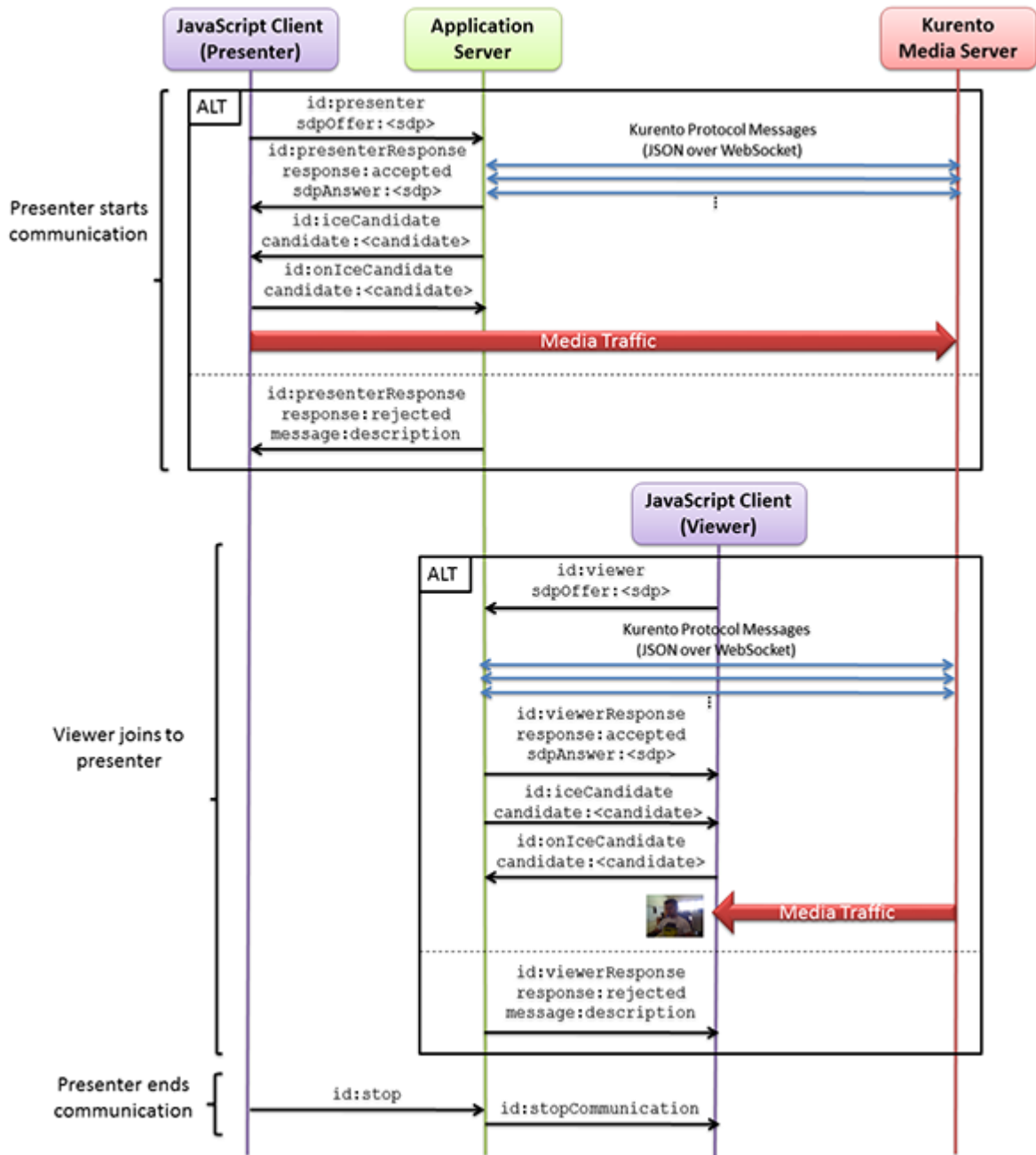


Fig. 27: One to many video call signaling protocol

(continued from previous page)

```

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;

    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);
    }
}

```

(continues on next page)

(continued from previous page)

```

        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* (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);
      }

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

(continues on next page)

(continued from previous page)

```

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

        webRtcEndpoint.on('IceCandidateFound', 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) {

```

(continues on next page)

(continued from previous page)

```

        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('IceCandidateFound', 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);

```

(continues on next page)

(continued from previous page)

```

        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 `IceCandidateFound` 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.

```
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);
    }
}
```

(continues on next page)

(continued from previous page)

```
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.

```
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);
```

(continues on next page)

(continued from previous page)

```
        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" : "7.0.0"
}
```

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" : "7.0.0"
}
```

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

7.5 WebRTC One-To-One video call

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

7.5.1 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.git
cd kurento/tutorials/java/one2one-call/
git checkout 7.0.0
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 a 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:

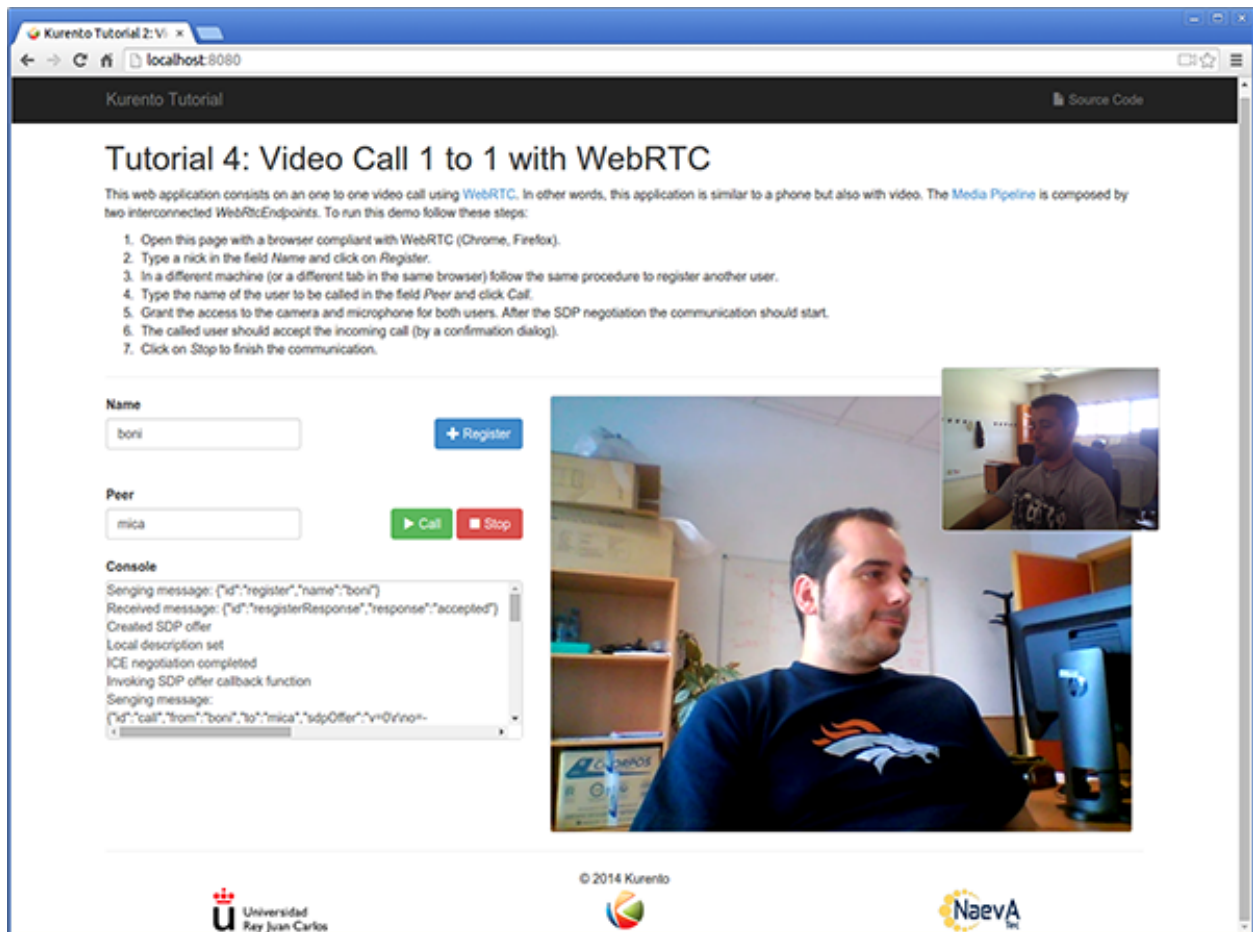


Fig. 28: One to one video call screenshot



Fig. 29: One to one video call Media Pipeline

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

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

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

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

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

Application Server Logic

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

Note: You can use whatever Java server side technology you prefer to build web applications with Kurento. For example, a pure Java EE application, SIP Servlets, Play, 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:

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();
    }

    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
```

(continues on next page)

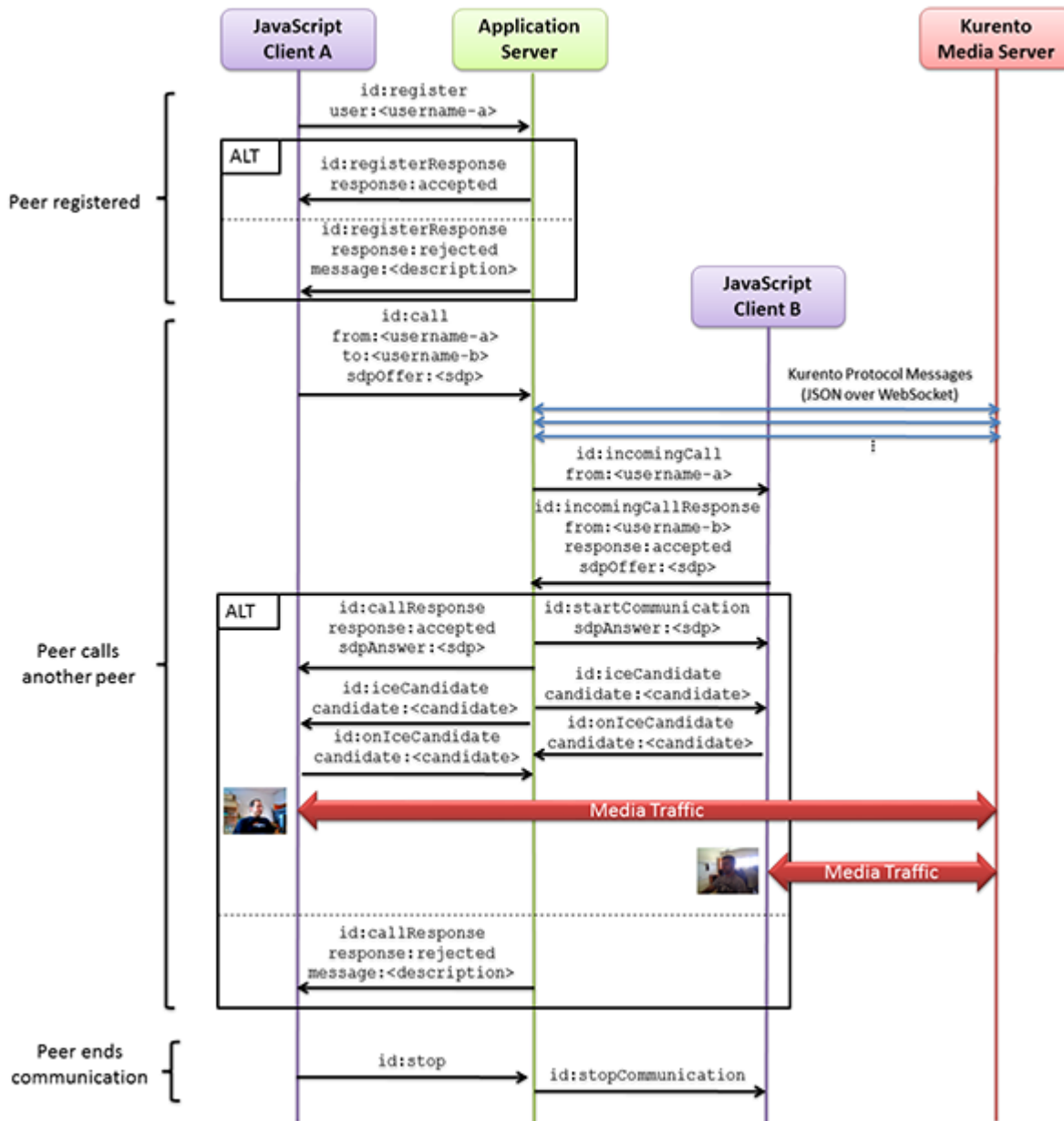


Fig. 30: One to many one call signaling protocol

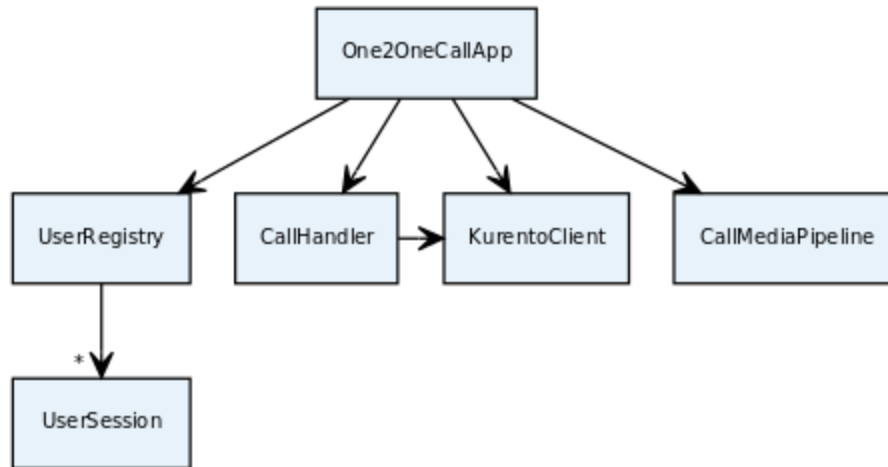


Fig. 31: Server-side class diagram of the one to one video call app

(continued from previous page)

```

    registry.addHandler(callHandler(), "/call");
}

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

```

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

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

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

```

public class CallHandler extends TextWebSocketHandler {

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

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

    @Autowired
    private KurentoClient kurento;

    @Autowired

```

(continues on next page)

(continued from previous page)

```

private UserRegistry registry;

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

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

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

private void handleErrorResponse(Throwable t, WebSocketSession session,

```

(continues on next page)

(continued from previous page)

```

        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").getString();

    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 {

```

(continues on next page)

(continued from previous page)

```

    registry.register(caller);
}

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

```

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

```

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

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

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

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

        caller.sendMessage(response);
    }
}

```

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

```

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

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

```

(continues on next page)

(continued from previous page)

```

        : stopperUser.getCallingTo() != null
          ? registry.getByNames(stopperUser.getCallingTo())
            : null;

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

```

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

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

```

private void incomingCallResponse(final UserSession callee, JsonObject jsonMessage)
    throws IOException {
    String callResponse = jsonMessage.get("callResponse").getAsString();
    String from = jsonMessage.get("from").getAsString();
    final UserSession calleeer = registry.getByNames(from);
    String to = calleeer.getCallingTo();

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

        CallMediaPipeline pipeline = null;
        try {
            pipeline = new CallMediaPipeline(kurento);
            pipelines.put(calleeer.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 {

```

(continues on next page)

(continued from previous page)

```

        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();
callee.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 (callee.getSession()) {
                callee.getSession().sendMessage(new TextMessage(response.
↪toString()));
            }
        } catch (IOException e) {
            log.debug(e.getMessage());
        }
    }
});

String callerSdpAnswer = pipeline.generateSdpAnswerForCaller(callerSdpOffer);

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

synchronized (callee) {
    callee.sendMessage(startCommunication);
}

pipeline.getCalleeWebRtcEP().gatherCandidates();

JsonObject response = new JsonObject();
response.addProperty("id", "callResponse");
response.addProperty("response", "accepted");
response.addProperty("sdpAnswer", callerSdpAnswer);

synchronized (callee) {
    callee.sendMessage(response);
}

```

(continues on next page)

(continued from previous page)

```

        pipeline.getCallerWebRtcEP().gatherCandidates();
    } catch (Throwable t) {
        log.error(t.getMessage(), t);

        if (pipeline != null) {
            pipeline.release();
        }

        pipelines.remove(calleer.getSessionId());
        pipelines.remove(callee.getSessionId());

        JsonObject response = new JsonObject();
        response.addProperty("id", "callResponse");
        response.addProperty("response", "rejected");
        calleer.sendMessage(response);

        response = new JsonObject();
        response.addProperty("id", "stopCommunication");
        callee.sendMessage(response);
    }
} else {
    JsonObject response = new JsonObject();
    response.addProperty("id", "callResponse");
    response.addProperty("response", "rejected");
    calleer.sendMessage(response);
}
}

```

The media logic in this demo is implemented in the class `CallMediaPipeline`. As you can see, the media pipeline of this demo is quite simple: two `WebRtcEndpoint` elements directly interconnected. Please take note that the `WebRtcEndpoints` need to be connected twice, one for each media direction.

```

public class CallMediaPipeline {

    private MediaPipeline pipeline;
    private WebRtcEndpoint callerWebRtcEP;
    private WebRtcEndpoint calleeWebRtcEP;

    public CallMediaPipeline(KurentoClient kurento) {
        try {
            this.pipeline = kurento.createMediaPipeline();
            this.callerWebRtcEP = new WebRtcEndpoint.Builder(pipeline).build();
            this.calleeWebRtcEP = new WebRtcEndpoint.Builder(pipeline).build();

            this.callerWebRtcEP.connect(this.calleeWebRtcEP);
            this.calleeWebRtcEP.connect(this.callerWebRtcEP);
        } catch (Throwable t) {
            if (this.pipeline != null) {
                pipeline.release();
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

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: `registerResponse`, `callResponse`, `incomingCall`, `iceCandidate` and `startCommunication`. Convenient actions are taken to implement each step in the communication. For example, in functions `call` and `incomingCall` (for caller and callee respectively), the function `WebRtcPeer.Sendrecv` of *kurento-utils.js* is used to start a WebRTC communication.

```

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

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

```

(continues on next page)

(continued from previous page)

```

console.info('Received message: ' + message.data);

switch (parsedMessage.id) {
case 'registerResponse':
    registerResponse(parsedMessage);
    break;
case 'callResponse':
    callResponse(parsedMessage);
    break;
case 'incomingCall':
    incomingCall(parsedMessage);
    break;
case 'startCommunication':
    startCommunication(parsedMessage);
    break;
case 'stopCommunication':
    console.info("Communication ended by remote peer");
    stop(true);
    break;
case 'iceCandidate':
    webRtcPeer.addIceCandidate(parsedMessage.candidate, function (error) {
        if (!error) return;
        console.error("Error adding candidate: " + error);
    });
    break;
default:
    console.error('Unrecognized message', parsedMessage);
}
}

function incomingCall(message) {
    //If busy 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);

        from = message.from;
        var options = {
            localVideo: videoInput,
            remoteVideo: videoOutput,
            onIceCandidate: onIceCandidate,

```

(continues on next page)

(continued from previous page)

```

        onerror: onError
    }
    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
        function (error) {
            if(error) {
                return console.error(error);
            }
            webRtcPeer.generateOffer (onOfferIncomingCall);
        });

    } else {
        var response = {
            id : 'incomingCallResponse',
            from : message.from,
            callResponse : 'reject',
            message : 'user declined'
        };
        sendMessage(response);
        stop();
    }
}

function call() {
    if (document.getElementById('peer').value == '') {
        window.alert("You must specify the peer name");
        return;
    }
    setCallState(PROCESSING_CALL);
    showSpinner(videoInput, videoOutput);

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

```

Dependencies

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

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

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

7.5.2 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.js 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* in your system. In an Ubuntu machine, you can install it as follows:

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install -y nodejs
```

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.git
cd kurento/tutorials/javascript-node/one2one-call/
git checkout 7.0.0
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 a screenshot of this demo running in a web browser:

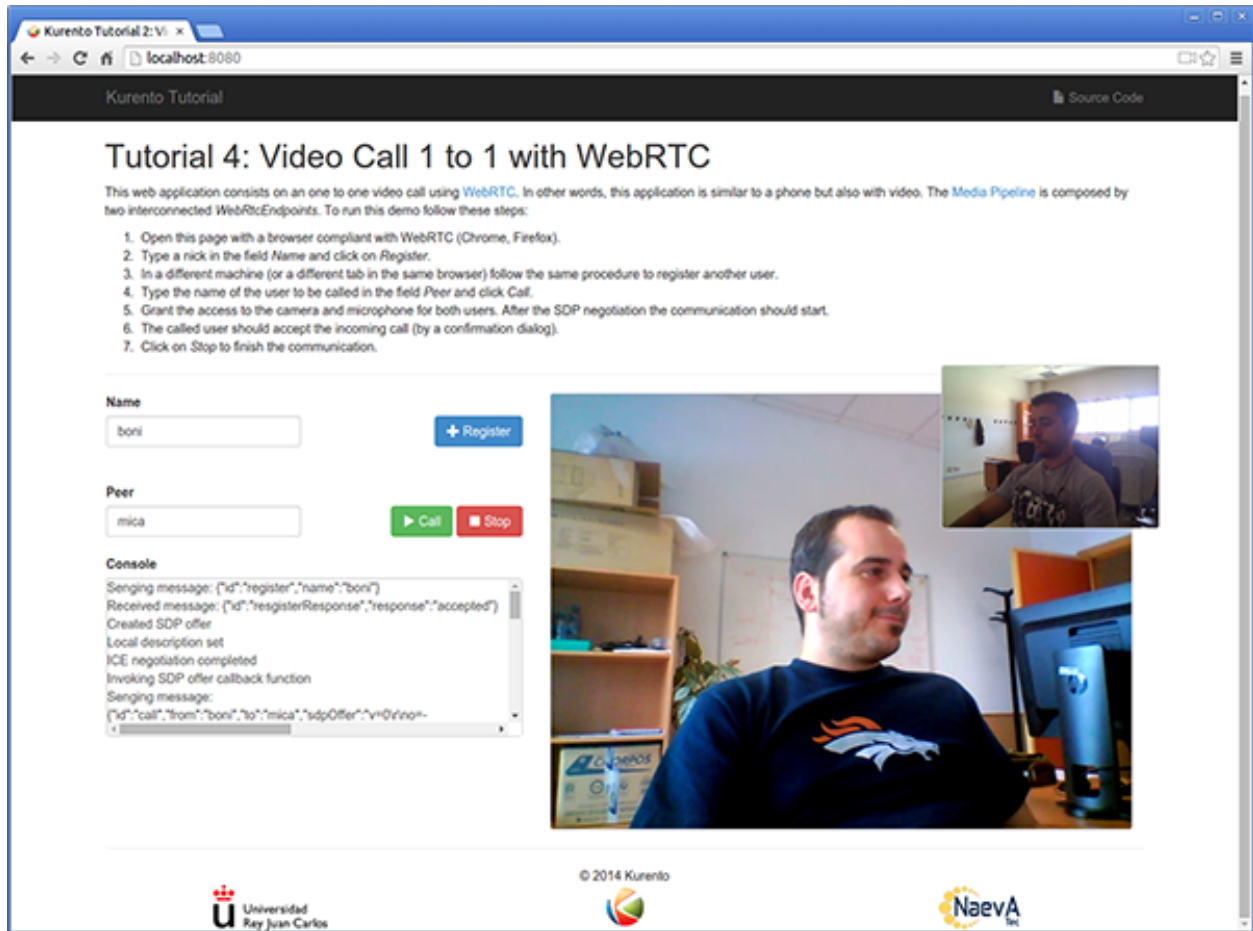


Fig. 32: One to one video call screenshot

The interface of the application (an HTML web page) is composed by two HTML5 video tags: one for the local stream and other for the remote peer stream). If two users, A and B, are using the application, the media flow goes this way: The video camera stream of user A is sent to the Kurento Media Server, which sends it to user B. In the same way, B sends to Kurento Media Server, which forwards it to A. This means that KMS is providing a B2B (back-to-back) call service.

To implement this behavior create a *Media Pipeline* composed by two WebRtC endpoints connected in B2B. The implemented media pipeline is illustrated in the following picture:

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



Fig. 33: One to one video call Media Pipeline

6. One of the users finishes the video communication

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

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

Application Server Logic

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

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

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

[...]
```

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

wss.on('connection', function(ws) {
  var sessionId = nextUniqueId();
  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) {
```

(continues on next page)

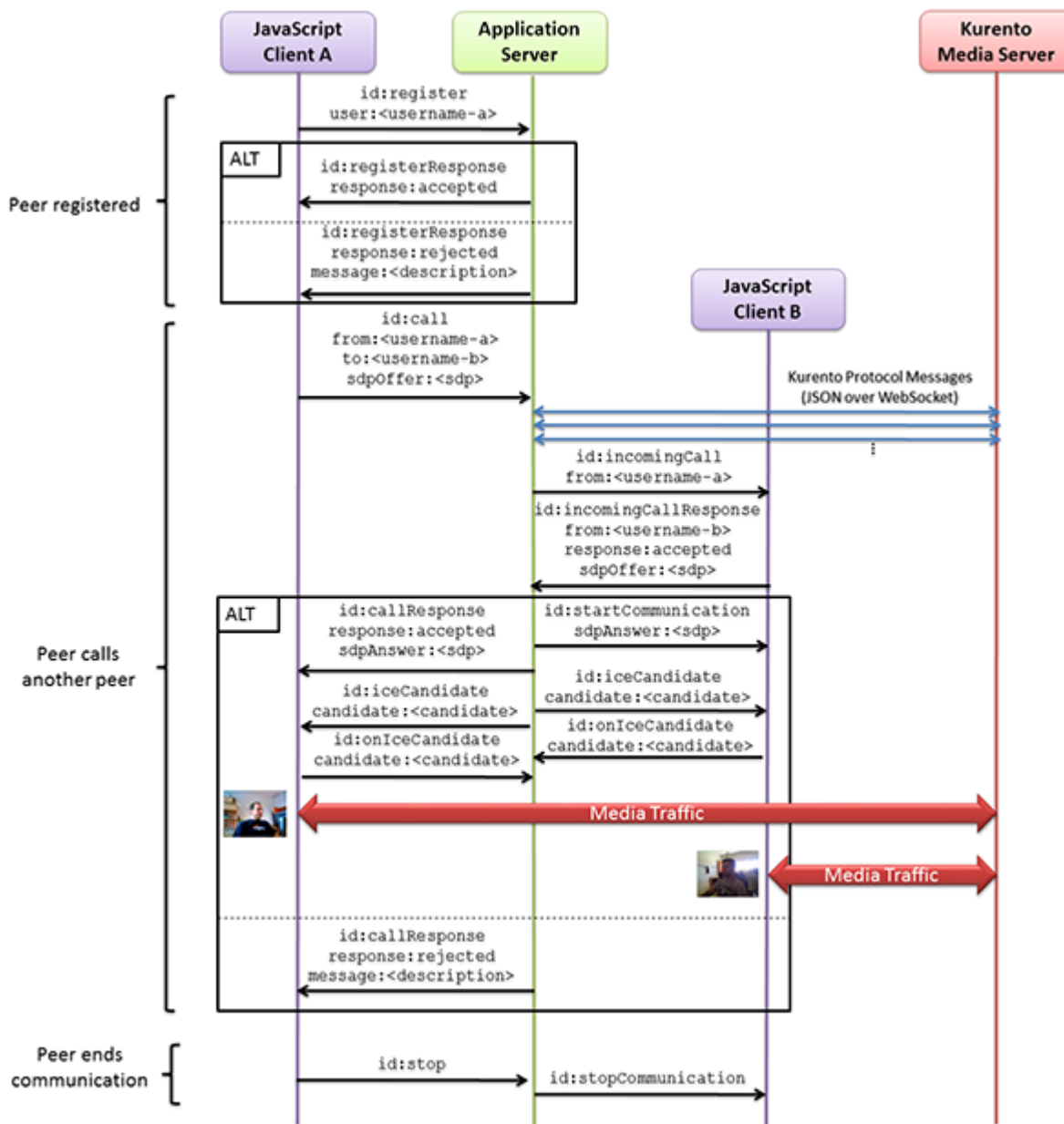


Fig. 34: One to many one call signaling protocol

(continued from previous page)

```

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

switch (message.id) {
case 'register':
    register(sessionId, message.name, ws);
    break;

case 'call':
    call(sessionId, message.to, message.from, message.sdpOffer);
    break;

case 'incomingCallResponse':
    incomingCallResponse(sessionId, message.from, message.callResponse, message.
↪sdpOffer, ws);
    break;

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

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

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

});
});

```

In order to perform a call, each user (the caller and the callee) must be register in the system. For this reason, in the server-side there is a class named `UserRegistry` to store and locate users. Then, the `register` message fires the execution of the following function:

```

// Represents registrar of users
function UserRegistry() {
    this.usersById = {};
    this.usersByName = {};
}

UserRegistry.prototype.register = function(user) {
    this.usersById[user.id] = user;
    this.usersByName[user.name] = user;
}

UserRegistry.prototype.unregister = function(id) {

```

(continues on next page)

(continued from previous page)

```

    var user = this.getById(id);
    if (user) delete this.usersById[id]
    if (user && this.getByName(user.name)) delete this.usersByName[user.name];
}

UserRegistry.prototype.getById = function(id) {
    return this.usersById[id];
}

UserRegistry.prototype.getByName = function(name) {
    return this.usersByName[name];
}

UserRegistry.prototype.removeById = function(id) {
    var userSession = this.usersById[id];
    if (!userSession) return;
    delete this.usersById[id];
    delete this.usersByName[userSession.name];
}

function register(id, name, ws, callback) {
    function onError(error) {
        ws.send(JSON.stringify({id:'registerResponse', response : 'rejected ', message:↵
↵error})));
    }

    if (!name) {
        return onError("empty user name");
    }

    if (userRegistry.getByName(name)) {
        return onError("User " + name + " is already registered");
    }

    userRegistry.register(new UserSession(id, name, ws));
    try {
        ws.send(JSON.stringify({id: 'registerResponse', response: 'accepted'}));
    } catch(exception) {
        onError(exception);
    }
}

```

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

```

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

var kurentoClient = null;

var argv = minimist(process.argv.slice(2), {
    default: {

```

(continues on next page)

(continued from previous page)

```

        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.getId(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 = {

```

(continues on next page)

(continued from previous page)

```

        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.getName(from)) {
        return onError(null, 'unknown from = ' + from);
    }
    var caller = userRegistry.getName(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) {

```

(continues on next page)

(continued from previous page)

```

        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 `IceCandidateFound` 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.getId(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);
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

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':
            registerResponse(parsedMessage);
            break;
        case 'callResponse':
            callResponse(parsedMessage);
            break;
        case 'incomingCall':
            incomingCall(parsedMessage);
            break;
        case 'startCommunication':
            startCommunication(parsedMessage);
            break;
        case 'stopCommunication':
            console.info("Communication ended by remote peer");
            stop(true);
            break;
        case 'iceCandidate':
            webRtcPeer.addIceCandidate(parsedMessage.candidate);
            break;
    }
}

```

(continues on next page)

(continued from previous page)

```

    default:
        console.error('Unrecognized message', parsedMessage);
    }
}

```

On the one hand, the function call is executed in the caller client-side, using the method `WebRtcPeer.WebRtcPeerSendrecv` of *kurento-utils.js* to start a WebRTC communication in duplex mode. On the other hand, the function `incomingCall` in the callee client-side uses also the method `WebRtcPeer.WebRtcPeerSendrecv` of *kurento-utils.js* to complete the WebRTC call.

```

function call() {
    if (document.getElementById('peer').value == '') {
        window.alert("You must specify the peer name");
        return;
    }

    setCallState(PROCESSING_CALL);

    showSpinner(videoInput, videoOutput);

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

    webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(
        error) {
        if (error) {
            console.error(error);
            setCallState(NO_CALL);
        }

        this.generateOffer(function(error, offerSdp) {
            if (error) {
                console.error(error);
                setCallState(NO_CALL);
            }
            var message = {
                id : 'call',
                from : document.getElementById('name').value,
                to : document.getElementById('peer').value,
                sdpOffer : offerSdp
            };
            sendMessage(message);
        });
    });
}

function incomingCall(message) {
    // If busy just reject without disturbing user
    if (callState != NO_CALL) {

```

(continues on next page)

(continued from previous page)

```

    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);
                }
                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);
}

```

(continues on next page)

(continued from previous page)

```
}  
}
```

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" : "7.0.0"  
}
```

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" : "7.0.0"  
}
```

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

7.6 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.

7.6.1 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.git
cd kurento/tutorials/java/one2one-call-advanced/
git checkout 7.0.0
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 a 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:

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

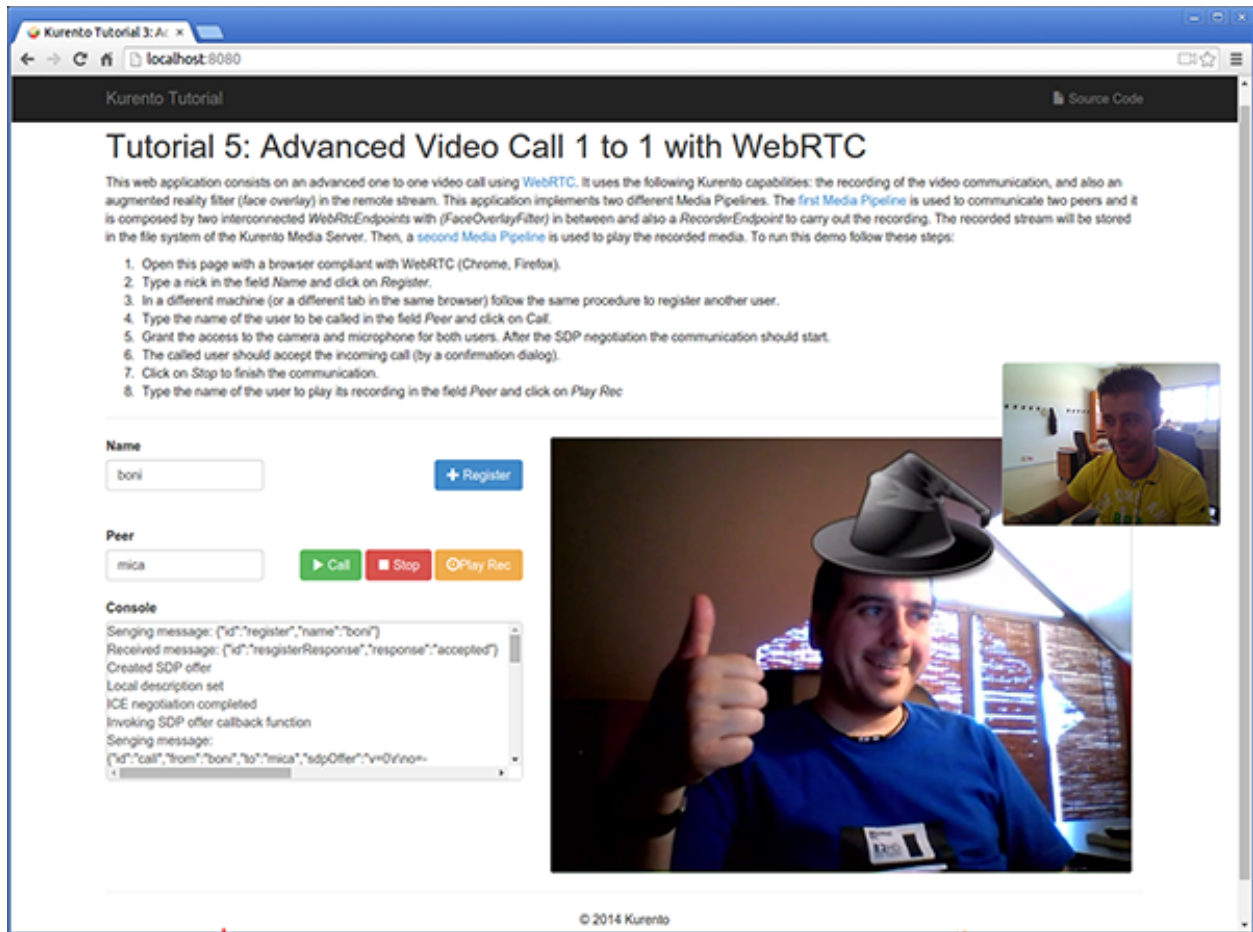


Fig. 35: Advanced one to one video call screenshot

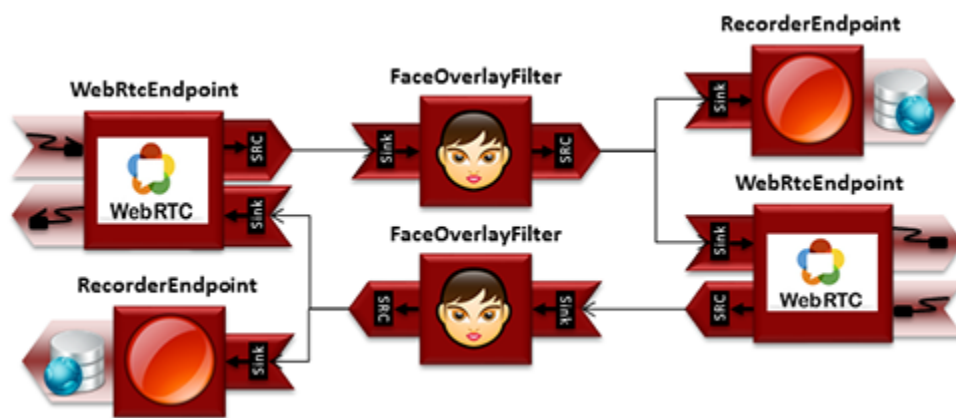


Fig. 36: Advanced one to one video call media pipeline (1)

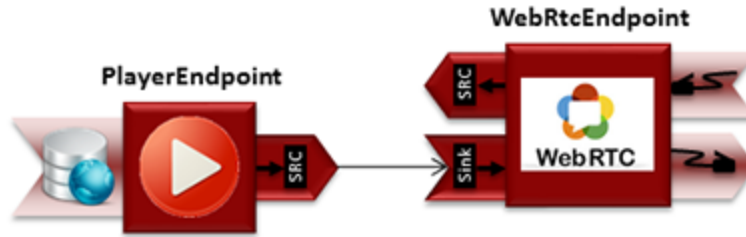


Fig. 37: Advanced one to one video call media pipeline (2)

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:

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

```
@EnableWebSocket
@SpringBootApplication
public class One2OneCallAdvApp implements WebSocketConfigurer {

    final static String DEFAULT_APP_SERVER_URL = "https://localhost:8443";

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

(continues on next page)

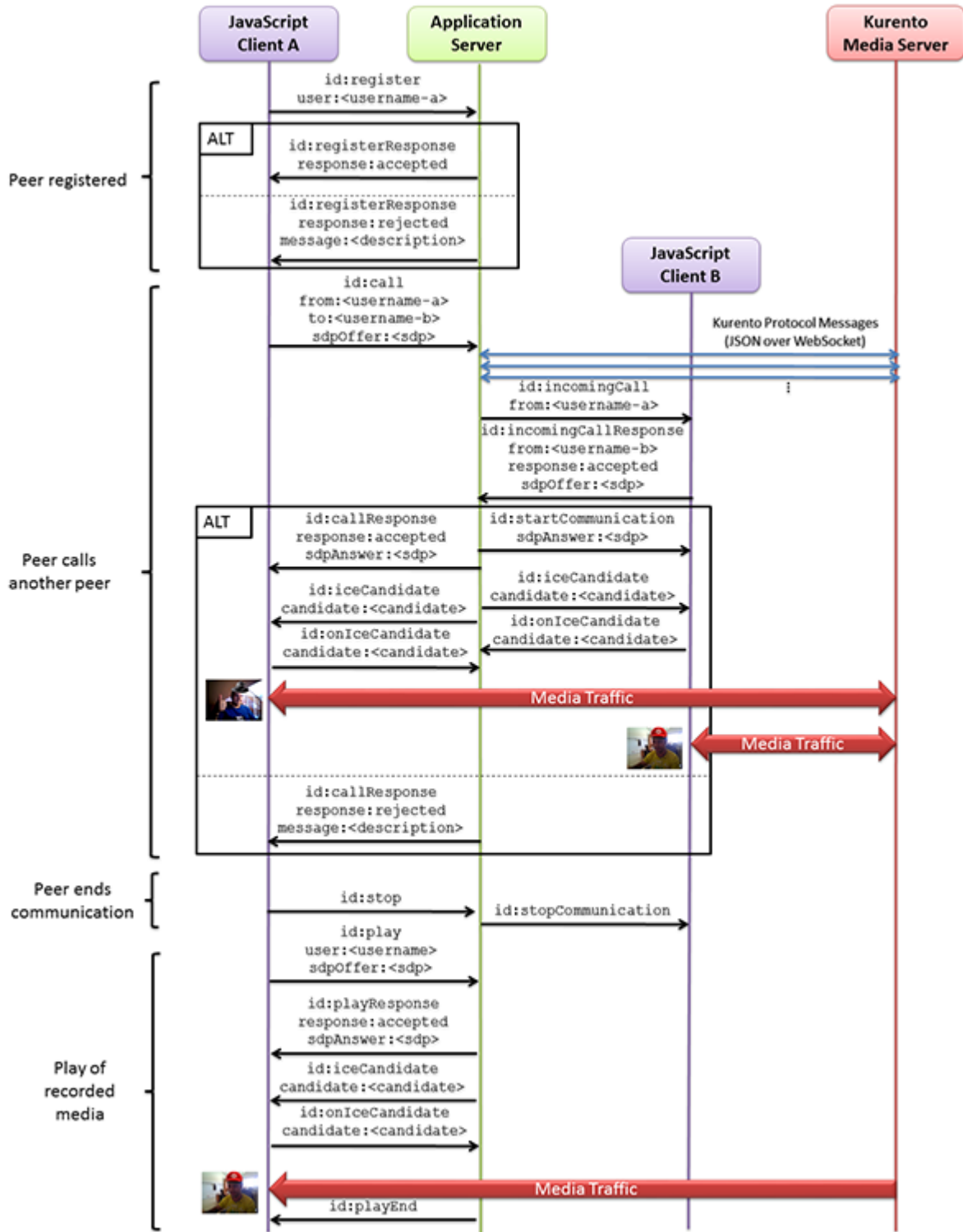


Fig. 38: Advanced one to one video call signaling protocol

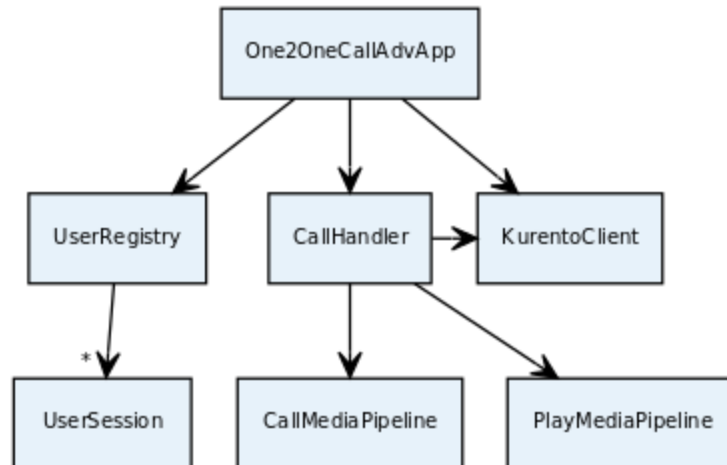


Fig. 39: Server-side class diagram of the advanced one to one video call app

(continued from previous page)

```

}

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

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

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

public static void main(String[] args) throws Exception {
    new SpringApplication(One2OneCallAdvApp.class).run(args);
}
}

```

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

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

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

```

public class CallHandler extends TextWebSocketHandler {

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

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

    @Autowired
    private KurentoClient kurento;

    @Autowired
    private UserRegistry registry;

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

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

        switch (jsonMessage.get("id").getAsString()) {
            case "register":
                register(session, jsonMessage);
                break;
            case "call":
                call(user, jsonMessage);
                break;
            case "incomingCallResponse":
                incomingCallResponse(user, jsonMessage);
                break;
            case "play":
                play(user, jsonMessage);
                break;
            case "onIceCandidate": {
                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;
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
    case "stop":
        stop(session);
        releasePipeline(user);
    case "stopPlay":
        releasePipeline(user);
    default:
        break;
    }
}

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

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

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

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

public void releasePipeline(UserSession session) throws IOException {
    ...
}

private void play(final UserSession session, JsonObject jsonMessage)
    throws IOException {
    ...
}

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

```

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

```
private void register(WebSocketSession session, JsonObject jsonMessage)
    throws IOException {
    String name = jsonMessage.getAsJsonPrimitive("name").getString();

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

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

In the call method, the server checks if there are a registered user with the name specified in to message attribute and send an incomingCall message to it. Or, if there isn't any user with that name, a callResponse message is sent to caller rejecting the call.

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

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

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

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

        caller.sendMessage(response);
    }
}
```

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

browsers.

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

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

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

        CallMediaPipeline callMediaPipeline = new CallMediaPipeline(
            kurento, from, to);
        pipelines.put(caller.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");
```

(continues on next page)

(continued from previous page)

```

startCommunication.addProperty("sdpAnswer", calleeSdpAnswer);

synchronized (callee) {
    callee.sendMessage(startCommunication);
}

callMediaPipeline.getCallerWebRtcEP().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");

```

(continues on next page)

(continued from previous page)

```

        response.addProperty("response", "rejected");
        caller.sendMessage(response);
    }
}

```

Finally, the play method instantiates a PlayMediaPipeline object, which is used to create Media Pipeline in charge of the playback of the recorded streams in the Kurento Media Server.

```

private void play(final UserSession session, JsonObject jsonMessage)
    throws IOException {
    String user = jsonMessage.get("user").getAsString();
    log.debug("Playing recorded call of user '{}'", user);

    JsonObject response = new JsonObject();
    response.addProperty("id", "playResponse");

    if (registry.getByName(user) != null
        && registry.getBySession(session.getSession()) != null) {
        final PlayMediaPipeline playMediaPipeline = new PlayMediaPipeline(
            kurento, user, session.getSession());
        String sdpOffer = jsonMessage.get("sdpOffer").getAsString();

        session.setPlayingWebRtcEndpoint(playMediaPipeline.getWebRtc());

        playMediaPipeline.getPlayer().addEndOfStreamListener(
            new EventListener<EndOfStreamEvent>() {
                @Override
                public void onEvent(EndOfStreamEvent event) {
                    UserSession user = registry
                        .getBySession(session.getSession());
                    releasePipeline(user);
                    playMediaPipeline.sendPlayEnd(session.getSession());
                }
            });

        playMediaPipeline.getWebRtc().addIceCandidateFoundListener(
            new EventListener<IceCandidateFoundEvent>() {

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

```

(continues on next page)

(continued from previous page)

```

        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) {

```

(continues on next page)

(continued from previous page)

```

// Media pipeline
pipeline = kurento.createMediaPipeline();

// Media Elements (WebRtcEndpoint, RecorderEndpoint, FaceOverlayFilter)
webRtcCaller = new WebRtcEndpoint.Builder(pipeline).build();
webRtcCallee = new WebRtcEndpoint.Builder(pipeline).build();

recorderCaller = new RecorderEndpoint.Builder(pipeline, RECORDING_PATH
    + from + RECORDING_EXT).build();
recorderCallee = new RecorderEndpoint.Builder(pipeline, RECORDING_PATH
    + to + RECORDING_EXT).build();

String appServerUrl = System.getProperty("app.server.url",
    One2OneCallAdvApp.DEFAULT_APP_SERVER_URL);
FaceOverlayFilter faceOverlayFilterCaller = new FaceOverlayFilter.Builder(
    pipeline).build();
faceOverlayFilterCaller.setOverlaidImage(appServerUrl
    + "/img/mario-wings.png", -0.35F, -1.2F, 1.6F, 1.6F);

FaceOverlayFilter faceOverlayFilterCallee = new FaceOverlayFilter.Builder(
    pipeline).build();
faceOverlayFilterCallee.setOverlaidImage(
    appServerUrl + "/img/Hat.png", -0.2F, -1.35F, 1.5F, 1.5F);

// Connections
webRtcCaller.connect(faceOverlayFilterCaller);
faceOverlayFilterCaller.connect(webRtcCallee);
faceOverlayFilterCaller.connect(recorderCaller);

webRtcCallee.connect(faceOverlayFilterCallee);
faceOverlayFilterCallee.connect(webRtcCaller);
faceOverlayFilterCallee.connect(recorderCallee);
}

public void record() {
    recorderCaller.record();
    recorderCallee.record();
}

public String generateSdpAnswerForCaller(String sdpOffer) {
    return webRtcCaller.processOffer(sdpOffer);
}

public String generateSdpAnswerForCallee(String sdpOffer) {
    return webRtcCallee.processOffer(sdpOffer);
}

public MediaPipeline getPipeline() {
    return pipeline;
}

public WebRtcEndpoint getCallerWebRtcEP() {

```

(continues on next page)

(continued from previous page)

```

    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.

```

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);
            }
        });
    }
}

```

(continues on next page)

(continued from previous page)

```

public void sendPlayEnd(WebSocketSession session) {
    try {
        JsonObject response = new JsonObject();
        response.addProperty("id", "playEnd");
        session.sendMessage(new TextMessage(response.toString()));
    } catch (IOException e) {
        log.error("Error sending playEndOfStream message", e);
    }
}

public void play() {
    player.play();
}

public String generateSdpAnswer(String sdpOffer) {
    return webRtc.processOffer(sdpOffer);
}

public MediaPipeline getPipeline() {
    return pipeline;
}

public WebRtcEndpoint getWebRtc() {
    return webRtc;
}

public PlayerEndpoint getPlayer() {
    return player;
}
}

```

Client-Side

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

These libraries are linked in the [index.html](#) web page, and are used in the [index.js](#).

In the following snippet we can see the creation of the `WebSocket` (variable `ws`) in the path `/call`. Then, the `onmessage` listener of the `WebSocket` is used to implement the JSON signaling protocol in the client-side. Notice that there are six incoming messages to client: `resgisterResponse`, `callResponse`, `incomingCall`, `startCommunication`, `iceCandidate` and `play`. Convenient actions are taken to implement each step in the communication. On the one hand, in functions `call` and `incomingCall` (for caller and callee respectively), the function `WebRtcPeer.WebRtcPeerSendrecv` of *kurento-utils.js* is used to start a WebRTC communication. On the other hand in the function `play`, the function `WebRtcPeer.WebRtcPeerRecvonly` is called since the `WebRtcEndpoint` is used in receive-only.

```

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

ws.onmessage = function(message) {

```

(continues on next page)

(continued from previous page)

```

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

switch (parsedMessage.id) {
case 'registerResponse':
    registerResponse(parsedMessage);
    break;
case 'callResponse':
    callResponse(parsedMessage);
    break;
case 'incomingCall':
    incomingCall(parsedMessage);
    break;
case 'startCommunication':
    startCommunication(parsedMessage);
    break;
case 'stopCommunication':
    console.info("Communication ended by remote peer");
    stop(true);
    break;
case 'playResponse':
    playResponse(parsedMessage);
    break;
case 'playEnd':
    playEnd();
    break;
case 'iceCandidate':
    webRtcPeer.addIceCandidate(parsedMessage.candidate, function (error) {
        if (!error) return;
        console.error("Error adding candidate: " + error);
    });
    break;
default:
    console.error('Unrecognized message', parsedMessage);
}
}

function incomingCall(message) {
    // If busy 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?')) {

```

(continues on next page)

(continued from previous page)

```

showSpinner(videoInput, videoOutput);

from = message.from;
var options = {
    localVideo: videoInput,
    remoteVideo: videoOutput,
    onIceCandidate: onIceCandidate
}
webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,
    function (error) {
        if(error) {
            return console.error(error);
        }
        this.generateOffer (onOfferIncomingCall);
    });
} else {
    var response = {
        id : 'incomingCallResponse',
        from : message.from,
        callResponse : 'reject',
        message : 'user declined'
    };
    sendMessage(response);
    stop();
}
}

function call() {
    if (document.getElementById('peer').value == '') {
        document.getElementById('peer').focus();
        window.alert("You must specify the peer name");
        return;
    }
    setCallState(DISABLED);
    showSpinner(videoInput, videoOutput);

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

function play() {
    var peer = document.getElementById('peer').value;

```

(continues on next page)

(continued from previous page)

```

    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*:

```
<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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.7 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.

7.7.1 Java - Group Call

This tutorial shows how to work with the concept of rooms, allowing to connect several clients between them using *WebRTC* technology, creating a multiconference.

Note: 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.git
cd kurento/tutorials/java/group-call/
git checkout 7.0.0
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 `GroupCallApp`. As you can see, the *KurentoClient* is instantiated in this class as a Spring Bean. This bean is used to create **Kurento Media Pipelines**, which are used to add media capabilities to the application. In this instantiation we see that we need to specify to the client library the location of the Kurento Media Server. In this example, we assume it is located at *localhost* listening in port 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.

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

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

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

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

    @Bean
```

(continues on next page)

(continued from previous page)

```

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 four different kind of incoming messages to the application server: *joinRoom*, *receiveVideoFrom*, *leaveRoom* and *onIceCandidate*. These messages are treated in the *switch* clause, taking the proper steps in each case.

```

public class CallHandler extends TextWebSocketHandler {

    private static final Logger log = LoggerFactory.getLogger(CallHandler.class);

    private static final Gson gson = new GsonBuilder().create();

    @Autowired
    private RoomManager roomManager;

    @Autowired
    private UserRegistry registry;

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

        final UserSession user = registry.getBySession(session);

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

        switch (jsonMessage.get("id").getAsString()) {

```

(continues on next page)

(continued from previous page)

```

    case "joinRoom":
        joinRoom(jsonMessage, session);
        break;
    case "receiveVideoFrom":
        final String senderName = jsonMessage.get("sender").getAsString();
        final UserSession sender = registry.getByName(senderName);
        final String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
        user.receiveVideoFrom(sender, sdpOffer);
        break;
    case "leaveRoom":
        leaveRoom(user);
        break;
    case "onIceCandidate":
        JsonObject candidate = jsonMessage.get("candidate").getAsJsonObject();

        if (user != null) {
            IceCandidate cand = new IceCandidate(candidate.get("candidate").getAsString(),
                candidate.get("sdpMid").getAsString(), candidate.get("sdpMLineIndex").
↪getAsInt());
            user.addCandidate(cand, jsonMessage.get("name").getAsString());
        }
        break;
    default:
        break;
}
}

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws ↪
↪Exception {
    ...
}

private void joinRoom(JsonObject params, WebSocketSession session) throws IOException {
    ...
}

private void leaveRoom(UserSession user) throws IOException {
    ...
}
}

```

In the following snippet, we can see the `afterConnectionClosed` method. Basically, it removes the `UserSession` from `registry` and throws out the user from the room.

```

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws ↪
↪Exception {
    UserSession user = registry.removeBySession(session);
    roomManager.getRoom(user.getRoomName()).leave(user);
}

```

In the `joinRoom` method, the server checks if there are a registered room with the name specified, add the user into

this room and registries the user.

```
private void joinRoom(JsonObject params, WebSocketSession session) throws IOException {
    final String roomName = params.get("room").getAsString();
    final String name = params.get("name").getAsString();
    log.info("PARTICIPANT {}: trying to join room {}", name, roomName);

    Room room = roomManager.getRoom(roomName);
    final UserSession user = room.join(name, session);
    registry.register(user);
}
```

The leaveRoom method finish the video call from one user.

```
private void leaveRoom(UserSession user) throws IOException {
    final Room room = roomManager.getRoom(user.getRoomName());
    room.leave(user);
    if (room.getParticipants().isEmpty()) {
        roomManager.removeRoom(room);
    }
}
```

Client-Side Logic

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

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

```
var ws = new WebSocket('wss://' + location.host + '/groupcall');
var participants = {};
var name;

window.onbeforeunload = function() {
    ws.close();
};

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

    switch (parsedMessage.id) {
        case 'existingParticipants':
            onExistingParticipants(parsedMessage);
            break;
    }
}
```

(continues on next page)

(continued from previous page)

```

    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');
    }
}

```

(continues on next page)

(continued from previous page)

```

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

function onExistingParticipants(msg) {
  var constraints = {
    audio : true,
    video : {
      mandatory : {
        maxWidth : 320,
        maxFrameRate : 15,
        minFrameRate : 15
      }
    }
  };
  console.log(name + " registered in room " + room);
  var participant = new Participant(name);
  participants[name] = participant;
  var video = participant.getVideoElement();

  var options = {
    localVideo: video,
    mediaConstraints: constraints,
    onIceCandidate: participant.onIceCandidate.bind(participant)
  };
  participant.rtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendonly(options,
    function (error) {
      if(error) {
        return console.error(error);
      }
      this.generateOffer (participant.offerToReceiveVideo.bind(participant));
    });
  msg.data.forEach(receiveVideo);
}

function leaveRoom() {
  sendMessage({
    id : 'leaveRoom'
  });

  for ( var key in participants) {
    participants[key].dispose();
  }

  document.getElementById('join').style.display = 'block';
  document.getElementById('room').style.display = 'none';
}

```

(continues on next page)

(continued from previous page)

```

    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*:

```

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

```

(continues on next page)

(continued from previous page)

```
<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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.8 Media Elements metadata

This tutorial detects and draws faces present in the webcam video. It connects filters: `KmsDetectFaces` and the `KmsShowFaces`.

7.8.1 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.git
cd kurento/tutorials/java/facedetector/
git checkout 7.0.0
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 `kurento-module-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 kurento-module-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.

```
@EnableWebSocket
@SpringBootApplication
public class MetadataApp implements WebSocketConfigurer {

    static final String DEFAULT_APP_SERVER_URL = "https://localhost:8443";

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

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

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

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

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

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

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

```

public class MetadataHandler extends TextWebSocketHandler {

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

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

    @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) {

```

(continues on next page)

(continued from previous page)

```

    ...
}
}

```

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

```

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

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

        // Media logic
        KmsShowFaces showFaces = new KmsShowFaces.Builder(pipeline).build();
        KmsDetectFaces detectFaces = new KmsDetectFaces.Builder(pipeline).build();

        webRtcEndpoint.connect(detectFaces);
        detectFaces.connect(showFaces);
        showFaces.connect(webRtcEndpoint);

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

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

```

(continues on next page)

(continued from previous page)

```

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

    webRtcEndpoint.gatherCandidates();

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

```

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

```

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

```

Client-Side Logic

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

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

```

var ws = new WebSocket('wss://' + location.host + '/metadata');

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

    switch (parsedMessage.id) {
        case 'startResponse':
            startResponse(parsedMessage);
            break;
        case 'error':
            if (state == I_AM_STARTING) {
                setState(I_CAN_START);
            }
    }
}

```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

```
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*:

```
<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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.9 WebRTC Media Player

This tutorial reads a file from disk or from any URL, and plays the video to WebRTC.

7.9.1 Java - Player

This tutorial reads a file from disk or from any URL, and plays the video to WebRTC. 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.git
cd kurento/tutorials/java/player/
git checkout 7.0.0
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 **PlayerEndpoint** plays a video and **WebRtcEndpoint** shows it.

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

```
@EnableWebSocket
@SpringBootApplication
public class PlayerApp implements WebSocketConfigurer {

    private static final String KMS_WS_URI_PROP = "kms.url";
    private static final String KMS_WS_URI_DEFAULT = "ws://localhost:8888/kurento";

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

    @Bean
    public KurentoClient kurentoClient() {
        return KurentoClient.create(System.getProperty(KMS_WS_URI_PROP, KMS_WS_URI_DEFAULT));
    }

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

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

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

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

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

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

```
public class PlayerHandler extends TextWebSocketHandler {

    @Autowired
    private KurentoClient kurento;

    private final Logger log = LoggerFactory.getLogger(PlayerHandler.class);
    private final Gson gson = new GsonBuilder().create();
    private final ConcurrentHashMap<String, PlayerMediaPipeline> pipelines =
        new ConcurrentHashMap<>();

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) throws
↳Exception {
        JsonObject jsonMessage = gson.fromJson(message.getPayload(), JsonObject.class);
        String sessionId = session.getId();
        log.debug("Incoming message {} from sessionId", jsonMessage, sessionId);

        try {
            switch (jsonMessage.get("id").getAsString()) {
                case "start":
                    start(session, jsonMessage);
                    break;
                case "stop":
                    stop(sessionId);
                    break;
                case "pause":
                    pause(sessionId);
                    break;
                case "resume":
                    resume(session);
                    break;
                case "doSeek":
                    doSeek(session, jsonMessage);
                    break;
                case "getPosition":
                    getPosition(session);
                    break;
                case "onIceCandidate":
                    onIceCandidate(sessionId, jsonMessage);
                    break;
                default:
                    sendError(session, "Invalid message with id " + jsonMessage.get("id").
↳getAsString());
                    break;
            }
        } catch (Throwable t) {
            log.error("Exception handling message {} in sessionId {}", jsonMessage, sessionId,
↳t);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        sendError(session, t.getMessage());
    }
}

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

private void pause(String sessionId) {
    ...
}

private void resume(final WebSocketSession session) {
    ...
}

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

private void getPosition(final WebSocketSession session) {
    ...
}

private void stop(String sessionId) {
    ...
}

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

```

In the following snippet, we can see the start method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (WebRtcEndpoint and PlayerEndpoint) and makes the connections between them and plays the video. A startResponse message is sent back to the client with the SDP answer. When the MediaConnected event is received, info about the video is retrieved and sent back to the client in a videoInfo message.

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    final UserSession user = new UserSession(); MediaPipeline pipeline =
    kurento.createMediaPipeline(); user.setMediaPipeline(pipeline);
    WebRtcEndpoint webRtcEndpoint = new
    WebRtcEndpoint.Builder(pipeline).build();
    user.setWebRtcEndpoint(webRtcEndpoint); String videourl =
    jsonMessage.get("videourl").getString(); final PlayerEndpoint
    playerEndpoint = new PlayerEndpoint.Builder(pipeline, videourl).build();
    user.setPlayerEndpoint(playerEndpoint); users.put(session.getId(), user);

    playerEndpoint.connect(webRtcEndpoint);
}

```

(continues on next page)

(continued from previous page)

```

// 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);
    }
});

```

(continues on next page)

(continued from previous page)

```

playerEndpoint.addEndOfStreamListener(new
EventListener<EndOfStreamEvent>() {
    @Override public void onEvent(EndOfStreamEvent event) {
        log.info("EndOfStreamEvent: {}", event.getTimestampMillis());
        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");

```

(continues on next page)

(continued from previous page)

```
        response.addProperty("message", "Seek failed"); sendMessage(session,
        response.toString());
    }
}
```

The `getPosition` calls the method `getPosition` of the `PlayerEndpoint` of the current *user*. A position message is sent back to the client with the actual position of the video.

```
private void getPosition(final WebSocketSession session) {
    UserSession user = users.get(session.getId());

    if (user != null) {
        long position = user.getPlayerEndpoint().getPosition();

        JsonObject response = new JsonObject(); response.addProperty("id",
        "position"); response.addProperty("position", position);
        sendMessage(session, response.toString());
    }
}
```

The `stop` method is quite simple: it searches the *user* by *sessionId* and stops the `PlayerEndpoint`. Finally, it releases the media elements and removes the user from the list of active users.

```
private void stop(String sessionId) {
    UserSession user = users.remove(sessionId);

    if (user != null) {
        user.release();
    }
}
```

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

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


Client-Side Logic

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

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

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

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

  switch (parsedMessage.id) {
    case 'startResponse':
      startResponse(parsedMessage);
      break;
    case 'error':
      if (state == I_AM_STARTING) {
        setState(I_CAN_START);
      }
      onError('Error message from server: ' + parsedMessage.message);
      break;
    case 'playEnd':
      playEnd();
      break;
    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);
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

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
    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
    }

    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);
}

```

(continues on next page)

(continued from previous page)

```

}

function onError(error) {
    console.error(error);
}

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

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

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

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

function pause() {
    togglePause()
    console.log('Pausing video ...');
    var message = {
        id : 'pause'
    }
    sendMessage(message);
}

function resume() {
    togglePause()
    console.log('Resuming video ...');
    var message = {
        id : 'resume'
    }
    sendMessage(message);
}

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

        var message = {

```

(continues on next page)

(continued from previous page)

```
        id : 'stop'
    }
    sendMessage(message);
}
hideSpinner(video);
}

function playEnd() {
    setState(I_CAN_START);
    hideSpinner(video);
}

function doSeek() {
    var message = {
        id : 'doSeek',
        position: document.getElementById("seekPosition").value
    }
    sendMessage(message);
}

function getPosition() {
    var message = {
        id : 'getPosition'
    }
    sendMessage(message);
}

function showVideoData(parsedMessage) {
    //Show video info
    isSeekable = parsedMessage.isSeekable;
    if (isSeekable) {
        document.getElementById('isSeekable').value = "true";
        enableButton('#doSeek', 'doSeek()');
    } else {
        document.getElementById('isSeekable').value = "false";
    }

    document.getElementById('initSeek').value = parsedMessage.initSeekable;
    document.getElementById('endSeek').value = parsedMessage.endSeekable;
    document.getElementById('duration').value = parsedMessage.videoDuration;

    enableButton('#getPosition', 'getPosition()');
}

function sendMessage(message) {
    var jsonMessage = JSON.stringify(message);
    console.log('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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.10 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.

7.10.1 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.git
cd kurento/tutorials/java/datachannel-send-qr/
git checkout 7.0.0
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 `kurento-module-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 kurento-module-datachannelexample
```

Understanding this example

To implement this behavior we have to create a *Media Pipeline* composed by one **PlayerEndpoint**, one **KmsSendData** and one **WebRtcEndpoint**. The **PlayerEndpoint** plays a video and it detects QR codes into the images. The info about detected codes is sent through data channels (**KmsSendData**) from the Kurento media server to the browser (**WebRtcEndpoint**). The browser shows the info in a text form.

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

```
@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) {
```

(continues on next page)

(continued from previous page)

```

    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.

```

public class SendDataChannelHandler extends TextWebSocketHandler {

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

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

    @Autowired
    private KurentoClient kurento;

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

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

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

                UserSession user = users.get(session.getId());

```

(continues on next page)

(continued from previous page)

```

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

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

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

```

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

```

private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {
        // User session
        UserSession user = new UserSession();
        MediaPipeline pipeline = kurento.createMediaPipeline();
        user.setMediaPipeline(pipeline);
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).
↪useDataChannels()
            .build();
        user.setWebRtcEndpoint(webRtcEndpoint);
        PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline,
            "https://raw.githubusercontent.com/Kurento/test-files/main/video/filter/
↪barcodes.webm").build();
        user.setPlayer(player);
        users.put(session.getId(), user);

        // ICE candidates
        webRtcEndpoint.addIceCandidateFoundListener(new EventListener
↪<IceCandidateFoundEvent>() {
            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
            }
        });
    }
}

```

(continues on next page)

(continued from previous page)

```

        response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
    }
    try {
        synchronized (session) {
            session.sendMessage(new TextMessage(response.toString()));
        }
    } catch (IOException e) {
        log.debug(e.getMessage());
    }
}
});

// Media logic
KmsSendData kmsSendData = new KmsSendData.Builder(pipeline).build();

player.connect(kmsSendData);
kmsSendData.connect(webRtcEndpoint);

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

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

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

webRtcEndpoint.gatherCandidates();
player.play();

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

```

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

```

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

```

Client-Side Logic

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

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

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

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

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

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

  var servers = null;
  var configuration = null;
}
```

(continues on next page)

(continued from previous page)

```

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);
}

```

(continues on next page)

(continued from previous page)

```
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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.11 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.

7.11.1 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.

Note: This demo needs the `kurento-module-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 kurento-module-datachannelexample
```

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.git
cd kurento/tutorials/java/datachannel-show-text/
git checkout 7.0.0
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 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.

The additional Kurento module *datachannexample* (installed with the package “*kurento-module-datachannexample*”) uses a `textoverlay` GStreamer filter in order to print text on top of the remote 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.

```
@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
```

(continues on next page)

(continued from previous page)

```

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

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

```

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

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

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

```

public class ShowDataChannelHandler extends TextWebSocketHandler {

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

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

    @Autowired
    private KurentoClient kurento;

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

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

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

```

(continues on next page)

(continued from previous page)

```

        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.

```

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) {

```

(continues on next page)

(continued from previous page)

```

        session.sendMessage(new TextMessage(response.toString()));
    }
} catch (IOException e) {
    log.debug(e.getMessage());
}
}
});

// Media logic
KmsShowData kmsShowData = new KmsShowData.Builder(pipeline).build();

webRtcEndpoint.connect(kmsShowData);
kmsShowData.connect(webRtcEndpoint);

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

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

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

webRtcEndpoint.gatherCandidates();

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

```

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

```

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

```

Client-Side Logic

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

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

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

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

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

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

  var servers = null;
  var configuration = null;
}
```

(continues on next page)

(continued from previous page)

```

var peerConnection = new RTCPeerConnection(servers, configuration);

console.log("Creating channel");
var dataConstraints = null;

channel = peerConnection.createDataChannel(getChannelName (), dataConstraints);

channel.onopen = onSendChannelStateChange;
channel.onclose = onSendChannelStateChange;

function onSendChannelStateChange(){
    if(!channel) return;
    var readyState = channel.readyState;
    console.log("sencChannel state changed to " + readyState);
    if(readyState == 'open'){
        dataChannelSend.disabled = false;
        dataChannelSend.focus();
        $('#send').attr('disabled', false);
    } else {
        dataChannelSend.disabled = true;
        $('#send').attr('disabled', true);
    }
}

var sendButton = document.getElementById('send');
var dataChannelSend = document.getElementById('dataChannelSend');

sendButton.addEventListener("click", function(){
    var data = dataChannelSend.value;
    console.log("Send button pressed. Sending data " + data);
    channel.send(data);
    dataChannelSend.value = "";
});

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

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

function closeChannels(){

```

(continues on next page)

(continued from previous page)

```

    if(channel){
        channel.close();
        $('#dataChannelSend').disabled = true;
        $('#send').attr('disabled', true);
        channel = null;
    }
}

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

function onError(error) {
    console.error(error);
}

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

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

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

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

function stop() {
    console.log("Stopping video call ...");
    setState(I_CAN_START);
    if (webRtcPeer) {
        closeChannels();

        webRtcPeer.dispose();
        webRtcPeer = null;
    }
}

```

(continues on next page)

(continued from previous page)

```

    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*:

```

<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>
</dependencies>

```

(continues on next page)

(continued from previous page)

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

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

7.11.2 JavaScript - Hello World with Data Channels

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/hello-world-data-channel/
git checkout 7.0.0
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 **kurento-module-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 kurento-module-datachannelexample
```

Understanding this example

The logic of the application is quite simple: the local stream is sent to Kurento Media Server, which returns it back to the client without modifications. To implement this behavior we need to create a [Media Pipeline](#) with a single [Media Element](#), i.e. of type **WebRtcEndpoint**, which holds the capability of exchanging full-duplex (bidirectional) WebRTC media flows. It is important to set value of property `useDataChannels` to true during **WebRtcEndpoint** creation. This media element is connected to itself in order to deliver back received Media.

The application creates a channel between **PeerConnection** and **WebRtcEndpoint** used for message delivery.

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

JavaScript Logic

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

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

In addition, these two JavaScript libraries are also required:

- **Bootstrap** : Web framework for developing responsive web sites.
- **jquery.js** : Cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

- **adapter.js** : WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **ekko-lightbox** : Module for Bootstrap to open modal images, videos, and galleries.
- **demo-console** : Custom JavaScript console.

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

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

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

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

    [...]
})
```

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

Inside this function, a call to *generateOffer* is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the *KurentoClient* class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port TCP 8433 at the same host than the HTTP serving the application.

```
[...]

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:

```
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": "7.0.0",  
  "kurento-utils": "7.0.0"  
}
```

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

7.12 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.

7.12.1 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.git  
cd kurento/tutorials/java/hello-world-recording/  
git checkout 7.0.0  
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 **RecorderEndpoint**.

The second part of this demo shows how to play recorded media. To achieve this, we need to create a *Media Pipeline* composed by a **WebRtcEndpoint** and a **PlayerEndpoint**. The *uri* property of the player is the uri of the recorded file.

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

(continues on next page)

(continued from previous page)

```

    return KurentoClient.create();
}

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

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

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

```

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

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

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

```

public class HelloWorldRecHandler extends TextWebSocketHandler {

    private static final String RECORDER_FILE_PATH = "file:///tmp/HelloWorldRecorded.webm";

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

    @Autowired
    private UserRegistry registry;

    @Autowired
    private KurentoClient kurento;

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

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

        UserSession user = registry.getBySession(session);
        if (user != null) {
            log.debug("Incoming message from user '{}': {}", user.getId(), jsonMessage);

```

(continues on next page)

(continued from previous page)

```

    } else {
        log.debug("Incoming message from new user: {}", jsonMessage);
    }

    switch (jsonMessage.get("id").getAsString()) {
        case "start":
            start(session, jsonMessage);
            break;
        case "stop":
        case "stopPlay":
            if (user != null) {
                user.release();
            }
            break;
        case "play":
            play(user, session, jsonMessage);
            break;
        case "onIceCandidate": {
            JsonObject jsonCandidate = jsonMessage.get("candidate").getAsJsonObject();

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

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

private void play(UserSession user, final WebSocketSession session, JsonObject
↪jsonMessage) {
    ...
}

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

```

In the following snippet, we can see the start method. It handles the ICE candidates gathering, creates a Media Pipeline, creates the Media Elements (WebRtcEndpoint and RecorderEndpoint) and make the connections among

them. A startResponse message is sent back to the client with the SDP answer.

```
private void start(final WebSocketSession session, JsonObject jsonMessage) {
    try {

        // 1. Media logic (webRtcEndpoint in loopback)
        MediaPipeline pipeline = kurento.createMediaPipeline();
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
        webRtcEndpoint.connect(webRtcEndpoint);

        MediaProfileSpecType profile = getMediaProfileFromMessage(jsonMessage);

        RecorderEndpoint recorder = new RecorderEndpoint.Builder(pipeline, RECORDER_FILE_
↳PATH)
            .withMediaProfile(profile).build();

        connectAccordingToProfile(webRtcEndpoint, recorder, profile);

        // 2. Store user session
        UserSession user = new UserSession(session);
        user.setMediaPipeline(pipeline);
        user.setWebRtcEndpoint(webRtcEndpoint);
        registry.register(user);

        // 3. SDP negotiation
        String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
        String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

        // 4. Gather ICE candidates
        webRtcEndpoint.addIceCandidateFoundListener(new EventListener
↳<IceCandidateFoundEvent>() {
            @Override
            public void onEvent(IceCandidateFoundEvent event) {
                JsonObject response = new JsonObject();
                response.addProperty("id", "iceCandidate");
                response.add("candidate", JsonUtils.toJsonObject(event.getCandidate()));
                try {
                    synchronized (session) {
                        session.sendMessage(new TextMessage(response.toString()));
                    }
                } catch (IOException e) {
                    log.error(e.getMessage());
                }
            }
        });

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

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

(continues on next page)

(continued from previous page)

```

        webRtcEndpoint.gatherCandidates();

        recorder.record();
    } catch (Throwable t) {
        log.error("Start error", t);
        sendError(session, t.getMessage());
    }
}

```

The play method, creates a Media Pipeline with the Media Elements (WebRtcEndpoint and PlayerEndpoint) and make the connections among them. It will then send the recorded media to the client.

```

private void play(UserSession user, final WebSocketSession session, JsonObject_
↪ jsonMessage) {
    try {

        // 1. Media logic
        final MediaPipeline pipeline = kurento.createMediaPipeline();
        WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();
        PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline, RECORDER_FILE_PATH).
↪ build();
        player.connect(webRtcEndpoint);

        // Player listeners
        player.addErrorListener(new EventListener<ErrorEvent>() {
            @Override
            public void onEvent(ErrorEvent event) {
                log.info("ErrorEvent for session '{}': {}", session.getId(), event.
↪ getDescription());
                sendPlayEnd(session, pipeline);
            }
        });
        player.addEndOfStreamListener(new EventListener<EndOfStreamEvent>() {
            @Override
            public void onEvent(EndOfStreamEvent event) {
                log.info("EndOfStreamEvent for session '{}'", session.getId());
                sendPlayEnd(session, pipeline);
            }
        });

        // 2. Store user session
        user.setMediaPipeline(pipeline);
        user.setWebRtcEndpoint(webRtcEndpoint);

        // 3. SDP negotiation
        String sdpOffer = jsonMessage.get("sdpOffer").getAsString();
        String sdpAnswer = webRtcEndpoint.processOffer(sdpOffer);

        JsonObject response = new JsonObject();
        response.addProperty("id", "playResponse");
        response.addProperty("sdpAnswer", sdpAnswer);
    }
}

```

(continues on next page)

(continued from previous page)

```

// 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':
      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 ...');
```

(continues on next page)

(continued from previous page)

```

var options = {
    localVideo : videoInput,
    remoteVideo : videoOutput,
    mediaConstraints : getConstraints(),
    onIceCandidate
}

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

function onOffer(error, offerSdp) {
    if (error)
        return console.error('Error generating the offer');
    console.info('Invoking SDP offer callback function ' + location.host);
    var message = {
        id : 'start',
        sdpOffer : offerSdp,
        mode : $('input[name="mode"]:checked').val()
    }
    sendMessage(message);
}

function onError(error) {
    console.error(error);
}

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

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

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

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

function stop() {

```

(continues on next page)

(continued from previous page)

```

var stopMessageId = (state == IN_CALL) ? 'stop' : 'stopPlay';
console.log('Stopping video while in ' + state + '...');
setState(POST_CALL);
if (webRtcPeer) {
    webRtcPeer.dispose();
    webRtcPeer = null;

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

function play() {
    console.log("Starting to play recorded video...");

    // Disable start button
    setState(DISABLED);
    showSpinner(videoOutput);

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

    var options = {
        remoteVideo : videoOutput,
        mediaConstraints : getConstraints(),
        onIceCandidate : onIceCandidate
    }

    webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
        function(error) {
            if (error)
                return console.error(error);
            webRtcPeer.generateOffer(onPlayOffer);
        });
}

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

function getConstraints() {
    var mode = $('input[name="mode"]:checked').val();
    var constraints = {

```

(continues on next page)

(continued from previous page)

```

        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*:

```

<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>

```

(continues on next page)

(continued from previous page)

```

    <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: You can find the latest version of Kurento Java Client at [Maven Central](#).

7.12.2 JavaScript - Recorder

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/recorder/
git checkout 7.0.0
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`: consisting of a **WebRtcEndpoint** and a **RecorderEndpoint**.

The second part of this demo shows how to play recorded media. To achieve this, we need to create a *Media Pipeline* composed by a **WebRtcEndpoint** and a **PlayerEndpoint**. The `uri` property of the player is the uri of the recorded file.

There are two implementations for this demo to be found in github:

- Using `callbacks`.
- Using `yield`.

Note: The snippets are based in demo with callbacks.

JavaScript Logic

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

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

In addition, these two JavaScript libraries are also required:

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

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

```
var startRecordButton = document.getElementById("start");

startRecordButton.addEventListener("click", startRecording);

function startRecording() {
    var options = {
        localVideo: videoInput,
        remoteVideo: videoOutput
    };

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

    [...]
}
```

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

Inside this function, a call to `generateOffer` is performed. This function accepts a callback in which the SDP offer is received. In this callback we create an instance of the `KurentoClient` class that will manage communications with the Kurento Media Server. So, we need to provide the URI of its WebSocket endpoint. In this example, we assume it's listening in port TCP 8433 at the same host than the HTTP serving the application.

```
[...]

var args = getopt(location.search,
{
  default:
  {
    ws_uri: 'wss://' + location.hostname + ':8433/kurento',
    file_uri: 'file:///tmp/recorder_demo.webm', // file to be stored in media server
    ice_servers: undefined
  }
});

[...]

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

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

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

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

```
var elements =
[
  {type: 'RecorderEndpoint', params: {uri : args.file_uri}},
  {type: 'WebRtcEndpoint', params: {}}
]

pipeline.create(elements, function(error, elements){
  if (error) return onError(error);

  var recorder = elements[0]
  var webRtc   = elements[1]

  setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

  webRtc.processOffer(offer, function(error, answer) {
    if (error) return onError(error);
```

(continues on next page)

(continued from previous page)

```

    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 recorder 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);
  });
});

```

(continues on next page)

(continued from previous page)

```
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": "7.0.0",  
    "kurento-utils": "7.0.0"  
}
```

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

7.13 WebRTC statistics

This tutorial implements a *WebRTC loopback* and shows how to collect WebRTC statistics.

7.13.1 JavaScript - Loopback stats

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/loopback-stats/
git checkout 7.0.0
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.

```
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.

```
[...]

var args = getopt(location.search,
{
  default:
  {
    ws_uri: 'wss://' + location.hostname + ':8433/kurento',
    ice_servers: undefined
  }
});

[...]

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

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

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

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

```
pipeline.create("WebRtcEndpoint", function(error, webRtc) {
  if (error) return onError(error);

  webRtcEndpoint = webRtc;

  setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

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

    webRtc.gatherCandidates(onError);

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

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

    console.log("Loopback established");

    webRtcEndpoint.on('MediaStateChanged', function(event) {
      if (event.newState == "CONNECTED") {
        console.log("MediaState is CONNECTED ... printing stats...")
        activateStatsTimeout();
      }
    });
  });
});
```

(continues on next page)

(continued from previous page)

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

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": "7.0.0",  
  "kurento-utils": "7.0.0"  
}
```

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

7.14 Chroma Filter

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

7.14.1 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 `kurento-module-chroma` should be also installed:

```
sudo apt-get install kurento-module-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.git
cd kurento/tutorials/java/chroma/
git checkout 7.0.0
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:

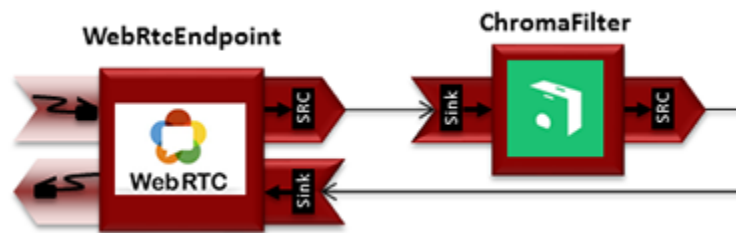
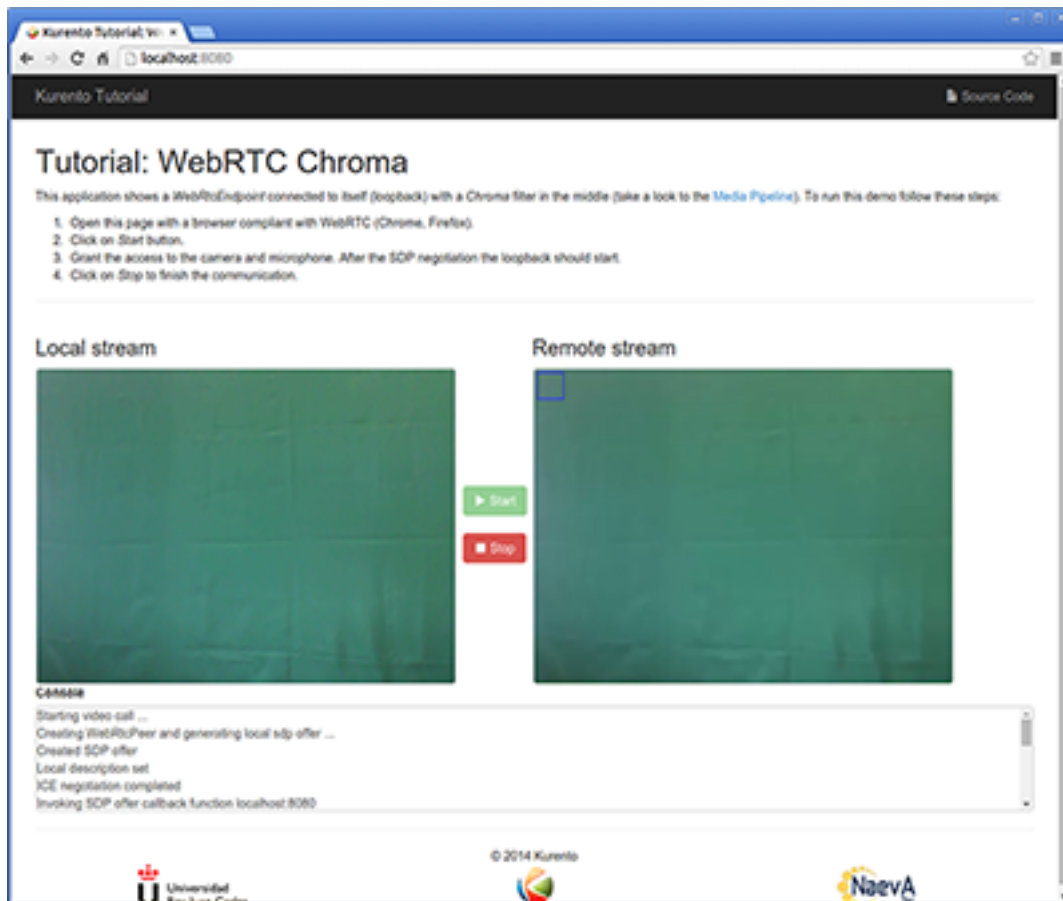


Fig. 40: *WebRTC with Chroma filter Media Pipeline*

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. 41: *Chroma calibration stage*

In the first second of the demo, a calibration process is done, by detecting the color inside that square. When the calibration is finished, the square disappears and the chroma is substituted with the configured image. Take into account that this process requires good lighting condition. Otherwise the chroma substitution will not be perfect. This behavior can be seen in the upper right corner of the following screenshot:

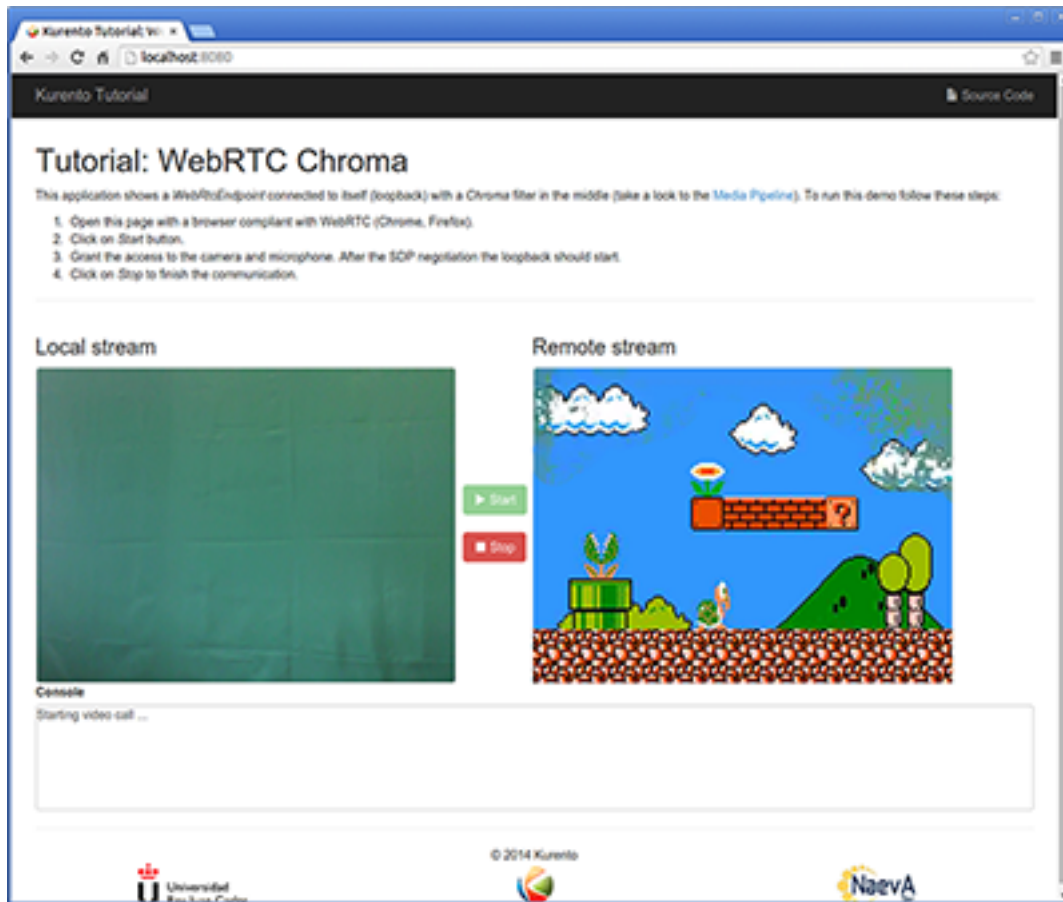


Fig. 42: Chroma filter in action

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

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

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

                @Override
```

(continues on next page)

(continued from previous page)

```

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

ChromaFilter chromaFilter = new ChromaFilter.Builder(pipeline,
    new WindowParam(5, 5, 40, 40)).build();
String appServerUrl = System.getProperty("app.server.url",
    ChromaApp.DEFAULT_APP_SERVER_URL);
chromaFilter.setBackground(appServerUrl + "/img/mario.jpg");

webRtcEndpoint.connect(chromaFilter);
chromaFilter.connect(webRtcEndpoint);

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

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);

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

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

```

Dependencies

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

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

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

7.14.2 JavaScript Module - Chroma Filter

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/chroma/
git checkout 7.0.0
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:

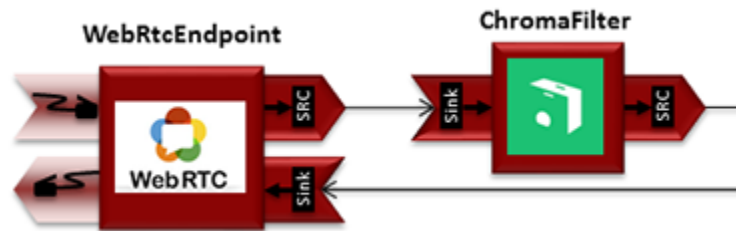


Fig. 43: WebRTC with Chroma filter Media Pipeline

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

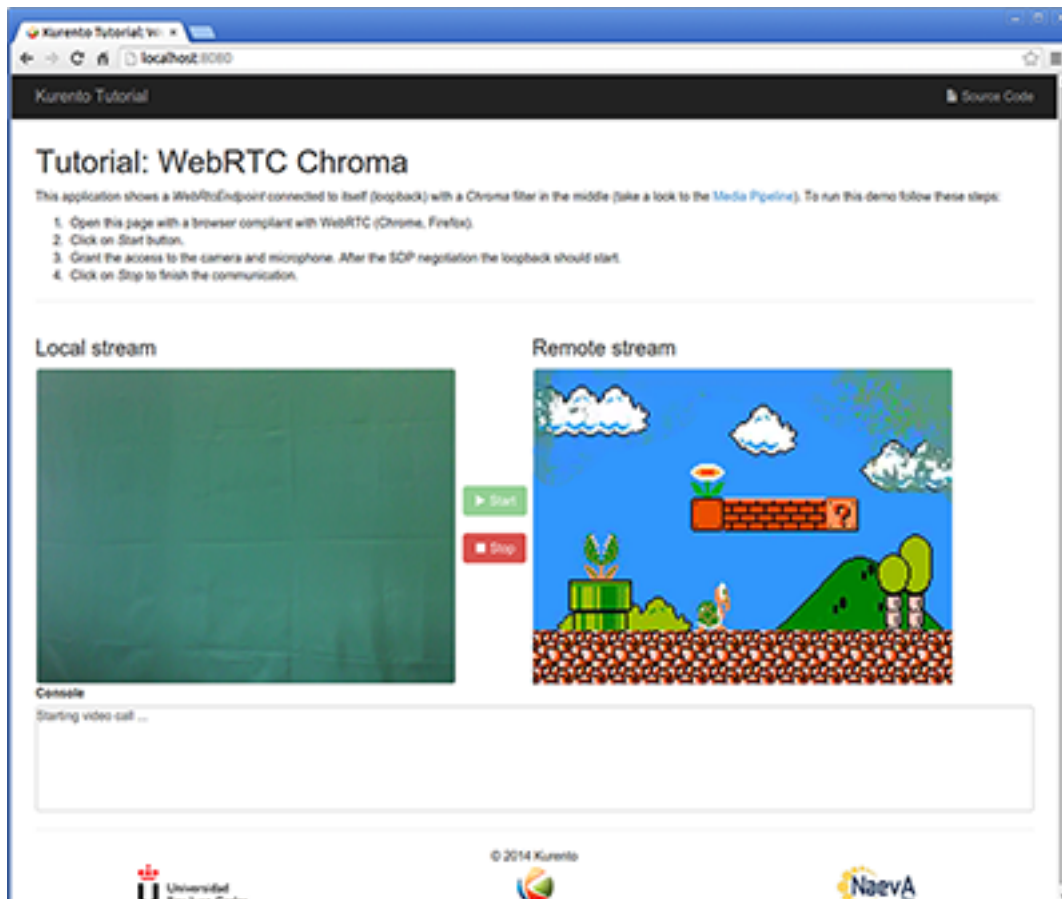
This example is a modified version of the *Magic Mirror* tutorial. In this case, this demo uses a **Chroma** instead of **FaceOverlay** filter.

In order to perform chroma detection, there must be a color calibration stage. To accomplish this step, at the beginning of the demo, a little square appears in upper left of the video, as follows:

In the first second of the demo, a calibration process is done, by detecting the color inside that square. When the calibration is finished, the square disappears and the chroma is substituted with the configured image. Take into account that this process requires lighting condition. Otherwise the chroma substitution will not be perfect. This behavior can be seen in the upper right corner of the following screenshot:

Note: Modules can have options. For configure these options, you need get the constructor to them. In JavaScript and Node.js, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

Fig. 44: *Chroma calibration stage*

Fig. 45: *Chroma filter in action*

The media pipeline of this demo is implemented in the JavaScript logic as follows:

```
...
kurentoClient.register('kurento-module-chroma')
const WindowParam = kurentoClient.getComplexType('chroma.WindowParam')
...

kurentoClient(args.ws_uri, function(error, client) {
  if (error) return onError(error);

  client.create('MediaPipeline', function(error, _pipeline) {
    if (error) return onError(error);

    pipeline = _pipeline;

    console.log("Got MediaPipeline");

    pipeline.create('WebRtcEndpoint', function(error, webRtc) {
      if (error) return onError(error);

      setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

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

        console.log("SDP answer obtained. Processing...");

        webRtc.gatherCandidates(onError);
        webRtcPeer.processAnswer(sdpAnswer);
      });

      console.log("Got WebRtcEndpoint");

      var options =
      {
        window: WindowParam({
          topRightCornerX: 5,
          topRightCornerY: 5,
          width: 30,
          height: 30
        })
      }

      pipeline.create('chroma.ChromaFilter', options, function(error, filter) {
        if (error) return onError(error);

        console.log("Got Filter");

        filter.setBackground(args.bg_uri, function(error) {
          if (error) return onError(error);

          console.log("Set Image");
        });
      });
    });
  });
});
```

(continues on next page)

(continued from previous page)

```

    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": "7.0.0",
    "kurento-utils": "7.0.0"
    "kurento-module-pointerdetector": "7.0.0"
}

```

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

```
bower install
```

Note: You can find the latest versions at [Bower](#).

7.14.3 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.js 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 `kurento-module-chroma` should be also installed:

```
sudo apt-get install kurento-module-chroma
```

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

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

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.git  
cd kurento/tutorials/javascript-node/chroma/  
git checkout 7.0.0  
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
```

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:

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.

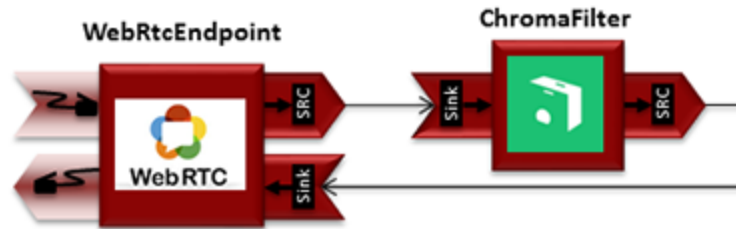


Fig. 46: WebRTC with Chroma filter Media Pipeline

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:

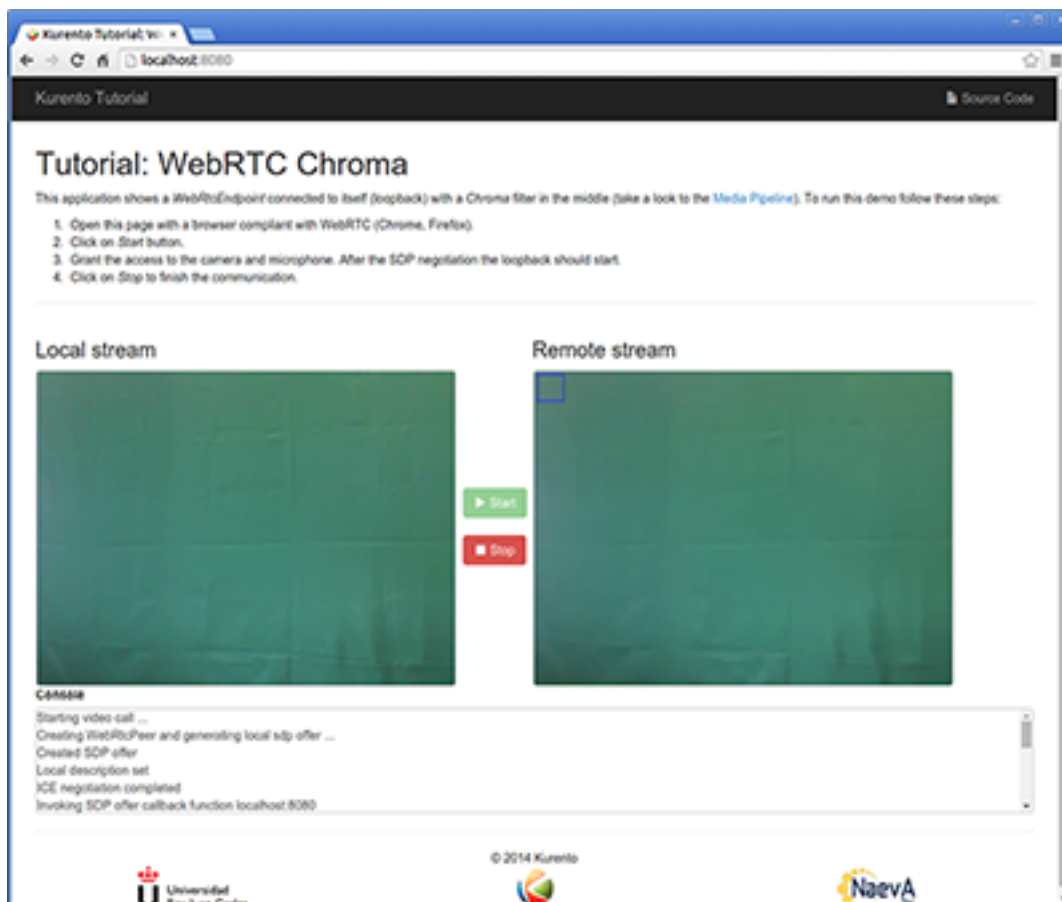


Fig. 47: Chroma calibration stage

In the first second of the demo, a calibration process is done, by detecting the color inside that square. When the calibration is finished, the square disappears and the chroma is substituted with the configured image. Take into account that this process requires lighting condition. Otherwise the chroma substitution will not be perfect. This behavior can be seen in the upper right corner of the following screenshot:

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the

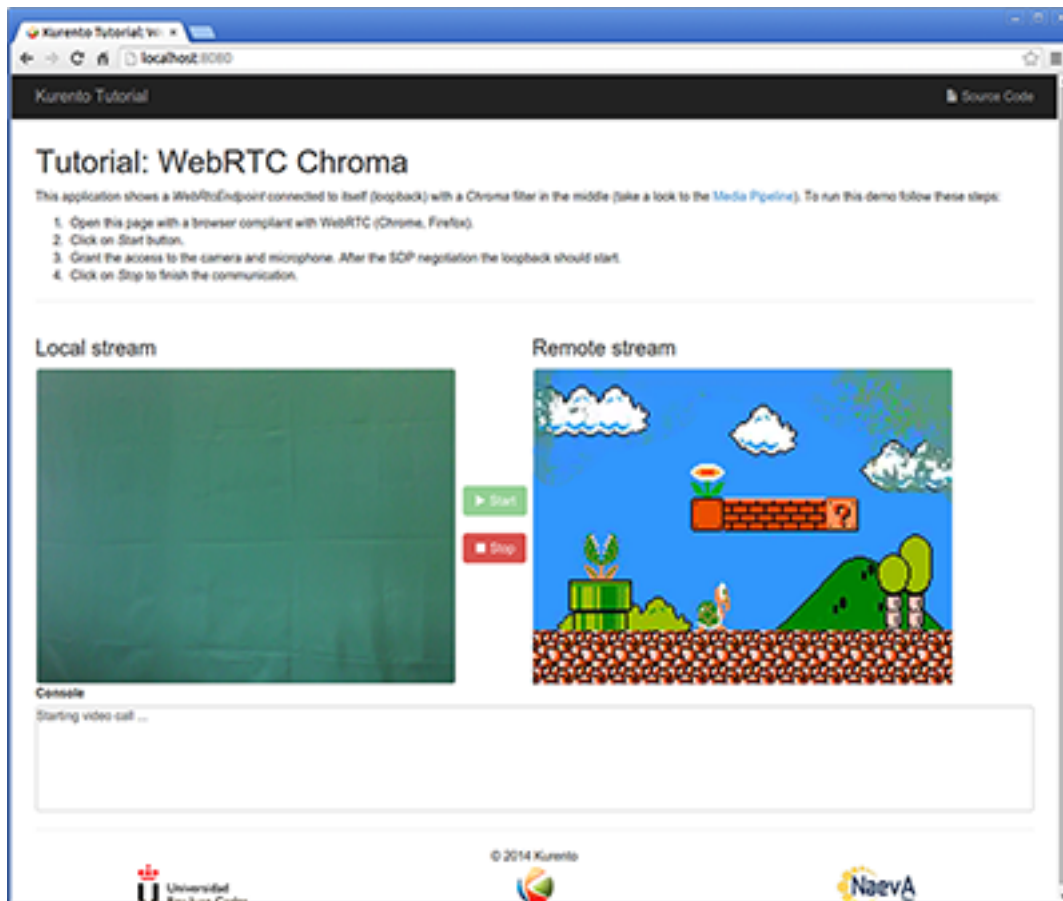


Fig. 48: Chroma filter in action

code.

The media pipeline of this demo is implemented in the JavaScript logic as follows:

```
...
kurento.register('kurento-module-chroma');
...

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

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

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

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

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

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

          webRtcEndpoint.on('IceCandidateFound', 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) {
```

(continues on next page)

(continued from previous page)

```

        pipeline.release();
        return callback(error);
    }

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

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

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

        var options = {
            window: kurento.getComplexType('chroma.WindowParam')({
                topRightCornerX: 5,
                topRightCornerY: 5,
                width: 30,
                height: 30
            })
        }
        pipeline.create('chroma.ChromaFilter', options, function(error, filter) {
            if (error) {
                return callback(error);
            }

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

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

```

(continues on next page)

(continued from previous page)

```
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" : "7.0.0"
}
```

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" : "7.0.0",
  "kurento-module-pointerdetector": "7.0.0"
}
```

Note: You can find the latest versions at [npm](#) and [Bower](#).

7.15 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.

7.15.1 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 `kurento-module-crowddetector` should be also installed:

```
sudo apt-get install kurento-module-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.git
cd kurento/tutorials/java/crowddetector/
git checkout 7.0.0
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:

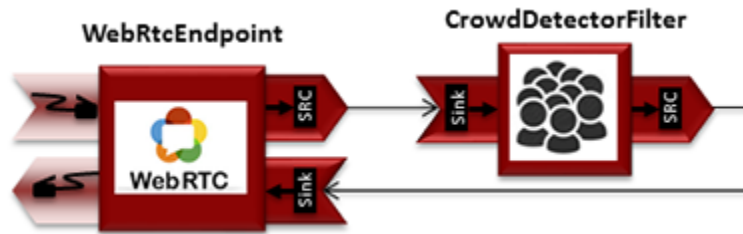


Fig. 49: WebRTC with crowdDetector filter Media Pipeline

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 analysed 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 a 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.

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*.



Fig. 50: *Crowd detection sample*

- `setFluidityLevelMed`: Fluidity level (0-100%) for the category *medium*.
- `setFluidityLevelMax`: Fluidity level (0-100%) for the category *maximum*.
- `setFluidityNumFramesToEvent`: Number of consecutive frames detecting a fluidity level to rise a event.
- `setOccupancyLevelMin`: Occupancy level (0-100%) for the category *minimum*.
- `setOccupancyLevelMed`: Occupancy level (0-100%) for the category *medium*.
- `setOccupancyLevelMax`: Occupancy level (0-100%) for the category *maximum*.
- `setOccupancyNumFramesToEvent`: Number of consecutive frames detecting a occupancy level to rise a event.
- `setSendOpticalFlowEvent`: Boolean value that indicates whether or not directions events are going to be tracked by the filter. Be careful with this feature, since it is very demanding in terms of resource usage (CPU, memory) in the media server. Set to true this parameter only when you are going to need directions events in your client-side.
- `setOpticalFlowNumFramesToEvent`: Number of consecutive frames detecting a direction level to rise a event.
- `setOpticalFlowNumFramesToReset`: Number of consecutive frames detecting a occupancy level in which the counter is reset.
- `setOpticalFlowAngleOffset`: Counterclockwise offset of the angle. This parameters is useful to move the default axis for directions (0°=north, 90°=east, 180°=south, 270°=west).

All in all, the media pipeline of this demo is implemented as follows:

```
// Media Logic (Media Pipeline and Elements)
MediaPipeline pipeline = kurento.createMediaPipeline();
pipelines.put(session.getId(), pipeline);

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

List<RegionOfInterest> rois = new ArrayList<>();
List<RelativePoint> points = new ArrayList<RelativePoint>();

points.add(new RelativePoint(0, 0));
points.add(new RelativePoint(0.5F, 0));
```

(continues on next page)

(continued from previous page)

```

points.add(new RelativePoint(0.5F, 0.5F));
points.add(new RelativePoint(0, 0.5F));

RegionOfInterestConfig config = new RegionOfInterestConfig();

config.setFluidityLevelMin(10);
config.setFluidityLevelMed(35);
config.setFluidityLevelMax(65);
config.setFluidityNumFramesToEvent(5);
config.setOccupancyLevelMin(10);
config.setOccupancyLevelMed(35);
config.setOccupancyLevelMax(65);
config.setOccupancyNumFramesToEvent(5);
config.setSendOpticalFlowEvent(false);
config.setOpticalFlowNumFramesToEvent(3);
config.setOpticalFlowNumFramesToReset(3);
config.setOpticalFlowAngleOffset(0);

rois.add(new RegionOfInterest(points, config, "roi0"));

CrowdDetectorFilter crowdDetectorFilter = new CrowdDetectorFilter.Builder(
    pipeline, rois).build();

webRtcEndpoint.connect(crowdDetectorFilter);
crowdDetectorFilter.connect(webRtcEndpoint);

// addEventListener to crowddetector
crowdDetectorFilter.addCrowdDetectorDirectionListener(
    new EventListener<CrowdDetectorDirectionEvent>() {
        @Override
        public void onEvent(CrowdDetectorDirectionEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "directionEvent");
            response.addProperty("roiId", event.getRoiID());
            response.addProperty("angle",
                event.getDirectionAngle());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });

crowdDetectorFilter.addCrowdDetectorFluidityListener(
    new EventListener<CrowdDetectorFluidityEvent>() {
        @Override
        public void onEvent(CrowdDetectorFluidityEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "fluidityEvent");
            response.addProperty("roiId", event.getRoiID());

```

(continues on next page)

(continued from previous page)

```

        response.addProperty("level",
            event.getFluidityLevel());
        response.addProperty("percentage",
            event.getFluidityPercentage());
        try {
            session.sendMessage(new TextMessage(response
                .toString()));
        } catch (Throwable t) {
            sendError(session, t.getMessage());
        }
    }
});

crowdDetectorFilter.addCrowdDetectorOccupancyListener(
    new EventListener<CrowdDetectorOccupancyEvent>() {
        @Override
        public void onEvent(CrowdDetectorOccupancyEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "occupancyEvent");
            response.addProperty("roiId", event.getRoiID());
            response.addProperty("level",
                event.getOccupancyLevel());
            response.addProperty("percentage",
                event.getOccupancyPercentage());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });

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

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);
session.sendMessage(new TextMessage(response.toString()));

webRtcEndpoint.gatherCandidates();

```

Dependencies

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

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

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

7.15.2 JavaScript Module - Crowd Detector Filter

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/crowddetector/
git checkout 7.0.0
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:

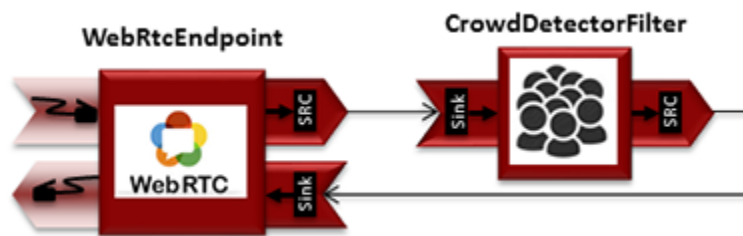


Fig. 51: *WebRTC with crowdDetector filter Media Pipeline*

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 a screenshot of a running example of `CrowdDetectorFilter`. In the picture below, we can see that there are two ROIs (bounded with white lines in the video). On these ROIs, we can see two different colors over the original video stream: red zones are drawn over detected static crowds (or moving slowly). Blue zones are drawn over the detected crowds moving fast.



Fig. 52: Crowd detection sample

Regarding crowd events, there are three types of events, namely:

- `CrowdDetectorFluidityEvent`. Event raised when a certain level of fluidity is detected in a ROI. Fluidity can be seen as the level of general movement in a crowd.
- `CrowdDetectorOccupancyEvent`. Event raised when a level of occupancy is detected in a ROI. Occupancy can be seen as the level of agglomeration in stream.
- `CrowdDetectorDirectionEvent`. Event raised when a movement direction is detected in a ROI by a crowd.

Both fluidity as occupancy are quantified in a relative metric from 0 to 100%. Then, both attributes are qualified into three categories: i) Minimum (min); ii) Medium (med); iii) Maximum (max).

Regarding direction, it is quantified as an angle (0-360°), where 0 is the direction from the central point of the video to the top (i.e., north), 90 correspond to the direction to the right (east), 180 is the south, and finally 270 is the west.

With all these concepts, now we can check out the Java server-side code of this demo. As depicted in the snippet below, we create a ROI by adding `RelativePoint` instances to a list. Each ROI is then stored into a list of `RegionOfInterest` instances.

Then, each ROI should be configured. To do that, we have the following methods:

- `fluidityLevelMin`: Fluidity level (0-100%) for the category *minimum*.
- `fluidityLevelMed`: Fluidity level (0-100%) for the category *medium*.

- `fluidityLevelMax`: Fluidity level (0-100%) for the category *maximum*.
- `fluidityNumFramesToEvent`: Number of consecutive frames detecting a fluidity level to rise a event.
- `occupancyLevelMin`: Occupancy level (0-100%) for the category *minimum*.
- `occupancyLevelMed`: Occupancy level (0-100%) for the category *medium*.
- `occupancyLevelMax`: Occupancy level (0-100%) for the category *maximum*.
- `occupancyNumFramesToEvent`: Number of consecutive frames detecting a occupancy level to rise a event.
- `sendOpticalFlowEvent`: Boolean value that indicates whether or not directions events are going to be tracked by the filter. Be careful with this feature, since it is very demanding in terms of resource usage (CPU, memory) in the media server. Set to true this parameter only when you are going to need directions events in your client-side.
- `opticalFlowNumFramesToEvent`: Number of consecutive frames detecting a direction level to rise a event.
- `opticalFlowNumFramesToReset`: Number of consecutive frames detecting a occupancy level in which the counter is reset.
- `opticalFlowAngleOffset`: Counterclockwise offset of the angle. This parameters is useful to move the default axis for directions (0°=north, 90°=east, 180°=south, 270°=west).

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

All in all, the media pipeline of this demo is implemented as follows:

```
...
kurentoClient.register('kurento-module-crowddetector')
const RegionOfInterest      = kurentoClient.getComplexType('crowddetector.
↳RegionOfInterest')
const RegionOfInterestConfig = kurentoClient.getComplexType('crowddetector.
↳RegionOfInterestConfig')
const RelativePoint         = kurentoClient.getComplexType('crowddetector.RelativePoint
↳')
...

kurentoClient(args.ws_uri, function(error, client) {
  if (error) return onError(error);

  client.create('MediaPipeline', function(error, p) {
    if (error) return onError(error);

    pipeline = p;

    console.log("Got MediaPipeline");

    pipeline.create('WebRtcEndpoint', function(error, webRtc) {
      if (error) return onError(error);

      console.log("Got WebRtcEndpoint");

      setIceCandidateCallbacks(webRtcPeer, webRtc, onError)
```

(continues on next page)

(continued from previous page)

```

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

    console.log("SDP answer obtained. Processing ...");

    webRtc.gatherCandidates(onError);

    webRtcPeer.processAnswer(sdpAnswer);
});

var options =
{
    rois:
    [
        RegionOfInterest({
            id: 'roi1',
            points:
            [
                RelativePoint({x: 0, y: 0}),
                RelativePoint({x: 0.5, y: 0}),
                RelativePoint({x: 0.5, y: 0.5}),
                RelativePoint({x: 0, y: 0.5})
            ],
            regionOfInterestConfig: RegionOfInterestConfig({
                occupancyLevelMin: 10,
                occupancyLevelMed: 35,
                occupancyLevelMax: 65,
                occupancyNumFramesToEvent: 5,
                fluidityLevelMin: 10,
                fluidityLevelMed: 35,
                fluidityLevelMax: 65,
                fluidityNumFramesToEvent: 5,
                sendOpticalFlowEvent: false,
                opticalFlowNumFramesToEvent: 3,
                opticalFlowNumFramesToReset: 3,
                opticalFlowAngleOffset: 0
            })
        })
    ]
}

pipeline.create('crowddetector.CrowdDetectorFilter', options, function(error, ↵
↵filter)
{
    if (error) return onError(error);

    console.log("Connecting...");

    filter.on('CrowdDetectorDirection', function (data){
        console.log("Direction event received in roi " + data.roiID +
            " with direction " + data.directionAngle);
    });
}

```

(continues on next page)

(continued from previous page)

```
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": "7.0.0",
  "kurento-utils": "7.0.0"
  "kurento-module-pointerdetector": "7.0.0"
}
```

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

```
bower install
```

Note: You can find the latest versions at [Bower](#).

7.15.3 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.js 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 `kurento-module-crowddetector` should be also installed:

```
sudo apt-get install kurento-module-crowddetector
```

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

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install -y nodejs
```

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.git
cd kurento/tutorials/javascript-node/crowddetector/
git checkout 7.0.0
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
```

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:

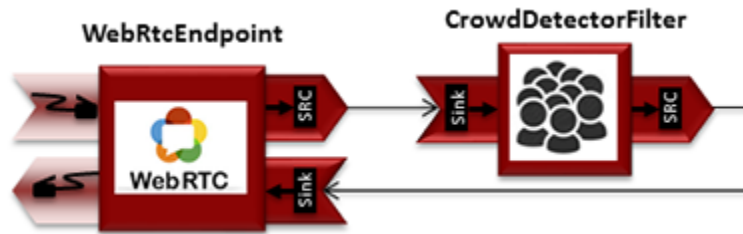


Fig. 53: WebRTC with crowdDetector filter Media Pipeline

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 a 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.

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*.



Fig. 54: *Crowd detection sample*

- fluidityLevelMed: Fluidity level (0-100%) for the category *medium*.
- fluidityLevelMax: Fluidity level (0-100%) for the category *maximum*.
- fluidityNumFramesToEvent: Number of consecutive frames detecting a fluidity level to rise a event.
- occupancyLevelMin: Occupancy level (0-100%) for the category *minimum*.
- occupancyLevelMed: Occupancy level (0-100%) for the category *medium*.
- occupancyLevelMax: Occupancy level (0-100%) for the category *maximum*.
- occupancyNumFramesToEvent: Number of consecutive frames detecting a occupancy level to rise a event.
- sendOpticalFlowEvent: Boolean value that indicates whether or not directions events are going to be tracked by the filter. Be careful with this feature, since it is very demanding in terms of resource usage (CPU, memory) in the media server. Set to true this parameter only when you are going to need directions events in your client-side.
- opticalFlowNumFramesToEvent: Number of consecutive frames detecting a direction level to rise a event.
- opticalFlowNumFramesToReset: Number of consecutive frames detecting a occupancy level in which the counter is reset.
- opticalFlowAngleOffset: Counterclockwise offset of the angle. This parameters is useful to move the default axis for directions (0°=north, 90°=east, 180°=south, 270°=west).

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

All in all, the media pipeline of this demo is implemented as follows:

```
...
kurento.register('kurento-module-crowddetector');
const RegionOfInterest      = kurento.getComplexType('crowddetector.RegionOfInterest');
const RegionOfInterestConfig = kurento.getComplexType('crowddetector.
↳RegionOfInterestConfig');
const RelativePoint         = kurento.getComplexType('crowddetector.RelativePoint');
...

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

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

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

      createMediaElements(pipeline, ws, function(error, webRtcEndpoint, filter) {
        if (error) {
```

(continues on next page)

(continued from previous page)

```

        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('IceCandidateFound', function(event) {
            var candidate = kurento.getComplexType('IceCandidate')(event.
↪candidate);

            ws.send(JSON.stringify({
                id : 'iceCandidate',
                candidate : candidate
            }));
        });

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

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

        webRtcEndpoint.gatherCandidates(function(error) {

```

(continues on next page)

(continued from previous page)

```

        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);
        });
    });
}

```

(continues on next page)

(continued from previous page)

```
}
```

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" : "7.0.0"  
}
```

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" : "7.0.0",  
  "kurento-module-pointerdetector": "7.0.0"  
}
```

Note: You can find the latest versions at [npm](#) and [Bower](#).

7.16 Plate Detector Filter

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

7.16.1 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 `kurento-module-platedetector` should be also installed:

```
sudo apt-get install kurento-module-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.git
cd kurento/tutorials/java/platedetector/
git checkout 7.0.0
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:

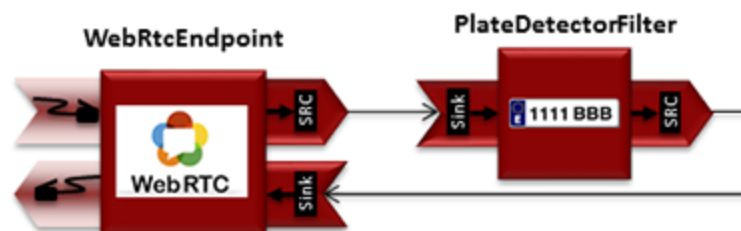


Fig. 55: *WebRTC with plateDetector filter Media Pipeline*

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:

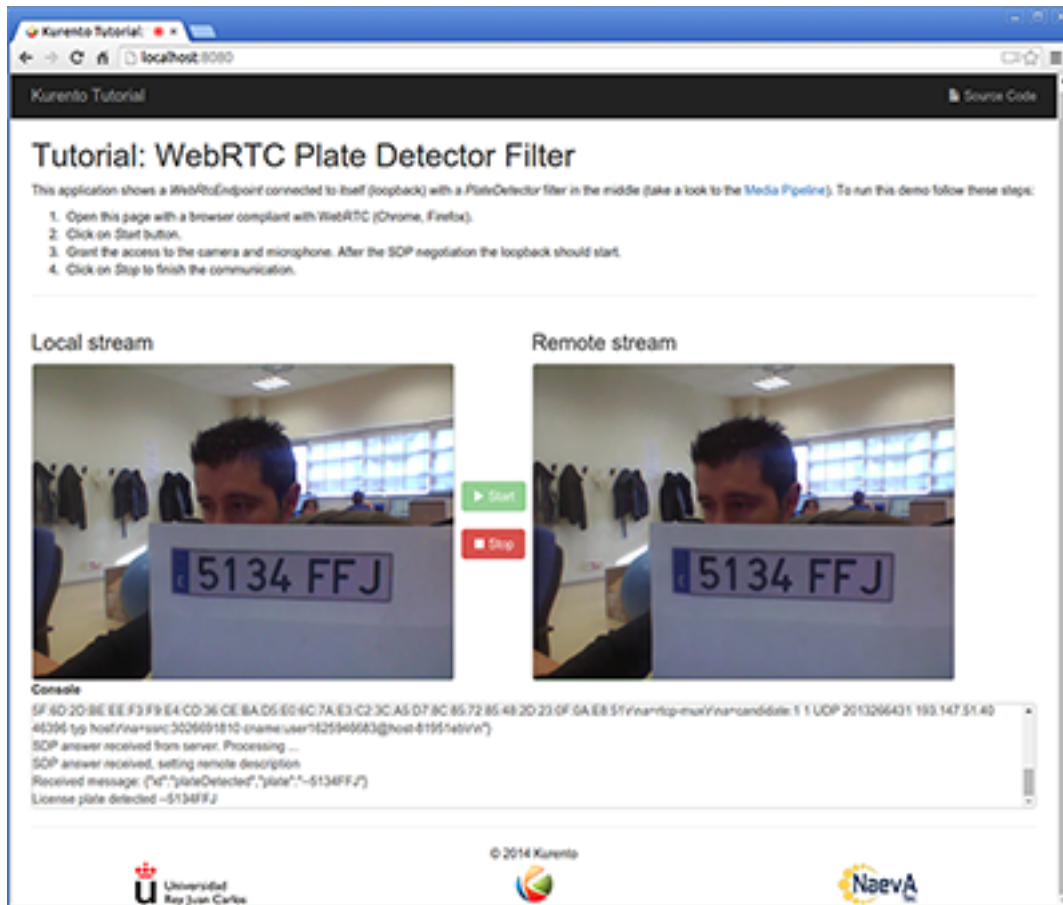


Fig. 56: Plate detector demo in action

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.

```
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>() {
```

(continues on next page)

(continued from previous page)

```

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

});

PlateDetectorFilter plateDetectorFilter = new PlateDetectorFilter.Builder(
    pipeline).build();

webRtcEndpoint.connect(plateDetectorFilter);
plateDetectorFilter.connect(webRtcEndpoint);

plateDetectorFilter
    .addPlateDetectedListener(new EventListener<PlateDetectedEvent>() {
        @Override
        public void onEvent(PlateDetectedEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "plateDetected");
            response.addProperty("plate", event.getPlate());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });

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

// Sending response back to client
JsonObject response = new JsonObject();
response.addProperty("id", "startResponse");
response.addProperty("sdpAnswer", sdpAnswer);

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

```

(continues on next page)

(continued from previous page)

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

Dependencies

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

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

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

7.16.2 JavaScript Module - Plate Detector Filter

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/platedetector/
git checkout 7.0.0
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 **kurento-module-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 kurento-module-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:

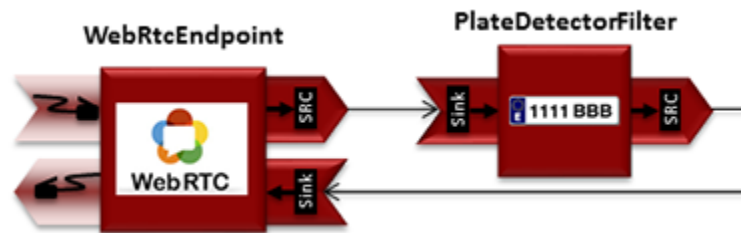


Fig. 57: WebRTC with plateDetector filter Media Pipeline

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:

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, 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`).

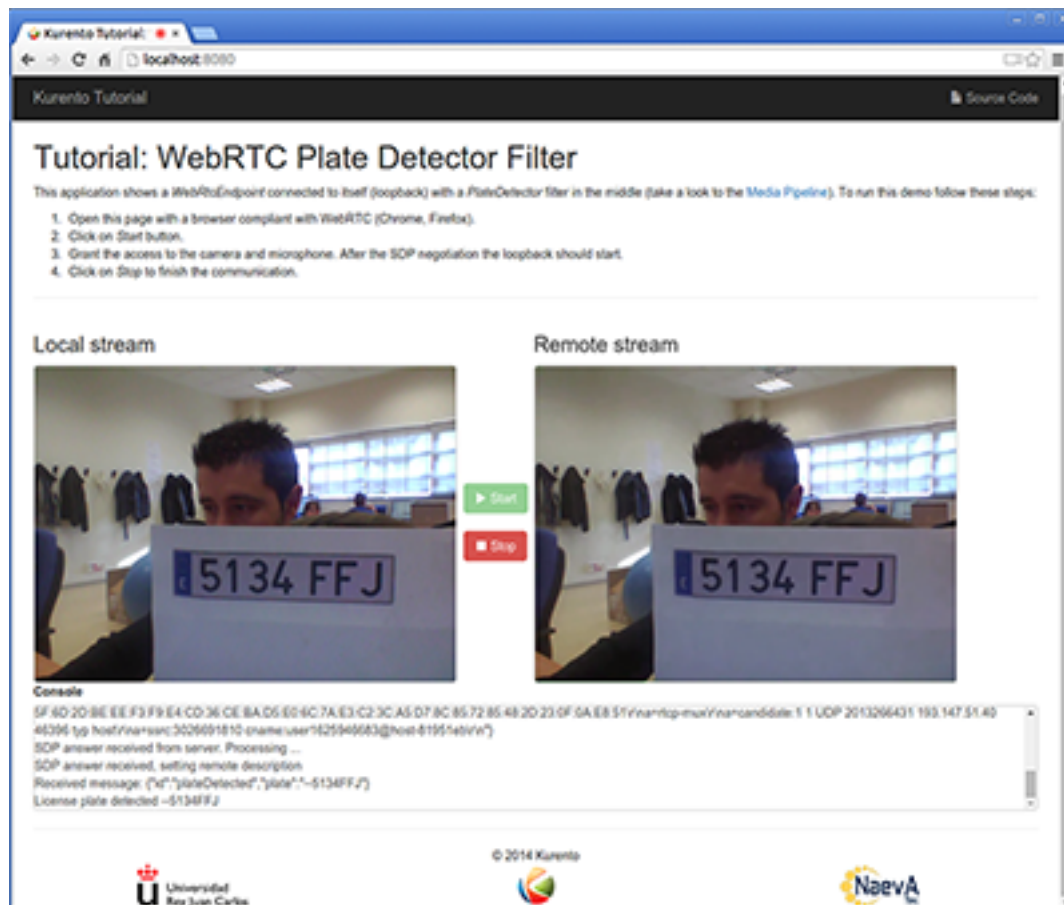


Fig. 58: Plate detector demo in action

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": "7.0.0",  
  "kurento-utils": "7.0.0"  
  "kurento-module-pointerdetector": "7.0.0"  
}
```

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

```
bower install
```

Note: You can find the latest versions at [Bower](#).

7.16.3 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.js 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 `kurento-module-platedetector` should be also installed:

```
sudo apt-get install kurento-module-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* in your system. In an Ubuntu machine, you can install it as follows:

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install -y nodejs
```

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.git
cd kurento/tutorials/javascript-node/platedetector/
git checkout 7.0.0
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
```

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:

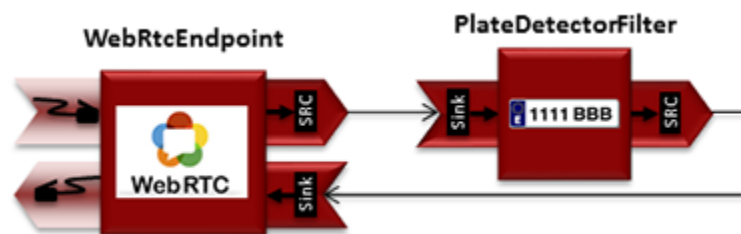


Fig. 59: WebRTC with plateDetector filter Media Pipeline

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:

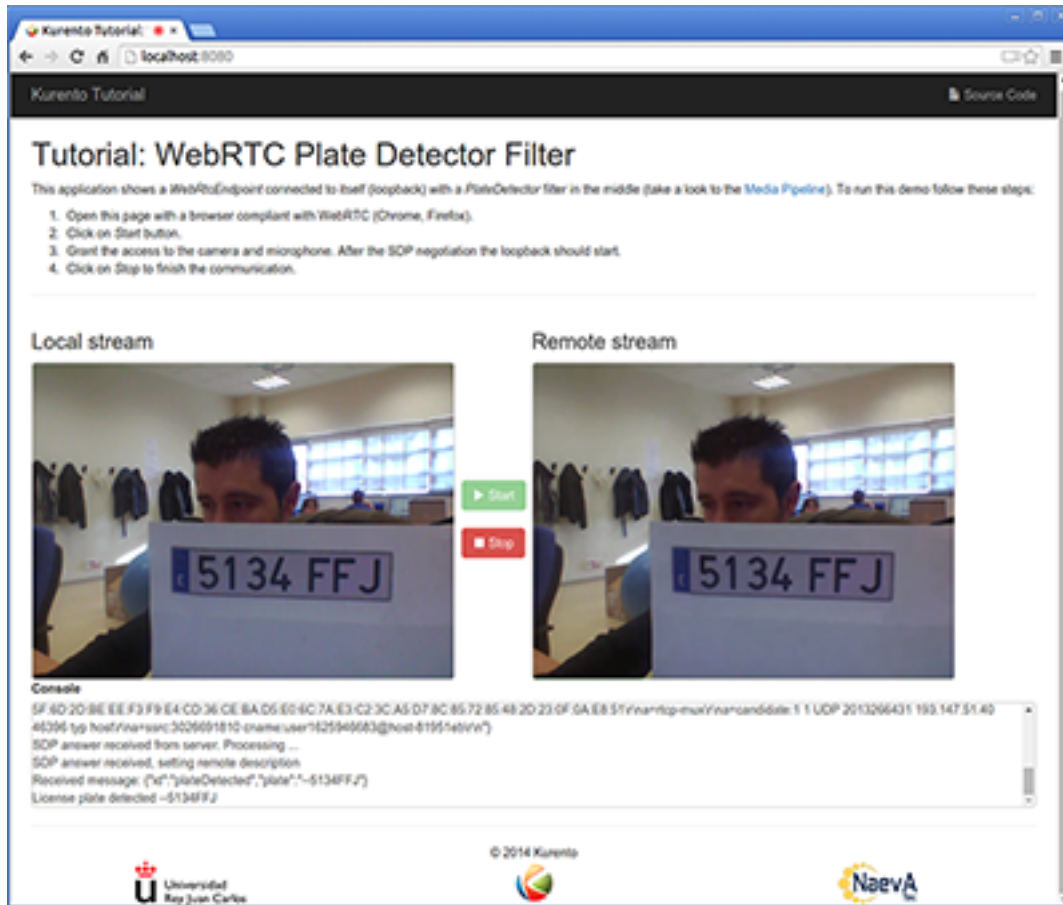


Fig. 60: Plate detector demo in action

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, 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');
  }
}
```

(continues on next page)

(continued from previous page)

```

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('IceCandidateFound', 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
                    }
                }

```

(continues on next page)

(continued from previous page)

```

        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" : "7.0.0"
}
```

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" : "7.0.0",
  "kurento-module-pointerdetector": "7.0.0"
}
```

Note: You can find the latest versions at [npm](#) and [Bower](#).

7.17 Pointer Detector Filter

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

7.17.1 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](#).

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 `kurento-module-pointerdetector` should be also installed:

```
sudo apt-get install kurento-module-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.git
cd kurento/tutorials/java/pointerdetector/
git checkout 7.0.0
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:

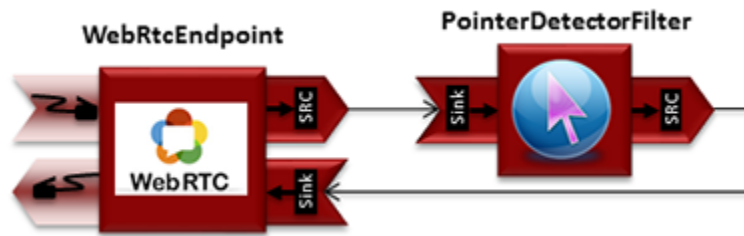


Fig. 61: WebRTC with PointerDetector filter in loopback Media Pipeline

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

This example is a modified version of the *Magic Mirror* tutorial. In this case, this demo uses a **PointerDetector** instead of **FaceOverlay** filter.

In order to perform pointer detection, there must be a calibration stage, in which the color of the pointer is registered by the filter. To accomplish this step, the pointer should be placed in a square visible in the upper left corner of the video after going through the filter, as follows:

When the desired color to track is filling that box, a calibration message is sent from the client to the server. This is done by clicking on the *Calibrate* blue button of the GUI.

After that, the color of the pointer is tracked in real time by Kurento Media Server. **PointerDetectorFilter** can also define regions in the screen called *windows* in which some actions are performed when the pointer is detected when the pointer enters (**WindowInEvent** event) and exits (**WindowOutEvent** event) the windows. This is implemented in the server-side logic as follows:

```
// Media Logic (Media Pipeline and Elements)
UserSession user = new UserSession();
MediaPipeline pipeline = kurento.createMediaPipeline();
user.setMediaPipeline(pipeline);
WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline)
    .build();
user.setWebRtcEndpoint(webRtcEndpoint);
users.put(session.getId(), user);

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

        @Override
        public void onEvent(IceCandidateFoundEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "iceCandidate");
            response.add("candidate", JsonUtils
                .toJsonObject(event.getCandidate()));
        }
    });
```

(continues on next page)

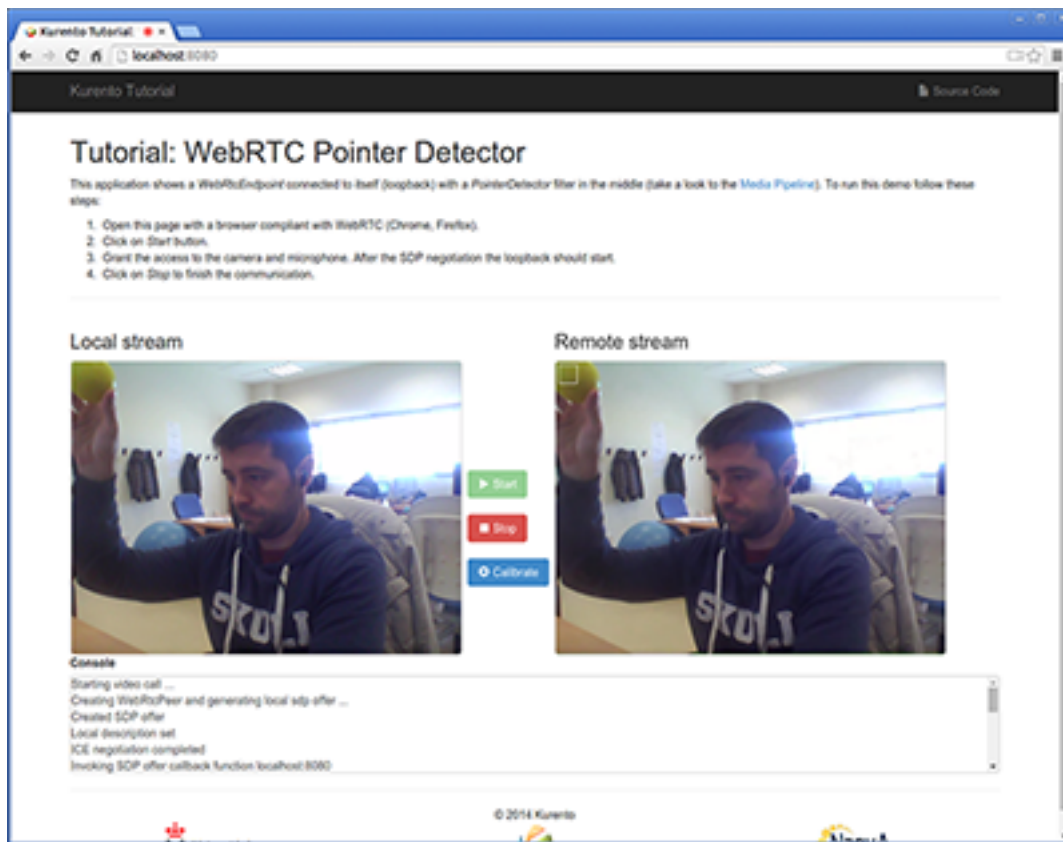


Fig. 62: *Pointer calibration stage*

(continued from previous page)

```

    try {
        synchronized (session) {
            session.sendMessage(new TextMessage(
                response.toString()));
        }
    } catch (IOException e) {
        log.debug(e.getMessage());
    }
}

});

pointerDetectorFilter = new PointerDetectorFilter.Builder(pipeline,
    new WindowParam(5, 5, 30, 30)).build();

pointerDetectorFilter
    .addWindow(new PointerDetectorWindowMediaParam("window0",
        50, 50, 500, 150));

pointerDetectorFilter
    .addWindow(new PointerDetectorWindowMediaParam("window1",
        50, 50, 500, 250));

webRtcEndpoint.connect(pointerDetectorFilter);
pointerDetectorFilter.connect(webRtcEndpoint);

pointerDetectorFilter
    .addWindowInListener(new EventListener<WindowInEvent>() {
        @Override
        public void onEvent(WindowInEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "windowIn");
            response.addProperty("roiId", event.getWindowId());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            } catch (Throwable t) {
                sendError(session, t.getMessage());
            }
        }
    });

pointerDetectorFilter
    .addWindowOutListener(new EventListener<WindowOutEvent>() {

        @Override
        public void onEvent(WindowOutEvent event) {
            JsonObject response = new JsonObject();
            response.addProperty("id", "windowOut");
            response.addProperty("roiId", event.getWindowId());
            try {
                session.sendMessage(new TextMessage(response
                    .toString()));
            }

```

(continues on next page)

(continued from previous page)

```

    } 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:

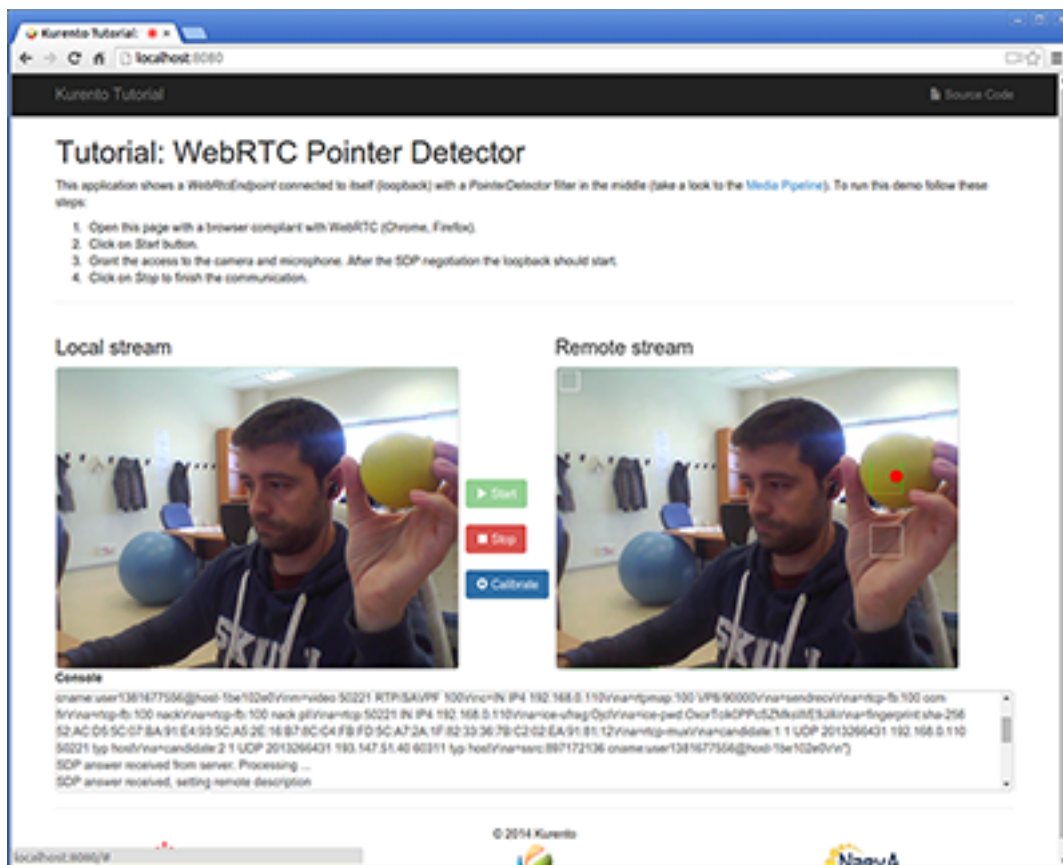


Fig. 63: Pointer tracking over a window

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>
</dependencies>
```

(continues on next page)

(continued from previous page)

```

</dependency>
<dependency>
  <groupId>org.webjars.bower</groupId>
  <artifactId>ekko-lightbox</artifactId>
</dependency>
</dependencies>

```

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

7.17.2 JavaScript Module - Pointer Detector Filter

Warning: Bower dependencies are not yet upgraded for Kurento 7.0.0.

Kurento tutorials that use pure browser JavaScript need to be rewritten to drop the deprecated Bower service and instead use a web resource packer. This has not been done, so these tutorials won't be able to download the dependencies they need to work. PRs would be appreciated!

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 -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/tutorials/javascript-browser/pointerdetector/
git checkout 7.0.0
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:

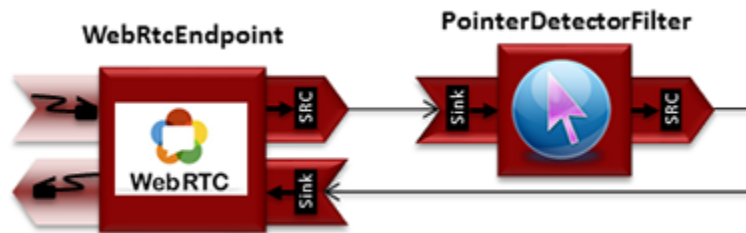
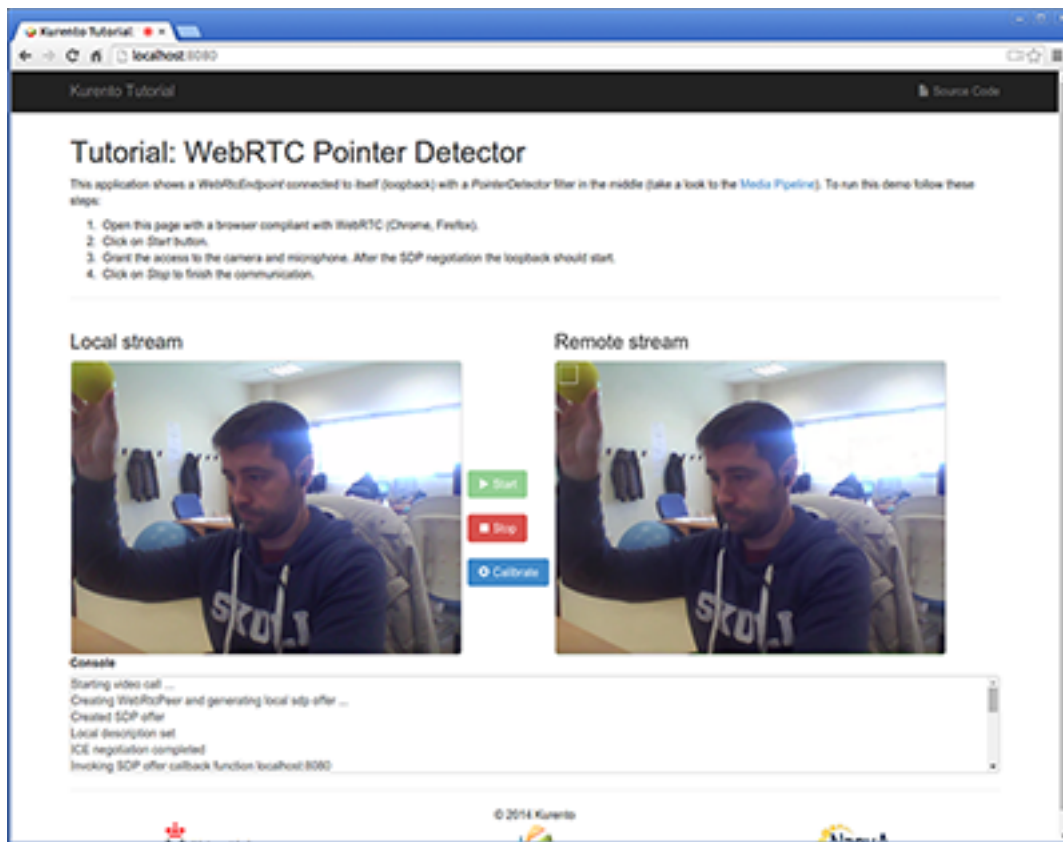


Fig. 64: *WebRTC with PointerDetector filter in loopback Media Pipeline*

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

This example is a modified version of the *Magic Mirror* tutorial. In this case, this demo uses a **PointerDetector** instead of **FaceOverlay** filter.

In order to perform pointer detection, there must be a calibration stage, in which the color of the pointer is registered by the filter. To accomplish this step, the pointer should be placed in a square in the upper left corner of the video, as follows:

Fig. 65: *Pointer calibration stage*

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

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

After that, the color of the pointer is tracked in real time by Kurento Media Server. `PointerDetectorFilter` can also define regions in the screen called *windows* in which some actions are performed when the pointer is detected when the pointer enters (`WindowIn` event) and exits (`WindowOut` event) the windows. This is implemented in the JavaScript logic as follows:

```
...
kurentoClient.register('kurento-module-pointerdetector')
const PointerDetectorWindowMediaParam = kurentoClient.getComplexType('pointerdetector.
↪PointerDetectorWindowMediaParam')
const WindowParam                      = kurentoClient.getComplexType('pointerdetector.
↪WindowParam')
...

kurentoClient(args.ws_uri, function(error, client) {
  if (error) return onError(error);

  client.create('MediaPipeline', function(error, _pipeline) {
    if (error) return onError(error);

    pipeline = _pipeline;

    console.log("Got MediaPipeline");

    pipeline.create('WebRtcEndpoint', function(error, webRtc) {
      if (error) return onError(error);

      console.log("Got WebRtcEndpoint");

      setIceCandidateCallbacks(webRtcPeer, webRtc, onError)

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

        console.log("SDP answer obtained. Processing ...");

        webRtc.gatherCandidates(onError);
        webRtcPeer.processAnswer(sdpAnswer);
      });

      var options =
      {
        calibrationRegion: WindowParam({
          topRightCornerX: 5,
          topRightCornerY: 5,
          width: 30,
```

(continues on next page)

(continued from previous page)

```

        height: 30
    });
};

pipeline.create('pointerdetector.PointerDetectorFilter', options, function(error, _
→filter) {
    if (error) return onError(error);

    filter = _filter;

    var options = PointerDetectorWindowMediaParam({
        id: 'window0',
        height: 50,
        width: 50,
        upperRightX: 500,
        upperRightY: 150
    });

    filter.addWindow(options, onError);

    var options = PointerDetectorWindowMediaParam({
        id: 'window1',
        height: 50,
        width: 50,
        upperRightX: 500,
        upperRightY: 250
    });

    filter.addWindow(options, onError);

    filter.on ('WindowIn', function (data){
        console.log ("Event window in detected in window " + data.windowId);
    });

    filter.on ('WindowOut', function (data){
        console.log ("Event window out detected in window " + data.windowId);
    });

    console.log("Connecting ...");
    client.connect(webRtc, filter, webRtc, function(error) {
        if (error) return onError(error);

        console.log("WebRtcEndpoint --> Filter --> WebRtcEndpoint");
    });
});
};
};
};
};

```

The following picture illustrates the pointer tracking in one of the defined windows:

In order to carry out the calibration process, this JavaScript function is used:

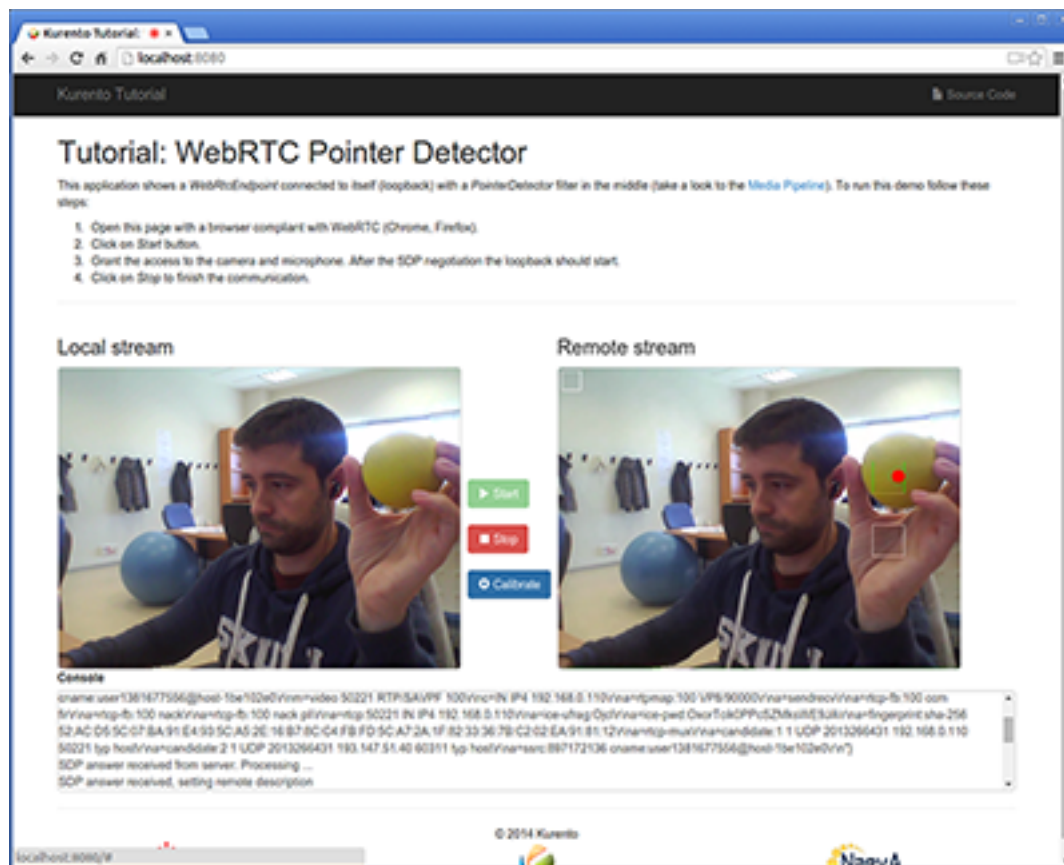


Fig. 66: *Pointer tracking over a window*

```
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"}]
```

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": "7.0.0",
  "kurento-utils": "7.0.0"
  "kurento-module-pointerdetector": "7.0.0"
}
```

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

```
bower install
```

Note: You can find the latest versions at *Bower*.

7.17.3 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.js 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 `kurento-module-pointerdetector` should be also installed:

```
sudo apt-get install kurento-module-pointerdetector
```

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

```
curl -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

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.git  
cd kurento/tutorials/javascript-node/pointerdetector/  
git checkout 7.0.0  
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
```

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:

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.

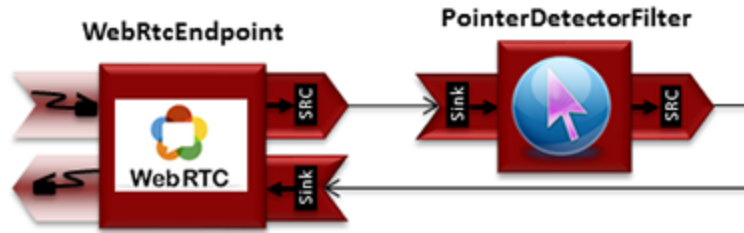


Fig. 67: WebRTC with PointerDetector filter in loopback Media Pipeline

In order to perform pointer detection, there must be a calibration stage, in which the color of the pointer is registered by the filter. To accomplish this step, the pointer should be placed in a square in the upper left corner of the video, as follows:

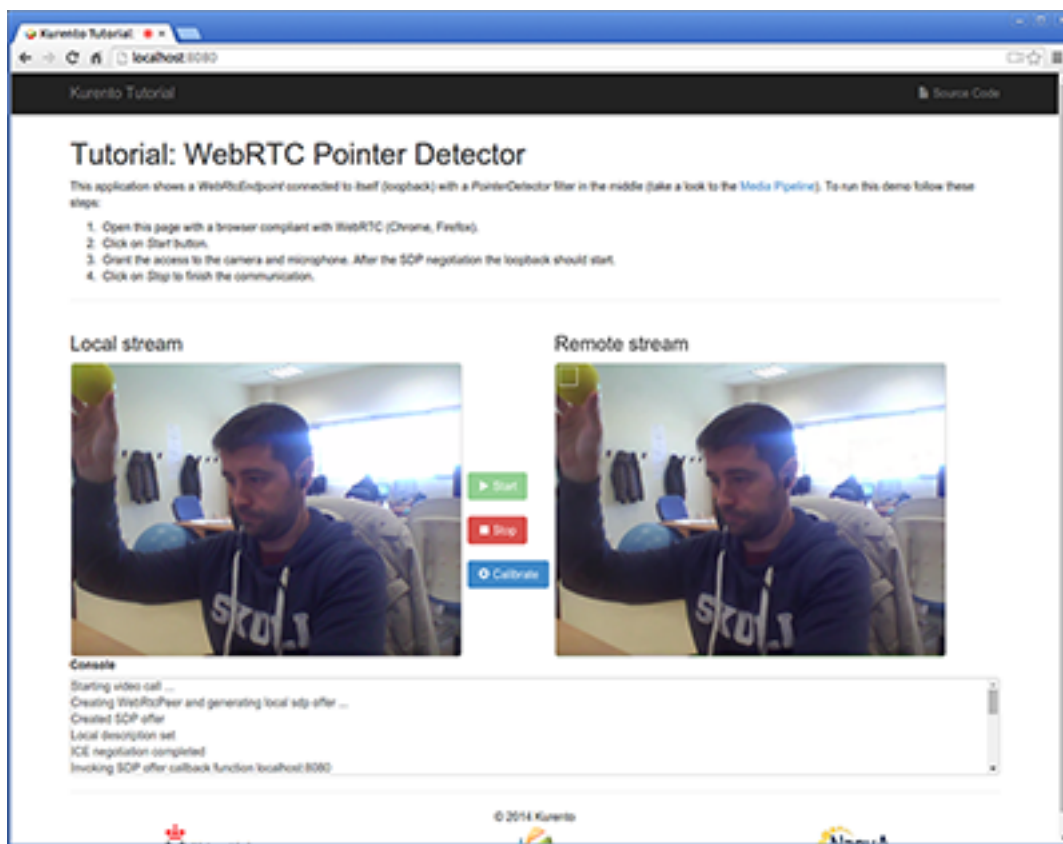


Fig. 68: Pointer calibration stage

Note: Modules can have options. For configuring these options, you'll need to get the constructor for them. In JavaScript and Node.js, you have to use `kurentoClient.getComplexType('qualifiedName')`. There is an example in the code.

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

After that, the color of the pointer is tracked in real time by Kurento Media Server. `PointerDetectorFilter` can also

define regions in the screen called *windows* in which some actions are performed when the pointer is detected when the pointer enters (WindowIn event) and exits (WindowOut event) the windows. This is implemented in the JavaScript logic as follows:

```
...
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('IceCandidateFound', function(event) {
            var candidate = kurento.getComplexType('IceCandidate')(event.
↪candidate);

            ws.send(JSON.stringify({
              id : 'iceCandidate',
              candidate : candidate
            }));
          });
        });
      });
    });
  });
}
```

(continues on next page)

(continued from previous page)

```

});

filter.on('WindowIn', function (_data) {
    return callback(null, 'WindowIn', _data);
});

filter.on('WindowOut', function (_data) {
    return callback(null, 'WindowOut', _data);
});

var options1 = PointerDetectorWindowMediaParam({
    id: 'window0',
    height: 50,
    width: 50,
    upperRightX: 500,
    upperRightY: 150
});
filter.addWindow(options1, function(error) {
    if (error) {
        pipeline.release();
        return callback(error);
    }
});

var options2 = PointerDetectorWindowMediaParam({
    id: 'window1',
    height: 50,
    width: 50,
    upperRightX: 500,
    upperRightY: 250
});
filter.addWindow(options2, function(error) {
    if (error) {
        pipeline.release();
        return callback(error);
    }
});

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

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

```

(continues on next page)

(continued from previous page)

```

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

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

        var options = {
            calibrationRegion: WindowParam({
                topRightCornerX: 5,
                topRightCornerY: 5,
                width: 30,
                height: 30
            })
        };

        pipeline.create('pointerdetector.PointerDetectorFilter', options, function(error,
→ filter) {
            if (error) {
                return callback(error);
            }

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

```

The following picture illustrates the pointer tracking in one of the defined windows:

In order to carry out the calibration process, this JavaScript function is used:

```

function calibrate() {
    if (webRtcPeer) {
        console.log("Calibrating...");
        var message = {
            id : 'calibrate'
        }
        sendMessage(message);
    }
}

```

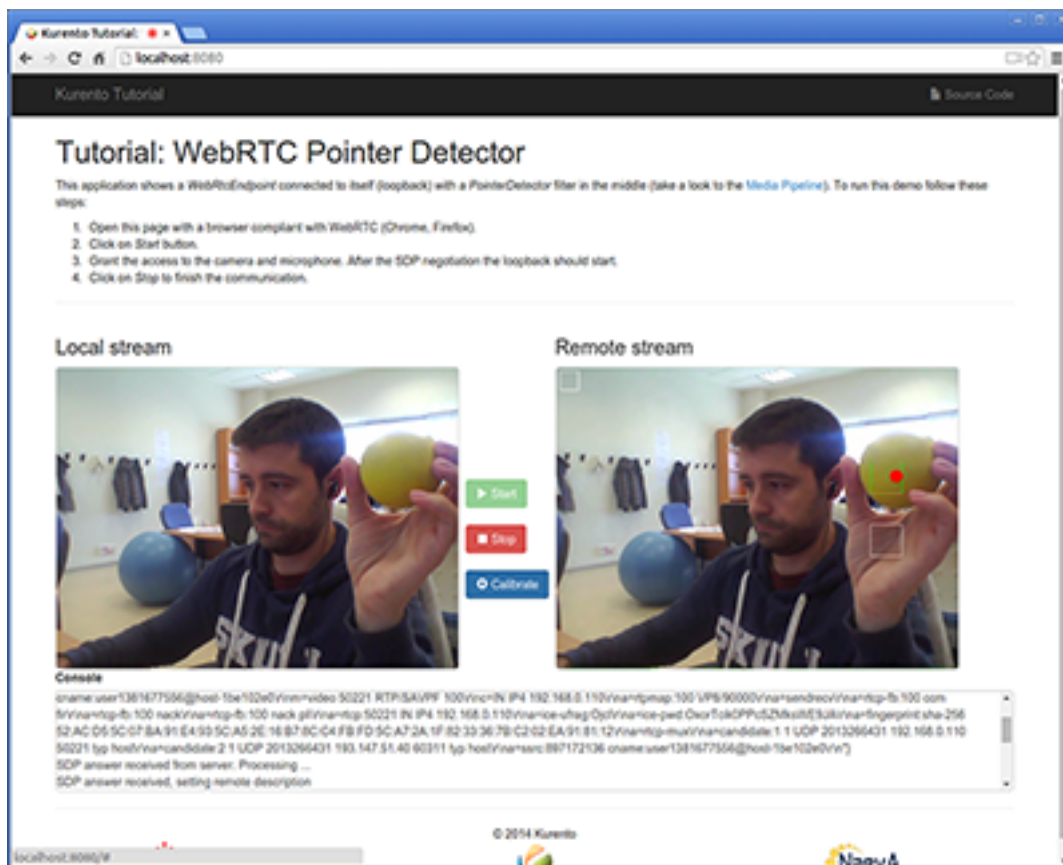


Fig. 69: Pointer tracking over a window

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" : "7.0.0"  
}
```

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" : "7.0.0",  
  "kurento-module-pointerdetector": "7.0.0"  
}
```

Note: You can find the latest versions at [npm](#) and [Bower](#).

WRITING KURENTO APPLICATIONS

Table of Contents

- *Writing Kurento Applications*
 - *Global Architecture*
 - *Application Architecture*
 - * *Communicating client, server and Kurento*
 - *1. Media negotiation phase (signaling)*
 - *2. Media exchange phase*
 - * *Real time WebRTC applications with Kurento*
 - *Media Plane*

8.1 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:

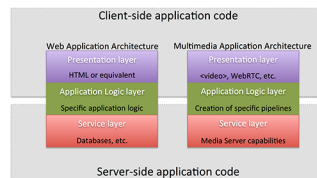


Fig. 1: *Layered architecture of web and multimedia applications. Applications created using Kurento (right) can be similar to standard Web applications (left). Both types of applications may choose to place the application logic at the client or at the server code.*

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**.

8.2 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:

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

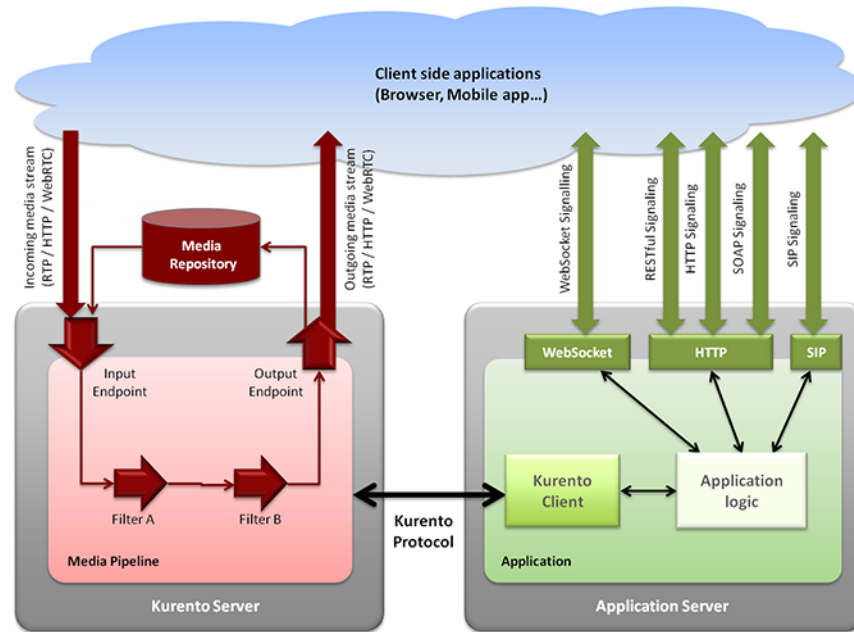


Fig. 2: Kurento Architecture. Kurento architecture follows the traditional separation between signaling and media planes.

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.

8.2.1 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:

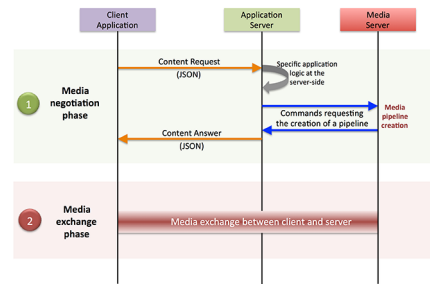


Fig. 3: Main interactions between architectural modules. These occur in two phases: negotiation and media exchange. Remark that the color of the different arrows and boxes is aligned with the architectural figures presented above. For example, orange arrows show exchanges belonging to the signaling plane, 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 response 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.

8.2.2 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:

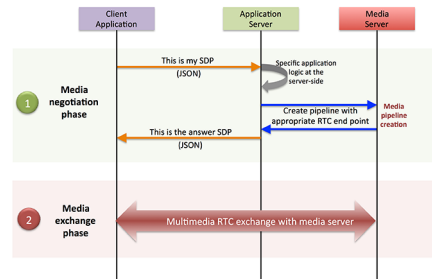


Fig. 4: Interactions in a WebRTC session. During the negotiation phase, an SDP Offer is sent to KMS, requesting the capabilities of the client. As a result, Kurento Media Server generates an SDP Answer that can be used by the client for establishing the media exchange.

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.

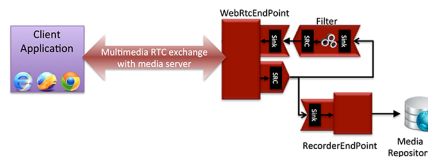


Fig. 5: Example pipeline for a WebRTC session. A *WebRtcEndpoint* is connected to a *RecorderEndpoint* storing the received media stream and to an *Augmented Reality* filter, which feeds its output media stream back to the client. As a result, the end user will receive its own image filtered (e.g. with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

8.3 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).

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:

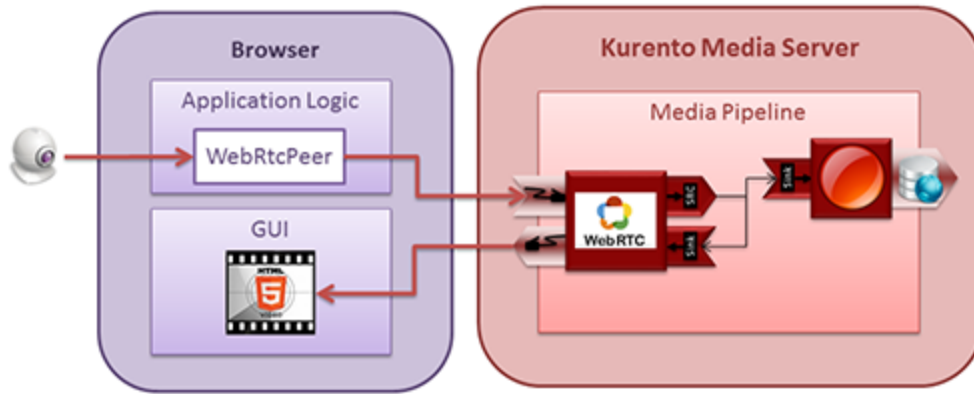


Fig. 6: Simple Example of a Media Pipeline

```
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

- *Writing Kurento Modules*
 - *Scaffolding and development*
 - * *OpenCV module*
 - * *GStreamer module*
 - * *For both kind of modules*
 - *Installation and usage*
 - * *Installing locally*
 - * *Installing in Docker*
 - * *Java client code*
 - * *JavaScript client code*
 - *Examples*

9.1 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 \
  kurento-media-server-dev
```

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

```
kurento-module-scaffold <Prefix> <CamelCaseName> <snake_case_name> [IsOpenCV? true|false]
```

<Prefix> should be an *UpperCamelCase* word that serves as an application- or library-specific namespace prefix in order to avoid name conflicts in case a similar plugin with the same name ever gets added to GStreamer. If an empty string ("") is given, then Gst will be used by default.

<CamelCaseName> and <snake_case_name> are the name of the new module, in *UpperCamelCase* (also known as *PascalCase*) and *snake_case*, respectively.

For example:

- For an OpenCV module:

```
kurento-module-scaffold App OpenCVModule opencv-module true
```

- For a GStreamer module:

```
kurento-module-scaffold App VideoModule video-module false
```

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.json), 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
```

If working with a GStreamer module, now you can verify that the new module is successfully loaded by GStreamer, with these commands:

```
# To check if the plugin is found and loaded:  
gst-inspect-1.0 --gst-plugin-path="$PWD/src/gst-plugins" | grep -i appvideomodule  
  
# To inspect all metadata exported by the plugin:  
gst-inspect-1.0 --gst-plugin-path="$PWD/src/gst-plugins" appvideomodule  
  
# To test the plugin directly with some video input  
# (this is just a sample for raw video; adapt as necessary!)  
gst-launch-1.0 --gst-plugin-path="$PWD/src/gst-plugins" \  
    uridecodebin uri='file:///path/to/video.mp4' ! videoconvert \  
    ! appvideomodule \  
    ! videoconvert ! autovideosink
```

Note that in this example *appvideomodule* is the GStreamer name of your module.

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

9.1.1 OpenCV module

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

```
{Name}Impl.cpp
{Name}Impl.hpp
{Name}OpenCVImpl.cpp
{Name}OpenCVImpl.hpp
```

The first two files contain the server-side implementation of the JSON-RPC API, and normally you won't need to modify them. The last two files will contain the logic of your module.

The file `{Name}OpenCVImpl.cpp` contains functions to deal with the methods and the parameters (you must implement the logic). Also, this file contains a class method called **process**. This function will be called with each new frame, so you must implement the logic of your filter in there.

9.1.2 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:

```
{prefix}{name}.cpp
{prefix}{name}.h
{name}.c
```

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

```
{Name}Impl.cpp
{Name}Impl.hpp
```

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

9.1.3 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*, *INTERFACE_LIB_EXTRA_INCLUDE_DIRS*, *INTERFACE_LIB_EXTRA_LIBRARIES*

These parameters allow to add additional source code to the static library. Files included in *INTERFACE_LIB_EXTRA_HEADERS* will be installed in the system as headers for this library. All the parameters accept a list as input.

- `SERVER_IMPL_LIB_EXTRA_SOURCES`, `SERVER_IMPL_LIB_EXTRA_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.
-

9.2 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 Kurento to know about the module, so clients can instantiate objects and types provided by it.

Warning: To avoid C++ issues with ABI compatibility (which are usually caused by mixing compiler versions) you should build your module on the same system that Kurento was built. For example, if you run Kurento on Ubuntu 20.04, you should compile your module also on Ubuntu 20.04.

Do not mix system versions. For example, do not build your module on Ubuntu 20.04, and then try to install it on Ubuntu 18.04.

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 SDK that gets auto-generated from the Kurento Module Descriptor files (*.kmd.json).

9.2.1 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 Kurento 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 \
    dpkg-dev

# Run dpkg-buildpackage to build Debian packages
dpkg-buildpackage -b -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 -l ../*.deb
../videomodule-dev_0.0.1~rc1_amd64.deb
../videomodule_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 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 Kurento is already installed:

```
sudo dpkg -i videomodule_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-media-server`, to instruct Kurento about where the plugin files have been copied:

```
KURENTO_MODULES_PATH="$KURENTO_MODULES_PATH:/path/to/module"
GST_PLUGIN_PATH="$GST_PLUGIN_PATH:/path/to/module"
```

Kurento 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 Kurento 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: 7.0.0
Found modules:
    'core' version 7.0.0
    'elements' version 7.0.0
    'filters' version 7.0.0
    'appvideomodule' version 0.0.1~0.gd61e201

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

9.2.2 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:7.0.0
COPY video-module_0.0.1~rc1_amd64.deb /
RUN dpkg -i /video-module_0.0.1~rc1_amd64.deb
```

Now build the new image:

```
$ docker build --tag kurento-with-video-module:7.0.0 .
Step 1/3 : FROM kurento/kurento-media-server:7.0.0
Step 2/3 : COPY video-module_0.0.1~rc1_amd64.deb /
Step 3/3 : RUN dpkg -i /video-module_0.0.1~rc1_amd64.deb
Successfully built d10d3b4a8202
Successfully tagged kurento-with-video-module:7.0.0
```

And verify your module is correctly loaded by Kurento:

```
$ docker run --rm kurento-with-video-module:7.0.0 --version
Kurento Media Server version: 7.0.0
Found modules:
    'core' version 7.0.0
    'elements' version 7.0.0
    'filters' version 7.0.0
    'appvideomodule' version 0.0.1~0.gd61e201

$ docker run --rm kurento-with-video-module:7.0.0 --list
Available factories:
```

(continues on next page)

(continued from previous page)

```
[...]
AppVideoModule
appvideomodule.AppVideoModule
```

9.2.3 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 either of these commands:

- `make java` (equivalent to `mvn clean package`) to build the Maven package.
- `make java_install` (equivalent to `mvn clean install`) to build the Maven package and install it into the local repository (typically located at `$HOME/.m2/`).

Finally, to actually use the module in your Maven project, you have to add the dependency to the `pom.xml` file:

```
<project>
...
<dependencies>
  <dependency>
    <groupId>org.kurento.module</groupId>
    <artifactId>{name}</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </dependency>
</dependencies>
...
</project>
```

Then you will be able to instantiate and use the new module in your Java code. For example, Kurento's [OpenCV plugin sample](#) is used like this:

```
import org.kurento.module.opencvpluginsample.OpenCVPluginSample;
[...]
final OpenCVPluginSample myFilter =
    new OpenCVPluginSample.Builder(pipeline).build();
myFilter.setFilterType(0);
[...]
myWebRtcEndpoint1.connect(myFilter);
myFilter.connect(myWebRtcEndpoint2);
```

The result is, as expected, that the OpenCV plugin sample applies a [Wikipedia: Canny edge detector](#) to the original image:

Remote stream



Fig. 1: Kurento's *OpenCV* plugin sample, applying a Canny edge detector

9.2.4 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.js*). To do that, you should add your JavaScript module as a dependency in your *bower.json* or *package.json* file, respectively:

```
"dependencies": {
  "{name}": "0.0.1"
}
```

9.3 Examples

Simple examples for both kinds of modules are available in the GitHub repo <https://github.com/Kurento/kurento>:

- `server/module-examples/gstreamer-example`
- `server/module-examples/opencv-example`

There are a lot of examples showing how to define methods, parameters or events in the “extra” modules that Kurento provides for demonstration purposes:

- `server/module-examples/pointerdetector/src/server/interface`
- `server/module-examples/crowddetector/src/server/interface`
- `server/module-examples/chroma/src/server/interface`
- `server/module-examples/platedetector/src/server/interface`

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

- `server/module-core`
- `server/module-elements`
- `server/module-filters`

FREQUENTLY ASKED QUESTIONS

Table of Contents

- *Frequently Asked Questions*
 - *NAT, ICE, STUN, TURN*
 - * *When are STUN and TURN needed?*
 - * *How does TURN work?*
 - * *How to install Coturn?*
 - * *How to test my STUN/TURN server?*
 - *Kurento in Docker*
 - * *How to change config files?*
 - *Bind mount*
 - *Docker volume*
 - *FROM image*
 - * *Where are my recordings?*
 - *Media Pipeline*
 - * *How many simultaneous participants are supported?*
 - * *How many Media Pipelines do I need for my Application?*
 - * *How many Endpoints do I need?*
 - * *Which participant corresponds to which Endpoint?*
 - * *How to get existing objects from the Media Server?*

10.1 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 a STUN/TURN server (Coturn):
 - [How to install Coturn?](#)
 - [How to test my STUN/TURN server?](#)
- Troubleshooting [WebRTC failures](#)
- Advanced knowledge: [NAT Types and NAT Traversal](#)

10.1.1 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.

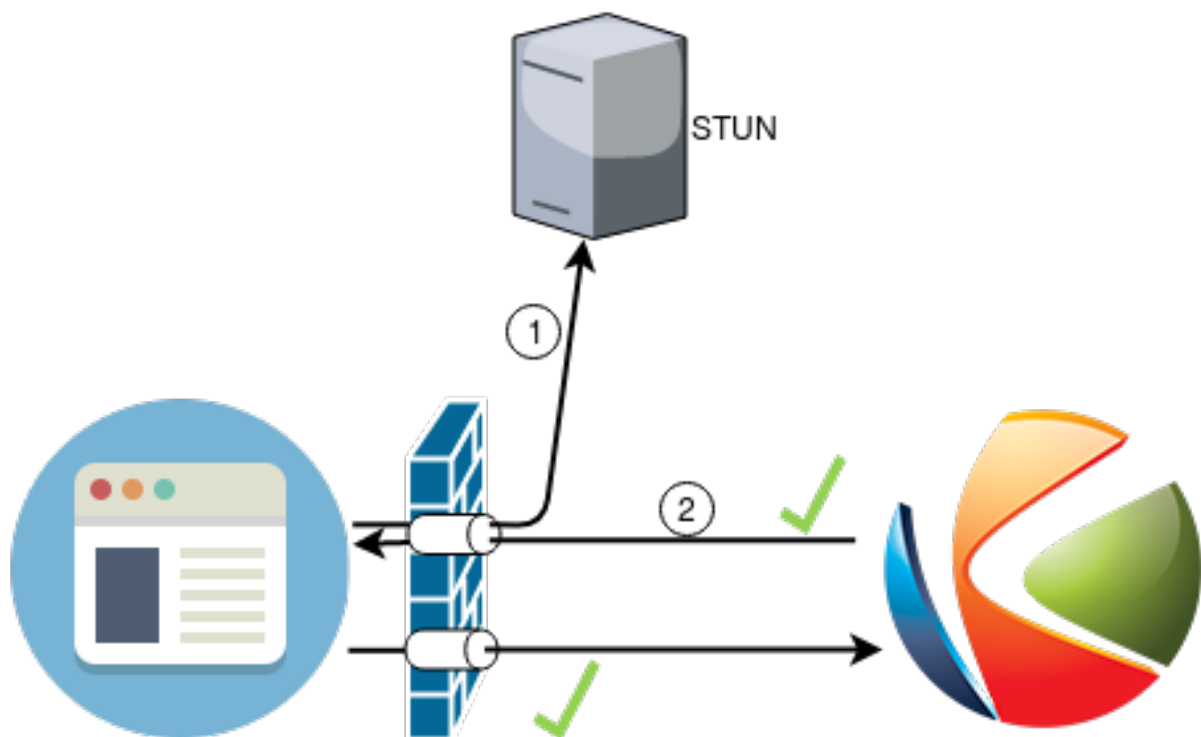
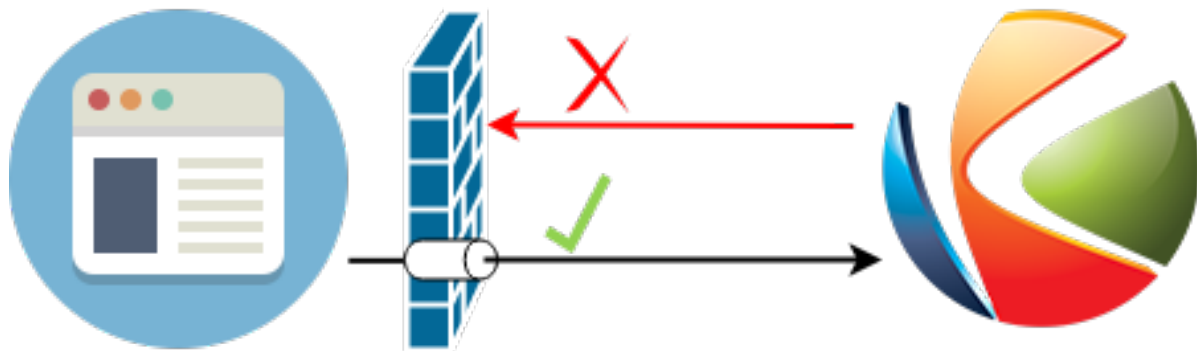
If Kurento is behind a NAT (e.g. if your media server is protected by a NAT firewall) you must provide it with your STUN or TURN server settings. Check [STUN/TURN Server Configuration](#) for ways to configure this in Kurento. Do the same also for any WebRTC clients (like web browsers, through the *iceServers* 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 Kurento.

In scenarios like this, the client is able to send data to Kurento because its NAT will allow outgoing packets. However, Kurento 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 Kurento:



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

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

Further reading:

- [Introduction to WebRTC protocols](#).
- [WebRTC - How many STUN/TURN servers do I need to specify?](#).
- [What are STUN, TURN, and ICE? \(archive\)](#).

10.1.2 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*.

10.1.3 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 \
coturn
```

To configure it for WebRTC, follow these steps:

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

This example configuration is a good baseline; it contains the minimum setup required for using Coturn with Kurento Media Server for WebRTC:

```
# 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>
```

(continues on next page)

(continued from previous page)

```
# STUN listener port for UDP and TCP.
# Default: 3478.
#listening-port=3478

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

# Uncomment to enable moderately verbose logs.
# Default: verbose mode OFF.
#verbose

# TURN fingerprints in messages.
fingerprint

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

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

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

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

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

Note:

- The *external-ip* is necessary in cloud providers that use internal NATs, such as AWS (Amazon EC2). Uncomment this line and write the machine's public IP address in the field *<CoturnIp>*. **It must be an IP address, not a domain name.**
- Comment out (or delete) all the TURN parameters if you only want Coturn acting as a STUN server.
- Create the destination log file, otherwise Coturn will not have permissions to create the file by itself:

```
sudo install -o turnserver -g turnserver -m 644 /dev/null /var/log/turn.log
```

- Other settings 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>

Warning: This example configures TURN authentication with the “*long-term credential*” method, where you write a static username and password in the fields `<TurnUser>` and `<TurnPassword>`.

While that is good enough for showcasing the Coturn setup here, for real-world scenarios you might want to use dynamically-generated passwords. This is more secure, because each individual participant can be provided with an exclusive one-time username and password.

Coturn can be integrated with external sources, such as PostgreSQL ([psql-userdb](#)), MySQL ([mysql-userdb](#)), MongoDB ([mongo-userdb](#)), or Redis ([redis-userdb](#)), and it even provides a [REST API](#) for time-limited credentials ([use-auth-secret](#)). You can handle any of these methods from your *Application Server*, then use the *Kurento API* to dynamically provide each individual WebRtcEndpoint with the correct parameters.

2. Edit the file `/etc/default/coturn` and uncomment or add this line:

```
TURN_SERVER_ENABLED=1
```

3. Start the Coturn system service:

```
sudo service coturn restart
```

4. The following ports should be open in your firewall / NAT / cloud provider’s *Security Group*:

- **listening-port** (default: 3478) UDP & TCP. You can skip opening one of the protocols if you disable either UDP or TCP in Coturn (for example, with `no-tcp`).
- All the range from **min-port** to **max-port** (default: 49152 to 65535). As per [RFC 8656](#), this is the port range that Coturn will use by default for TURN relay. Again, you can disable either of TCP or UDP (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.

5. Provide your STUN or TURN server settings to both Kurento Media Server and all WebRTC clients (like web browsers). Check [STUN/TURN Server Configuration](#) for ways to configure this in Kurento.
6. Check that your Coturn server is working. For that, follow the steps given in the next section.

10.1.4 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.

10.2 Kurento in Docker

Deploying Kurento Media Server in a container is the easiest install method, because it bundles all of the different modules and dependencies into a single, manageable unit. This makes installation and upgrades a trivial operation.

However, due to the nature of containers, it also makes configuration slightly more inconvenient, so in this section we'll provide a heads up in Docker concepts that could be very useful for users of [Kurento Docker images](#).

10.2.1 How to change config files?

To edit the configuration files used in your containers, first you'll need the actual files; run these commands to get default ones from a temporary container:

```
docker create --name temp kurento/kurento-media-server:7.0.0
docker cp temp:/etc/kurento/ ./kurento-files/
docker rm temp
```

After editing these files as needed, provide them to newly created containers. Next sections below show examples of how to do it.

Bind mount

A [bind-mount](#) will “inject” your files from the host machine into the Kurento container. This method is the simplest one to use if you are in control of the host system:

```
docker run -d --name kurento --network host \
  --mount type=bind,src="$PWD/kurento-files/",dst=/etc/kurento/ \
  kurento/kurento-media-server:7.0.0
```

The equivalent Docker Compose file would look like this:

```
version: "3.8"
services:
  kms:
    image: kurento/kurento-media-server:7.0.0
    network_mode: host
    volumes:
      - type: bind
        source: ./kurento-files/
        target: /etc/kurento/
```

Docker volume

A **volume** is a storage module that can be attached to containers. This has the benefit of not depending on the host filesystem, as everything is managed by Docker.

First, use an ephemeral container to create a new volume and populate it with your config files:

```
docker create --name temp \
  --mount type=volume,src=kurento-volume,dst=/etc/kurento/ \
  busybox
docker cp ./kurento-files/. temp:/etc/kurento/
docker rm temp
```

Then run your container as usual, mounting the volume in the appropriate path:

```
docker run -d --name kurento --network host \
  --mount type=volume,src=kurento-volume,dst=/etc/kurento/ \
  kurento/kurento-media-server:7.0.0
```

The equivalent Docker Compose file would look like this:

```
version: "3.8"
services:
  kms:
    image: kurento/kurento-media-server:7.0.0
    network_mode: host
    volumes:
      - type: volume
        source: kurento-volume
        target: /etc/kurento/
```

FROM image

Creating your own fully customized, self-contained image is a good choice to avoid that your containers depend on files stored in the host machine: The **FROM** feature of *Dockerfiles* can be used to derive directly from the official **Kurento Docker image**.

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

```
FROM kurento/kurento-media-server:7.0.0
COPY ./kurento-files /etc/kurento
```

Now, build the new image:

```
$ docker build --tag kurento-media-server-custom .
Step 1/2 : FROM kurento/kurento-media-server:7.0.0
Step 2/2 : COPY ./kurento-files /etc/kurento
Successfully built 3d2bedb31a9d
Successfully tagged kurento-media-server-custom
```

And use your new image “*kurento-media-server-custom*” in place of the original one.

10.2.2 Where are my recordings?

A frequent question, by users who are new to Docker, is where the *RecorderEndpoint* files are being stored, because they don't show up anywhere in the host file system. The answer is that KMS is recording files *inside the container's local storage*, in the path defined by the *RecorderEndpoint* constructor ([Java](#), [JavaScript](#)).

In general, running a Docker container **won't modify your host system** and **won't create new files** in it, at least by default. This is an integral part of how Docker containers work. To get those files out, you should use the mechanisms that Docker offers, like for example a [bind-mount](#) to the recording path.

10.3 Media Pipeline

These questions relate to the concept of *Media Pipeline* in Kurento, touching topics about architecture or performance.

10.3.1 How many simultaneous participants are supported?

This depends entirely on the performance of the machine where Kurento Media Server is running. The best thing you can do is performing an actual load test under your particular conditions.

The folks working on [OpenVidu](#) (a WebRTC platform based on Kurento) conducted a study that you might find interesting:

- [OpenVidu load testing](#): a systematic study of OpenVidu platform performance.

10.3.2 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. A *Media Element* can only communicate when they are part of the same Pipeline. Different Pipelines in the server are independent and isolated, 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.

10.3.3 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.

10.3.4 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.

10.3.5 How to get existing objects from the Media Server?

The usual workflow for an Application Server is to connect with the Media Server, and use RPC methods to *create* new MediaPipelines and Endpoints inside it. However, if you want to connect your Application Server with objects that *already exist* in the Media Server (as opposed to creating new ones), you can achieve it by querying by their ID. This is done with the “*describe*” method of the JSON-RPC API, as described in [Kurento Protocol](#).

Client API:

- Java: `KurentoClient.getById`.
- JavaScript: `KurentoClient.getMediaobjectById`.

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: **7.0.0**. 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:

Table of Contents

- *Troubleshooting Issues*
 - *Media Server Crashes*
 - *Corrupted Video*
 - * *About sender video encoding*
 - * *About H.264 & VP8 color encoding*
 - *Other Media Server issues*
 - * *Reached limit / Resource temporarily unavailable*
 - * *GStreamer-CRITICAL messages in the log*
 - * *CPU usage grows too high*
 - * *Memory usage grows too high*
 - * *Service init doesn't work*
 - * *OpenH264 not found*
 - * *Missing audio or video streams*
 - *Application Server*
 - * *KMS is not running*
 - * *KMS became unresponsive (due to network error or crash)*

- * *Node.js / NPM failures*
- * *Connection ends exactly after 60 seconds*
- * *“Expects at least 4 fields”*
- * *“Error: ‘operationParams’ is required”*
- *WebRTC failures*
 - * *ICE connection problems*
 - * *mDNS ICE candidate fails: Name or service not known*
- *Docker issues*
 - * *Publishing Docker ports eats memory*
 - * *Multicast fails in Docker*
- *Element-specific info*
 - * *PlayerEndpoint*
 - *RTSP broken audio*
 - *RTSP broken video*
 - *RTSP Video stuttering*
 - * *RecorderEndpoint*
 - *Zero-size video files*
 - *Smaller or low quality video files*
- *Browser*
 - * *Safari doesn’t work*

11.1 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: **7.0.0**.
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.

11.2 Corrupted Video

Problem

- Video image seems fine, but playback suffers from a lot of stuttering (i.e. it is not smooth, constantly “jumps” around). See here: [RTSP Video stuttering](#).
- Video playback is smooth (no color issues, no macroblocks, no excessive stuttering), but the perceived quality of the details is very poor.
- Video contains green or pink patches in some areas:
- Video contains huge blocks (aka. “*macroblocks*”) that are dragged around through the video:
- KMS logs contain lots of these messages (in bursts of several per second):

```
WARN rtpsource [...] duplicate or reordered packet (seqnr 32462, expected 32464)

WARN kmsutils [...] GAP of 3 ms at PTS=0:01:54.187106448 (packet loss?); will
↳request a new keyframe

WARN kmsutils [...] DISCONTINUITY at non-keyframe; will drop until keyframe
```

Reason

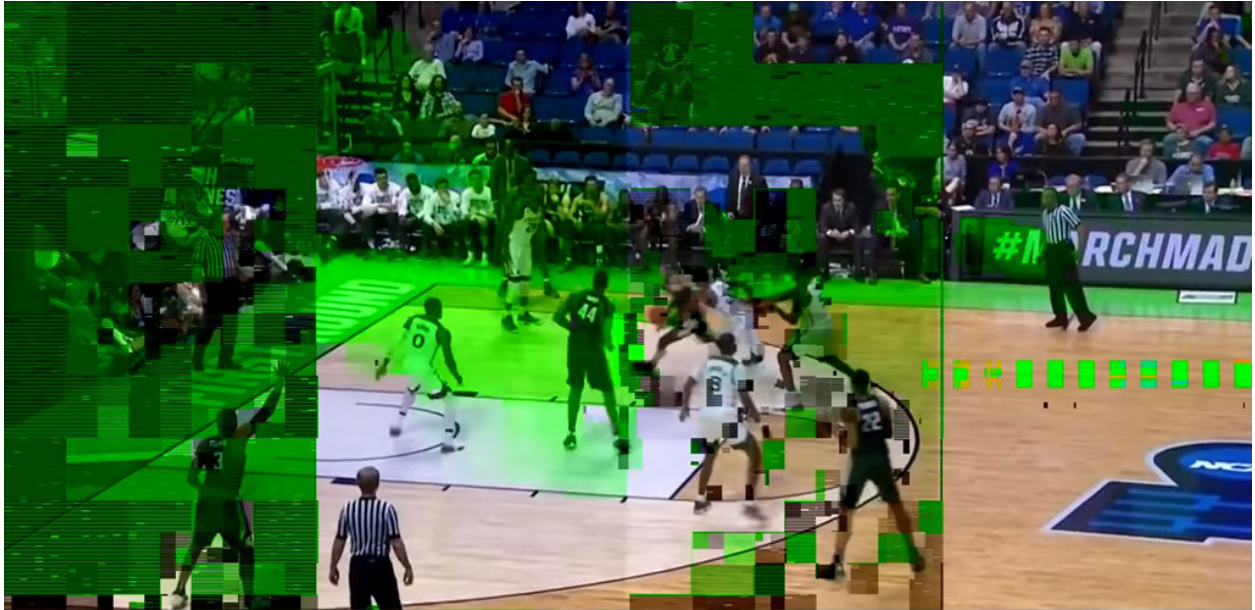


Fig. 1: *Green patches covering part of the picture.*



Fig. 2: *Macroblocks distorting the picture.*

- **Network congestion** or an otherwise weak network link is causing a high rate of **packet loss** and, in the case of WebRTC, an automatic degradation of video quality on the sender side. Most web browsers will automatically reduce their video output quality if they detect that the network is congested.
- Too much data is sent to Kurento's *PlayerEndpoint*, which is not able to process it all on time, causing it to drop parts of the video.
- A badly configured H.264 encoder in the sender side, especially when using a *PlayerEndpoint* to consume the video stream of an IP camera.

Solution

- For decoding errors (color issues, macroblocks) the most effective change you can do is to reduce the video resolution and/or quality (bitrate, framerate) at the sender. This will make the video smaller, helping it to travel through congested networks.
- Getting a stronger network link on both sender and receiver sides will always help. For example, moving closer to the Wifi access points, using Ethernet cables when possible, or moving to a better data coverage area.
- Configure your sender encoder with correct parameters. See the next section about video encoding.
- When the network link is not an issue, remember to change the default maximum bitrate of **500 Kbps** that Kurento uses to send WebRTC.

See also:

- *Configuring WebRTC bitrate.*
- WebRtcEndpoint API docs: [Java](#), [JavaScript](#).

11.2.1 About sender video encoding

The maximum bitrate for WebRTC video (used by web browsers such as Chrome) is **2 Mbps for perfect conditions**, so you should probably avoid pushing more than that in your application.

Regarding the video encoder at the sender side, keep in mind that the most compatible H.264 setting is the **Constrained Baseline Profile, Level 3.1**.

Lastly, note that Chrome not only adapts its own video sending bitrate according to network conditions, but also the resolution of the video. If you see a much lower resolution than expected, you should check the sender WebRTC stats, to see if it isn't because of Chrome deciding to do so.

See also:

- *Notes on browser video encoding.*

WebRTC will detect the bandwidth available on the network, and will adapt the video bitrate on the fly (and, in some cases such as the Chrome web browser, the resolution will change too). This adaptation is influenced by some properties of the network, such as **jitter**, **latency**, and **packet loss**. If your WebRTC video plays back smoothly but with a very poor quality, this mostly means that the network link itself is poor.

See also:

- *Congestion Control (RMCAT).*
- *REMB.*

11.2.2 About H.264 & VP8 color encoding

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*.

Kurento detects that packets have been lost in the network, and sends retransmission requests to the source of the WebRTC or RTP stream. However, if packet losses are too high due to a weak or congested network, enough losses will build up until the video decoding gets negatively affected.

Cisco has also 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.

11.3 Other Media Server issues

11.3.1 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”).

11.3.2 GStreamer-CRITICAL messages in the log

GLib and GStreamer use a lot of internal `assert()` calls, to catch bugs in their own source code. When an assertion fails, a warning message is printed to the logs and the program continues running. Not crashing is good, of course, but Kurento hitting bugs of an underlying library will cause problems, sooner or later.

So, it's in our best interest to watch out for these warnings. Report them to us if you see any in your logs! ;-)

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
```

These warnings don't really provide much debug information. To find out more, 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.

11.3.3 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:-2},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*").

11.3.4 Memory usage grows too high

Problem

Each new Session consumes some memory, but later the memory is not freed back to the system after the Kurento Session is closed.

Reason

The most common cause for increasingly growing memory usage is not a memory leak, but *Memory Fragmentation*.

Solution

Try using an alternative memory allocator to see if it solves the issue of memory fragmentation. Please have a look at *Using Jemalloc*.

If you still think there might be a memory leak in KMS, keep reading:

- Neither *top* nor *ps* are the right tool for the job to establish whether Kurento Media Server has a memory leak; **Valgrind** is.
- Tools like *top* or *ps* 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 marked as available for use by the same program, but not by others. So *top* or *ps* won't be able to "see" the memory after KMS frees it.

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 with an extra argument:

```
bin/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 KMS with this image might help finding memory-related issues.

11.3.5 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
```

11.3.6 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-1.0/libgstopenh264.so': libopenh264.so.0:
cannot open shared object file: No such file or directory
```

Also these conditions apply:

- Packages *openh264-gst-plugins-bad-1.0* 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 --reinstallopenh264
```

11.3.7 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*.

If your application is expected to work with **audio-only** or **video-only** streams, make sure that Kurento Pipeline elements are not connected with the default `connect(MediaElement)` method ([Java](#), [JavaScript](#)):

- Use the `connect(MediaElement, MediaType)` method ([Java](#), [JavaScript](#)).
- Monitor the *MediaFlowInStateChanged* and *MediaFlowOutStateChanged* events from all *MediaElements*.
- Make sure that the element providing media (the *source*) is firing a *MediaFlowOut* event, and that the receiver (the *sink*) is firing a corresponding *MediaFlowIn* event.

11.4 Application Server

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

11.4.1 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 [...] Starting Application on TEST with PID 16448
DEBUG [...] Executing getKmsUrlLoad(b843d6f6-02dd-49b4-96b6-f2fd2e8b1c8d) in KmsUrlLoader
DEBUG [...] Obtaining kmsUrl=ws://localhost:8888/kurento from config file or system
↳property
DEBUG [...] Connecting to kms in ws://localhost:8888/kurento
DEBUG [...] Creating JsonRPC NETTY WebSocket client
DEBUG [...] Enabling heartbeat with an interval of 240000 ms
DEBUG [...] [KurentoClient] Connecting websocket client to server ws://localhost:8888/
↳kurento
WARN [...] [KurentoClient] Error sending heartbeat to server. Exception:
↳[KurentoClient] Exception connecting to WebSocket server ws://localhost:8888/kurento
WARN [...] [KurentoClient] Stopping heartbeat and closing client: failure during
↳heartbeat mechanism
DEBUG [...] [KurentoClient] Connecting websocket client to server ws://localhost:8888/
↳kurento
```

(continues on next page)

(continued from previous page)

```

DEBUG [...] Sending error to all pending requests
WARN [...] [KurentoClient] Trying to close a JsonRpcClientNettyWebSocket with channel.
↳ == null
WARN [...] 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 [...] 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 [...] Starting Application on TEST with PID 21617
DEBUG [...] Executing getKmsUrlLoad(af479feb-dc49-4a45-8b1c-eedf8325c482) in KmsUrlLoader
DEBUG [...] Obtaining kmsUrl=ws://localhost:8888/kurento from config file or system.
↳ property
DEBUG [...] Connecting to kms in ws://localhost:8888/kurento
DEBUG [...] Creating JsonRPC NETTY WebSocket client
DEBUG [...] Enabling heartbeat with an interval of 240000 ms
DEBUG [...] [KurentoClient] Connecting websocket client to server ws://localhost:8888/
↳ kurento
INFO [...] [KurentoClient] Connecting native client
INFO [...] [KurentoClient] Creating new NioEventLoopGroup
INFO [...] [KurentoClient] Initiating new Netty channel. Will create new handler too!
DEBUG [...] [KurentoClient] channel active
DEBUG [...] [KurentoClient] WebSocket Client connected!
INFO [...] Started Application in 1.841 seconds (JVM running for 4.547)

```

11.4.2 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 to reconnect and continue working normally:

```

INFO [...] Started Application in 1.841 seconds (JVM running for 4.547)

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

INFO [...] [KurentoClient] channel closed
DEBUG [...] [KurentoClient] JsonRpcWsClient disconnected from ws://localhost:8888/
↳ kurento because Channel closed.
DEBUG [...] Disabling heartbeat. Interrupt if running is false
DEBUG [...] [KurentoClient] JsonRpcWsClient reconnecting to ws://localhost:8888/kurento.
DEBUG [...] [KurentoClient] Connecting websocket client to server ws://localhost:8888/
↳ kurento
INFO [...] [KurentoClient] Connecting native client

```

(continues on next page)

(continued from previous page)

```

INFO [...] [KurentoClient] Closing previously existing channel when connecting native.
↳ client
DEBUG [...] [KurentoClient] Closing client
INFO [...] [KurentoClient] Initiating new Netty channel. Will create new handler too!
WARN [...] [KurentoClient] Trying to close a JsonRpcClientNettyWebSocket with channel.
↳ == null
DEBUG [...] tryReconnectingForever = true
DEBUG [...] tryReconnectingMaxTime = 0
DEBUG [...] maxTimeReconnecting = 9223372036854775807
DEBUG [...] currentTime = 1510773733903
DEBUG [...] Stop connection retries: false
WARN [...] [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 [...] [KurentoClient] JsonRpcWsClient reconnecting to ws://localhost:8888/kurento.
DEBUG [...] [KurentoClient] Connecting websocket client to server ws://localhost:8888/
↳ kurento
INFO [...] [KurentoClient] Connecting native client
INFO [...] [KurentoClient] Creating new NioEventLoopGroup
INFO [...] [KurentoClient] Initiating new Netty channel. Will create new handler too!
DEBUG [...] [KurentoClient] channel active
DEBUG [...] [KurentoClient] WebSocket Client connected!
DEBUG [...] [KurentoClient] Req-> {"id":2,"method":"connect","jsonrpc":"2.0"}
DEBUG [...] [KurentoClient] <-Res {"id":2,"result":{"serverId":"1a3b4912-9f2e-45da-87d3-
↳ 430fef44720f","sessionId":"f2fd16b7-07f6-44bd-960b-dd1eb84d9952"},"jsonrpc":"2.0"}
DEBUG [...] [KurentoClient] Reconnected to the same session in server ws://
↳ localhost:8888/kurento

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

```

11.4.3 Node.js / NPM failures

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

11.4.4 Connection ends exactly after 60 seconds

This is typically caused by an intermediate proxy, which is prematurely ending the WebSocket session from the Application Server, and thus making the media server believe that all resources should be released.

For example, if **Nginx Reverse Proxy** is used, the default value of `proxy_read_timeout` is **60 seconds**, but the default Kurento *Ping/Pong keep-alive* mechanism works in intervals of 240 seconds.

This issue can also manifest itself with this (misleading) error message in the browser’s JavaScript console:

```
WebRTC: ICE failed, add a TURN server and see about:webrtc for more details
```

The solution is to increase the timeout value in your proxy settings.

11.4.5 “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-1.0* 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" }
]
```

11.4.6 “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.js application wanting to use the *ImageOverlayFilter* API (Java, JavaScript) 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:␣
↪ 400001, 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);
```

11.5 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: [STUN/TURN Server](#).
- 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. See also: [Connection ends exactly after 60 seconds](#).

- Have a look at these articles about troubleshooting WebRTC:
 - [Troubleshooting WebRTC Connection Issues \(archive\)](#).
 - [Common \(beginner\) mistakes in WebRTC \(archive\)](#).

11.5.1 ICE connection problems

If your application receives an *IceComponentStateChanged* 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.

11.5.2 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 libnss-mdns
sudo apt-get update ; sudo apt-get install 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".

11.6 Docker issues

11.6.1 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)

11.6.2 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 [...] 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:7.0.0
    container_name: kms
    restart: always
    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>

11.7 Element-specific info

11.7.1 PlayerEndpoint

RTSP broken audio

If you have your own RTSP tool generating OPUS encoded audio to be consumed in Kurento with a *PlayerEndpoint* (Java, JavaScript), and the resulting audio is very choppy and robotic, you should start by verifying that your encoding process is configured correctly for the OPUS frame size used in WebRTC.

This was the case for a user who later shared with us the reasons for the bad quality audio they were perceiving:

Bad audio quality

> *There was a mismatch between the incoming raw audio frame size and the opus encoding frame size, which resulted in a bad encoding cadence causing irregular encoded frame intervals.*

> We remedied this by ensuring that the incoming audio frame size and the opus encoding frame size are the same — or the incoming frame size is a divisor of the encoding frame size.

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**. However, these are outlier cases and normally a higher `networkCache` is needed.

In principle, `networkCache = 0` would mean that all RTP packets must be exactly on point at the expected times in the RTSP stream, or else they will be dropped. So even a slight amount of jitter or delay in the network might cause packets to be dropped when they arrive to the `PlayerEndpoint`.

`networkCache` translates directly to the `latency` property of GStreamer's `rtspsrc` element, which in turn is passed to the `rtpbin` and ultimately the `rtpjitterbuffer` inside it.

RTSP Video stuttering

Problem

`PlayerEndpoint` is used to consume an RTSP stream from some source (typically, an IP camera). However, the resulting video (e.g. after recording with `RecorderEndpoint`, or after relaying video to WebRTC viewers with `WebRtcEndpoint`) shows stuttering (i.e. the video playback is not smooth, it constantly “jumps” around).

Reason

The source video is too heavy and KMS is not able to process it on time, so it lags behind and ends up losing parts of it.

Solution

The most effective change you can do is to reduce the video resolution and/or quality (bitrate, framerate) at the sender.

Kurento Media Server is known to work well receiving videos of up to **720p** resolution (1280x720) at **30fps** and bitrate 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 buffers filling up and frames getting dropped. Try reducing the resolution of your input videos to see if this helps solving the issue.

See also:

- [Corrupted Video](#).

Background

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, showing up as video stuttering on the output.

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:-2},rtppjitterbuffer:5"
/usr/bin/kurento-media-server 2>&1 | grep -P 'rtppjitterbuffer.*(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.

11.7.2 RecorderEndpoint

Zero-size video files

Remember that the [client documentation](#) contains lots of important information about usage of the RecorderEndpoint.

Follow this checklist to make sure none of these are the cause of your issue:

- The RecorderEndpoint was created with a `mediaProfile` type that assumes *both* audio and video. If you intend to record audio-only or video-only media, select the appropriate `_AUDIO_ONLY` or `_VIDEO_ONLY` profile when creating the recorder instance. For example, to record a WebRTC screen capture (as obtained from a web browser's call to `MediaDevices.getDisplayMedia()`), choose `WEBM_VIDEO_ONLY` instead of just `WEBM`.
- The RecorderEndpoint was connected with the default `connect(MediaElement)` method (Java, JavaScript) (which assumes both audio and video), but the stream is audio-only or video-only.
 - Monitor the *MediaFlowInStateChanged* and *MediaFlowOutStateChanged* events from all `MediaElements`.
 - Make sure that the element providing media (the *source*) is firing a *MediaFlowOut* event, and that the RecorderEndpoint is firing a corresponding *MediaFlowIn* event.
 - If your recording should be only-audio or only-video, use the `connect(MediaElement, MediaType)` method (Java, JavaScript).
- Check the 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.

Smaller or low quality video files

Kurento will just record whatever arrives as input, so if your recordings have less quality or lower resolution than expected, this is because the source video was already sent like that.

In most situations, the real cause of this issue is the web browser encoding and sending a low bitrate or a low resolution video. Keep in mind that some browsers (Chrome, as of this writing) are able to dynamically adjust the output resolution; this means that the real size of the video coming out from Chrome will vary over time. Normally it starts small, and after some time it improves, when the browser detects that the available network bandwidth allows for it.

Check this section to get some advice about how to investigate low quality issues: [Corrupted Video](#).

11.8 Browser

11.8.1 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: **7.0.0**. 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](#).

12.1 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?

12.1.1 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](#).

12.1.2 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.

12.2 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 kurento@openvidu.io and let us know about your project!

CLIENT API REFERENCE

Currently, the Kurento project provides implementations of the *Kurento Protocol* 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.

13.1 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. 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*.

- **Reference:** [Kurento Client JavaDoc](#).

13.2 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.

- **Reference:** [Kurento Client JsDoc](#).

13.3 Kurento Js Utils

kurento-utils-js ([browser/kurento-utils-js/](#)) is a browser library that can be used to simplify creation and handling of [RTCPeerConnection](#) objects, to control the browser's [WebRTC API](#).

Warning: This library is not actively maintained. It was written to simplify the *Kurento Tutorials* and has several shortcomings for more advanced uses.

For real-world applications we recommend to **avoid using this library** and instead to write your JavaScript code directly against the browser's WebRTC API.

- **Reference:** [kurento-utils-js JsDoc](#).

KURENTO MODULES

Table of Contents

- *Kurento Modules*
 - *Media Elements and Media Pipelines*
 - *Endpoints*
 - *Filters*
 - *Hubs*
 - *Example Modules*

Kurento Media Server is controlled through the API it exposes, so application developers can use high level languages to interact with it. The Kurento project already provides SDK implementations of this API for several platforms: [Client API Reference](#).

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.

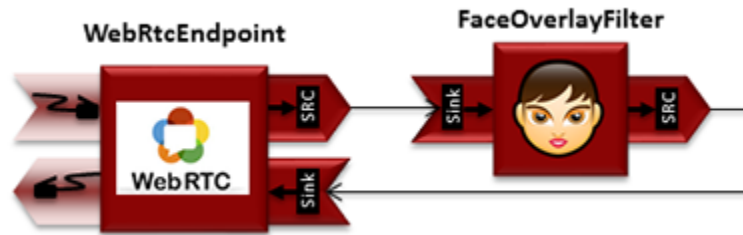


Fig. 1: Example of a Media Pipeline implementing an interactive multimedia application receiving media from a *WebRtcEndpoint*, overlaying an image on the detected faces and sending back the resulting stream

The Kurento API is **Object-Oriented**. 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 part of the main classes in the Kurento API:

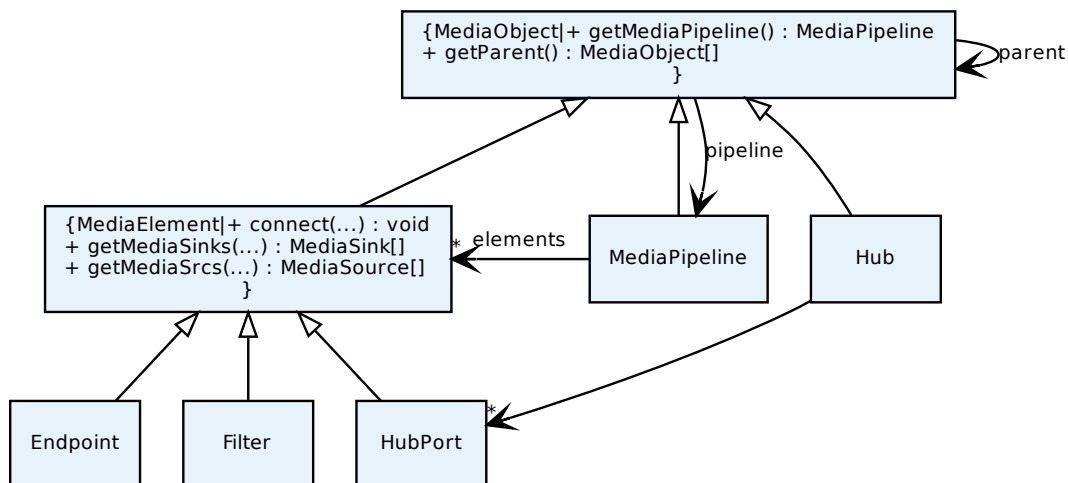


Fig. 2: Class diagram of main classes in Kurento API

14.2 Endpoints

WebRtcEndpoint: Input/output endpoint that provides media streaming for Real Time Communications (RTC) through the web. It implements *WebRTC* technology to communicate with browsers.



RtpEndpoint: Input/output endpoint that provides bidirectional content delivery capabilities with remote networked peers, through the *RTP* protocol. It uses *SDP* for media negotiation.



HttpPostEndpoint: Input endpoint that accepts media using HTTP POST requests like HTTP file upload function.



PlayerEndpoint: Input endpoint that retrieves content from file system, HTTP URL or RTSP URL and injects it into the Media Pipeline.



RecorderEndpoint: 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 main endpoint classes:

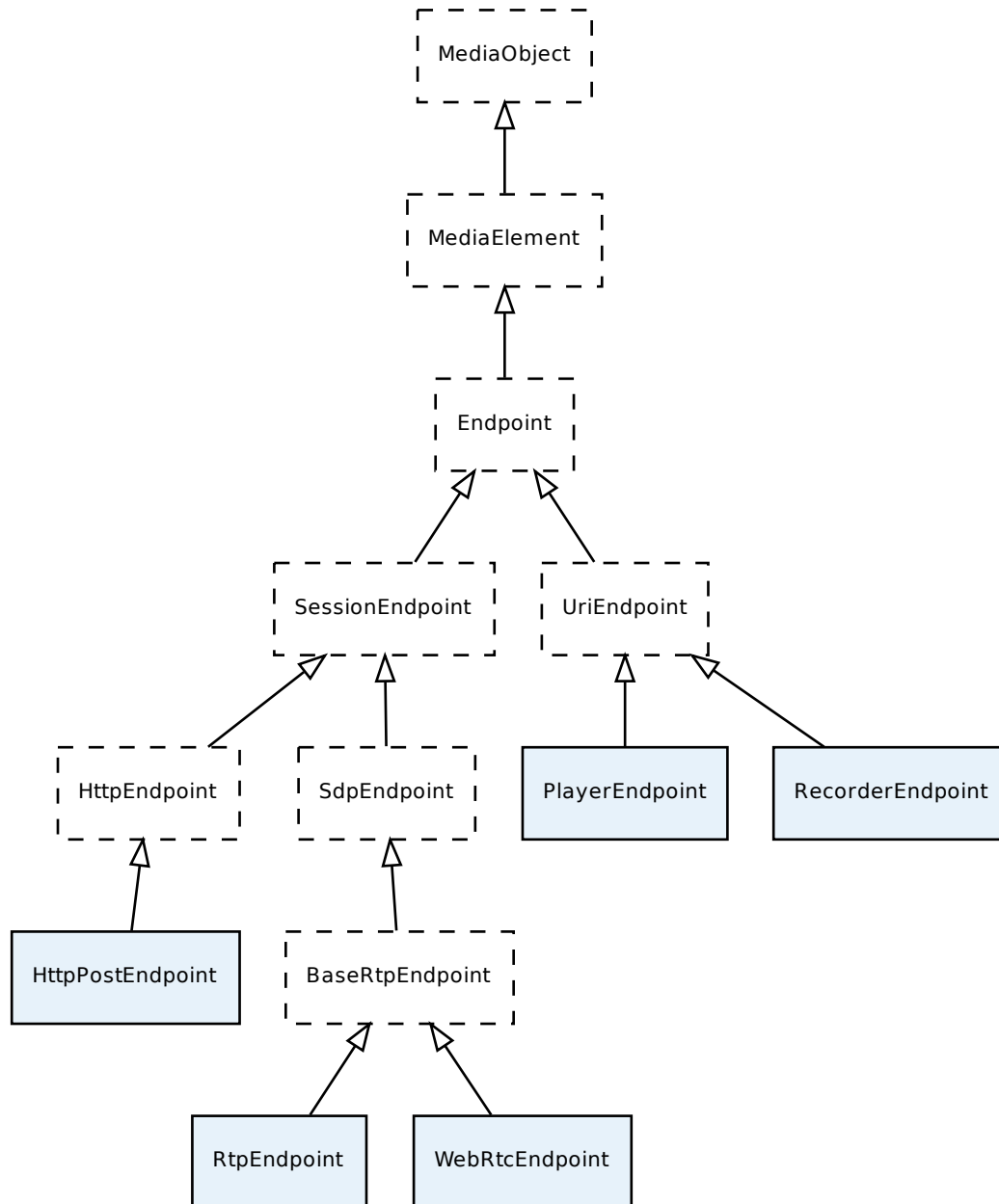


Fig. 3: Class diagram of Kurento Endpoints. In blue, the classes that a final API client will actually use.

14.3 Filters

Filters are MediaElements that perform media processing, Computer Vision, Augmented Reality, and so on.

ZBarFilter: Detects QR and bar codes in a video stream. When a code is found, the filter raises a *CodeFoundEvent*. Clients can add a listener to this event to execute some action.



FaceOverlayFilter: Detects faces in a video stream and overlays them with a configurable image.



GStreamerFilter: 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.



Usage of some popular GStreamer elements requires installation of additional packages. For example, overlay elements such as *timeoverlay* or *textoverlay* require installation of the **gststreamer1.0-x** package, which will also install the *Pango* rendering library.

The following class diagram shows the main filter classes:

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: Mixes the audio stream of its connected inputs and constructs a grid with the video streams of them.



DispatcherOneToMany: Sends a given input to all the connected output HubPorts.

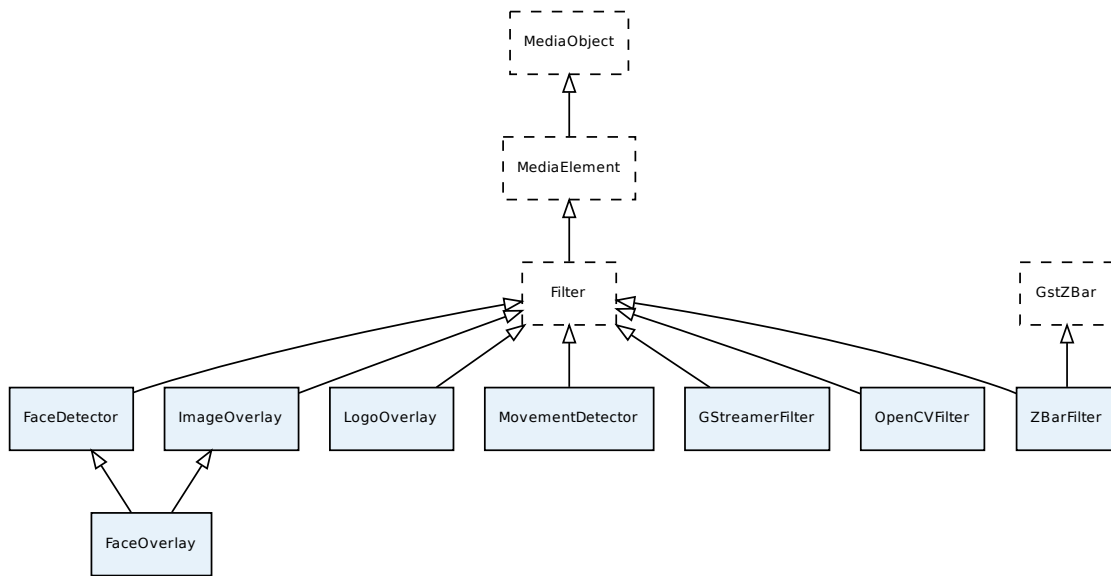


Fig. 4: Class diagram of Kurento Filters. In blue, the classes that a final API client will actually use.



Dispatcher: Routes between arbitrary input-output HubPort pairs.



The following class diagram shows the Hub classes:

14.5 Example Modules

In addition to the base features, there are some additional example modules provided **for demonstration purposes**:

These example modules are provided to show how to extend the base features of Kurento Media Server:

- **Chroma:** Takes a color range from the top-left area of the video, and makes it transparent, revealing another background image.
- **CrowdDetector:** Detects groups of people in video streams.
- **PlateDetector:** Detects vehicle license plates in video streams.

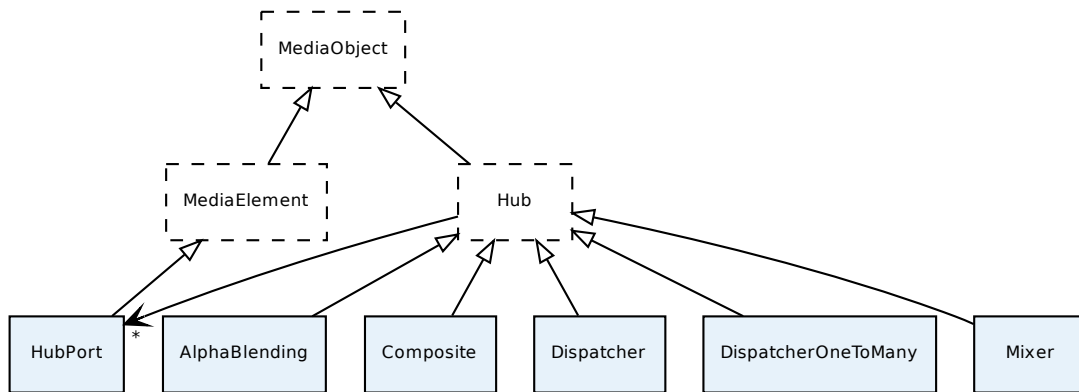


Fig. 5: Class diagram of Kurento Hubs. In blue, the classes that a final API client will actually use.

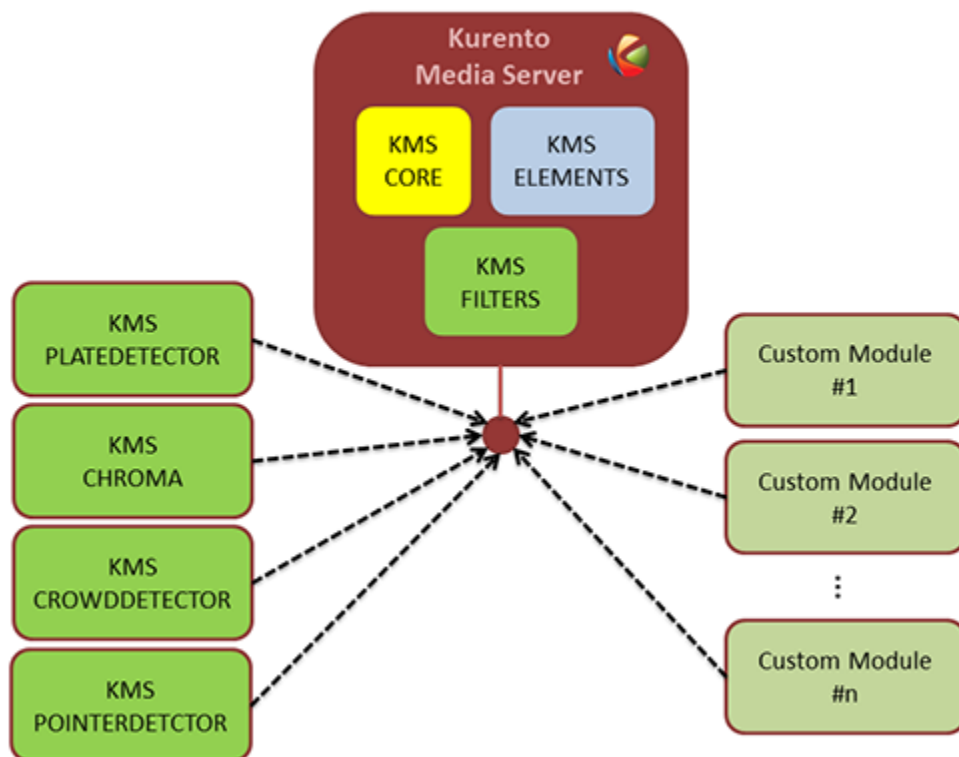


Fig. 6: **Kurento modules architecture** Kurento Media Server can be extended with example modules (chroma, crowd-detector, platedetector, pointerdetector) and also with other custom modules.

- **PointerDetector**: Detects pointers in video streams, based on color tracking.

Warning: These example modules **are just prototypes** and their results are not necessarily accurate or reliable. You can use them as programming guideline, but we strongly discourage anyone from using them in production environments.

All example modules come already preinstalled in the Kurento Docker images. For local installations, they can be installed separately with `apt-get`.

Taking into account these extra modules, the complete Kurento toolbox is extended as follows:

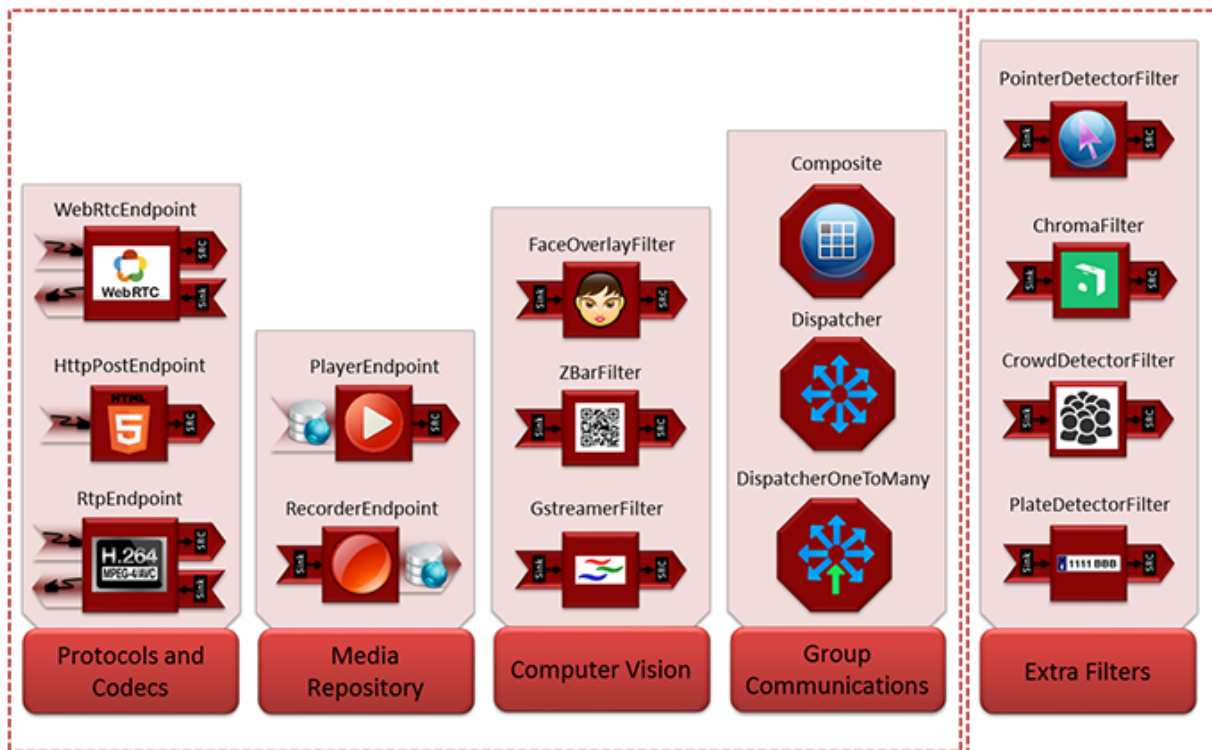


Fig. 7: **Extended Kurento Toolbox** The basic Kurento toolbox (left side of the picture) is extended with more Computer Vision and Augmented Reality filters (right side of the picture) provided by the example modules.

If you want to write your own modules, please read the section about [Writing Kurento Modules](#).

KURENTO PROTOCOL

Table of Contents

- *Kurento Protocol*
 - *JSON-RPC message format*
 - * *Request*
 - * *Successful Response*
 - * *Error Response*
 - *Kurento API over JSON-RPC*
 - * *ping*
 - * *create*
 - * *describe*
 - * *invoke*
 - * *release*
 - * *subscribe*
 - * *unsubscribe*
 - * *onEvent*
 - * *closeSession*
 - *Network issues*
 - *Example: WebRTC in loopback*
 - *Creating a custom Kurento Client*
 - * *Kurento Module Creator*

Kurento Media Server is controlled by means of an JSON-RPC API, implemented in terms of the **Kurento Protocol** specification as described in this document, based on *WebSocket* and *JSON-RPC*.

15.1 JSON-RPC message format

Kurento Protocol uses the *JSON-RPC* 2.0 Specification to encode its API messages. The following subsections describe the contents of the *JSON* messages that follow this spec.

15.1.1 Request

An *RPC call* is represented by sending a *request* message to a server. The *request* message has the following members:

- **jsonrpc**: A string specifying the version of the JSON-RPC protocol. It must be 2.0.
- **id**: A unique identifier established by the client that contains a string or number. The server must reply with the same value in the *response* message. This member is used to correlate the context between both messages.
- **method**: A string containing the name of the method to be invoked.
- **params**: A structured value that holds the parameter values to be used during the invocation of the method.

The following JSON shows a sample request for the creation of a *PlayerEndpoint* Media Element:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "create",
  "params": {
    "type": "PlayerEndpoint",
    "constructorParams": {
      "pipeline": "6829986",
      "uri": "http://host/app/video.mp4"
    },
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  },
}
```

15.1.2 Successful Response

When an *RPC call* is made, the server replies with a *response* message. In case of a successful response, the *response* message will contain the following members:

- **jsonrpc**: A string specifying the version of the JSON-RPC protocol. It must be 2.0.
- **id**: Must match the value of the *id* member in the *request* message.
- **result**: Its value is determined by the method invoked on the server.
- In case the connection is rejected, the response includes a message with a *rejected* attribute containing a message with a *code* and *message* attributes with the reason why the session was not accepted, and no *sessionId* is defined.

The following example shows a typical successful response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "value": "442352747",
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  },
}
```

(continues on next page)

(continued from previous page)

```
}
}
```

15.1.3 Error Response

When an *RPC call* is made, the server replies with a *response* message. In case of an error response, the *response* message will contain the following members:

- **jsonrpc**: A string specifying the version of the JSON-RPC protocol. It must be `2.0`.
- **id**: Must match the value of the *id* member in the *request* message. If there was an error in detecting the *id* in the *request* message (e.g. *Parse Error/Invalid Request*), *id* is *null*.
- **error**: A message describing the error through the following members:
 - **code**: An integer number that indicates the error type that occurred.
 - **message**: A string providing a short description of the error.
 - **data**: A primitive or structured value that contains additional information about the error. It may be omitted. The value of this member is defined by the server.

The following example shows a typical error response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": 33,
    "message": "Invalid parameter format"
  }
}
```

15.2 Kurento API over JSON-RPC

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**: Creates a new media object, i.e. a Media Pipeline, an Endpoint, or any other Media Element.
- **describe**: Retrieves an already existing object.
- **invoke**: Calls a method on an existing 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**: Marks a media object for garbage collection and release of the resources used by it.

The Kurento Protocol allows that Kurento Media Server sends requests to clients:

- **onEvent**: This request is sent from Kurento Media server to subscribed clients when an event occurs.

15.2.1 ping

In order to warrant the WebSocket connectivity between the client and the Kurento Media Server, a *keep-alive* method is implemented. This method is based on a *ping* method sent by the client, which must be replied with a *pong* message from the server. If no response is obtained in a time interval, the client will assume that the connectivity with the media server has been lost. The *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"
  }
}
```

15.2.2 create

This message requests the creation of an object from the Kurento API (Media Pipelines and Media Elements). The parameter *type* specifies the type of the object to be created. The parameter *constructorParams* contains all the information needed to create the object. Each message needs different *constructorParams* to create the object. These parameters are defined *per-module*.

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",
```

(continues on next page)

(continued from previous page)

```

    "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"
  },
  "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"
  }
}

```

15.2.3 describe

This message retrieves the information of an already existing object in the Media Server. This can be useful for cases where a newly started Application Server does already know the IDs of all objects it wants to manage, so it just needs to get a reference to them from the Media Server, instead of creating new ones. The *object* parameter contains the ID of the desired object that should be retrieved.

This example shows how to get a reference to a Media Pipeline that had been created earlier:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "describe",
  "params": {
    "object": "55c16267-2395-40af-af50-8555adc78f9c_kurento.MediaPipeline",
    "sessionId": "0cd20d0e-451f-4fd9-b3d4-dff33f90d328"
  }
}
```

The response to this request message is as follows:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "hierarchy": ["kurento.MediaObject"],
    "qualifiedType": "kurento.MediaPipeline",
    "sessionId": "0cd20d0e-451f-4fd9-b3d4-dff33f90d328",
    "type": "MediaPipeline"
  }
}
```

The response message contains the type information of the object that has just been retrieved. Other fields such as *id* and *sessionId* are those corresponding to the current RPC session.

The following example shows the retrieval of an already existing *PlayerEndpoint*; the mechanics are mostly the same, but in this case the response contains more details pertaining the class hierarchy of the Endpoint:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "describe",
  "params": {
    "object": "e9cbc8c2-d283-4e62-bb13-d34546d5cdf8_kurento.MediaPipeline/3a2abe27-6f9e-
↪4e08-9ac6-3a456b7979e7_kurento.PlayerEndpoint",
    "sessionId": "4b3c8344-5b47-4f40-bc2d-a2a0f82723d0"
  }
}
```

The response to this request message is as follows:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
```

(continues on next page)

(continued from previous page)

```

    "hierarchy": [
        "kurento.UriEndpoint",
        "kurento.Endpoint",
        "kurento.MediaElement",
        "kurento.MediaObject"
    ],
    "qualifiedType": "kurento.PlayerEndpoint",
    "sessionId": "4b3c8344-5b47-4f40-bc2d-a2a0f82723d0",
    "type": "PlayerEndpoint"
}

```

Lastly, this is what happens when trying to retrieve an object that does not exist in the server:

```

{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "describe",
  "params": {
    "object": "1234567890",
    "sessionId": "20587cfe-76aa-4451-ac73-55e33ae6ca2a"
  }
}

```

An error response will be returned:

```

{
  "jsonrpc": "2.0",
  "id": 5,
  "error": {
    "code": 40101,
    "data": { "type": "MEDIA_OBJECT_NOT_FOUND" },
    "message": "Object '1234567890' not found"
  }
}

```

15.2.4 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": 6,
  "method": "invoke",
  "params": {
    "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/76dcb8d7-5655-
    ↪ 445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",

```

(continues on next page)

(continued from previous page)

```
"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": 6,
  "result": {
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

15.2.5 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": 7,
  "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": 7,
  "result": {
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```


15.2.6 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": 8,
  "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": 8,
  "result": {
    "value": "052061c1-0d87-4fbd-9cc9-66b57c3e1280",
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

15.2.7 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": 9,
  "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": 9,
  "result": {
    "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
  }
}
```

15.2.8 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 data field contents are dependent on the type of event, but it generally contains these values:

- **source**: The object source of the event.
- **type**: The type of the event.
- **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.

This message has no *id* field due to the fact that no response is required.

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": [],
        "timestampMillis": "1441277150433",
        "type": "EndOfStream"
      },
      "object": "681f1bc8-2d13-4189-a82a-2e2b92248a21_kurento.MediaPipeline/e983997e-
↪ac19-4f4b-9575-3709af8c01be_kurento.PlayerEndpoint",
      "type": "EndOfStream"
    }
  }
}
```

Here, an example Error event is sent to notify about permission errors while trying to access the file system:

```
{
  "jsonrpc": "2.0",
  "method": "onEvent",
  "params": {
    "value": {
      "data": {
        "description": "Error code 6: Could not open file \"/tmp/invalid/path/test.webm\"
↪ for writing., [...] system error: Permission denied",
        "errorCode": 6,
        "source": "bdd15b54-9cfa-4036-8a1a-a17db06b78bc_kurento.MediaPipeline/5dd21f63-
↪ 643f-4562-a5d5-0ea0b6fd4a48_kurento.RecorderEndpoint",
        "tags": [],
        "timestampMillis": "1646657831138",
        "type": "RESOURCE_ERROR_OPEN"
      },
      "object": "bdd15b54-9cfa-4036-8a1a-a17db06b78bc_kurento.MediaPipeline/5dd21f63-
↪ 643f-4562-a5d5-0ea0b6fd4a48_kurento.RecorderEndpoint",
      "type": "Error"
    }
  }
}
```

For more info about Kurento events, check [Endpoint Events](#).

15.2.9 closeSession

If you inspect the JSON traffic between any of the Kurento clients and Kurento Media Server itself, you might notice that clients send a `closeSession` request. This is an undocumented command that was added for development purposes in the past, and was kept in the implementation. However, it does nothing in practice. You can safely ignore this method if you are implementing the Kurento Protocol on your own SDK.

15.3 Network issues

Resources handled by KMS are high-consuming. For this reason, KMS implements a garbage collector.

A Media Element is collected when the client is disconnected longer than 4 minutes. After that time, these media elements are disposed automatically. Therefore, the WebSocket connection between client and KMS should be active at all times. In case of 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": 10,
  "method": "connect",
  "params": {
    "sessionId": "4f5255d5-5695-4e1c-aa2b-722e82db5260"
  }
}
```

If KMS replies as follows ...

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "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": 10,
  "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": 10,
  "method": "connect"
}
```

15.4 Example: WebRTC in loopback

This section describes an example of the messages exchanged between a Kurento Client and the Kurento Media Server, in order to create a WebRTC in loopback. This example is fully depicted in the [Tutorials section](#). The steps are the following:

1. Client sends a request message in order to create a Media Pipeline:

```
{
  "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:

```
{
  "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*:

```
{
  "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:

```
{
  "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:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "invoke",
  "params": {
    "object": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-48ff-99ef-02f7fadabaf7_WebRtcEndpoint",
    "operation": "connect",
    "operationParams": {
      "sink": "c4a84b47-1acd-4930-9f6d-008c10782dfe_MediaPipeline/e72a1ff5-e416-48ff-99ef-02f7fadabaf7_WebRtcEndpoint"
    },
    "sessionId": "ba4be2a1-2b09-444e-a368-f81825a6168c"
  }
}
```

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

```
{
  "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:

```
{
  "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:

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

15.5 Creating a custom Kurento Client

In order to implement a Kurento Client you need to follow the reference documentation. The best way to know all details is to take a look 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

15.5.1 Kurento Module Creator

Kurento Clients contain code that is automatically generated from the IDL interface files, using a tool named **Kurento Module Creator**. This tool can 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 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 UTILS JS

Warning: This library is not actively maintained. It was written to simplify the [Kurento Tutorials](#) and has several shortcomings for more advanced uses.

For real-world applications we recommend to **avoid using this library** and instead to write your JavaScript code directly against the browser's [WebRTC API](#).

Table of Contents

- *Kurento Utils JS*
 - *Overview*
 - *How to use it*
 - *Examples*
 - *Using data channels*
 - *Reference documentation*
 - * *WebRtcPeer*
 - *MediaConstraints*
 - *Methods*
 - *getPeerConnection*
 - *showLocalVideo*
 - *getLocalStream*
 - *getRemoteStream*
 - *getCurrentFrame*
 - *processAnswer*
 - *processOffer*
 - *dispose*
 - *addIceCandidate*
 - *getLocalSessionDescriptor*
 - *getRemoteSessionDescriptor*

- *generateOffer*
- * *How to do screen share*
- *Source code*
- *Build for browser*

16.1 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](#).

16.2 How to use it

- **Minified file** - Download the file from [here](#).
- **NPM** - Install and use library in your Node.js 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>
```

16.3 Examples

There are several tutorials that show kurento-utils used in complete WebRTC applications developed on Java, Node.js and JavaScript. These tutorials are in GitHub, and you can download and run them at any time.

- **Java** - <https://github.com/Kurento/kurento/tutorials/java>
- **Node.js** - <https://github.com/Kurento/kurento/tutorials/javascript-node>
- **Browser JavaScript** - <https://github.com/Kurento/kurento/tutorials/javascript-browser>

In the following lines we will show how to use the library to create an `RTCPeerConnection`, and how to negotiate the connection with another peer. The library offers a `WebRtcPeer` object, which is a wrapper of the browser's `RTCPeerConnection` API. Peer connections can be of different types: unidirectional (send or receive only) or bidirectional (send and receive). The following code shows how to create the latter, in order to be able to send and receive media (audio and video). The code assumes that there are two video tags in the page that loads the script. These tags will be used to show the video as captured by your own client browser, and the media received from the other peer. The constructor receives a property that holds all the information needed for the configuration.

```

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

var constraints = {
  audio: true,
  video: {
    width: 640,
    framerate: 15
  }
};

var options = {
  localVideo: videoInput,
  remoteVideo: videoOutput,
  onIceCandidate : onIceCandidate,
  mediaConstraints: constraints
};

var webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options, function(error) {
  if(error) return onError(error)

  this.generateOffer(onOffer)
});

```

With this little code, the library takes care of creating the *RTCPeerConnection*, and invoking *getUserMedia* in the browser if needed. The constraints in the property are used in the invocation, and in this case both microphone and webcam will be used. However, this does not create the connection. This is only achieved after completing the SDP negotiation between peers. This process implies exchanging SDPs offer and answer and, since *Trickle ICE* is used, a number of candidates describing the capabilities of each peer. How the negotiation works is out of the scope of this document. More info can be found in [this link](#).

In the previous piece of code, when the *webRtcPeer* object gets created, the SDP offer is generated with *this.generateOffer(onOffer)*. The only argument passed is a function, that will be invoked 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.

```

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

  // We've made this function up sendOfferToRemotePeer(sdpOffer,
  function(sdpAnswer) {
    webRtcPeer.processAnswer(sdpAnswer);
  });
}

```

As we've commented before, the library assumes the use of *Trickle ICE* to complete the connection between both peers. In the configuration of the *webRtcPeer*, there is a reference to a *onIceCandidate* callback function. The library will use this function to send ICE candidates to the remote peer. Since this is particular to each application, we will just show

the signature

```
function onIceCandidate(candidate) {  
    // Send the candidate to the remote peer  
}
```

In turn, our client application must be able to receive ICE candidates from the remote peer. Assuming the signaling takes care of receiving those candidates, it is enough to invoke the following method in the *webRtcPeer* to consider the ICE candidate

```
webRtcPeer.addIceCandidate(candidate);
```

Following the previous steps, we have:

- Sent 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.

16.4 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** from the wrapped **RTCPeerConnection** object. This allows you to inject into and consume data from the pipeline. This data can be treated by each endpoint differently. For instance, a *WebRtcPeer* object in the browser, will have the same behavior as the *RTCDataChannel* (you can see a description [here](#)). Other endpoints could make use of this channel to send information: a filter that detects QR codes in a video stream, could send the detected code to the clients through a data channel. This special behavior should be specified in the filter.

The use of data channels in the *WebRtcPeer* object is indicated by passing the *dataChannels* flag in the options bag, along with the desired options.

```
var options = {  
    localVideo : videoInput,  
    remoteVideo : videoOutput,  
    dataChannels : true,  
    dataChannelConfig: {  
        id : getChannelName(),  
        onmessage : onMessage,  
        onopen : onOpen,  
        onclose : onClosed,  
        onbufferedamountlow : onbufferedamountlow,  
        onerror : onerror  
    },  
    onicecandidate : onIceCandidate  
}  
  
webRtcPeer = new kurentoUtils.WebRtcPeer.WebRtcPeerSendrecv(options,   
↪onWebRtcPeerCreated);
```

The values in *dataChannelConfig* are all optional. Once the *webRtcPeer* object is created, and after the connection has been successfully negotiated, users can send data through the data channel

```
webRtcPeer.send('your data stream here');
```

The format of the data you are sending, is determined by your application, and the definition of the endpoints that you are using.

The lifecycle of the underlying *RTCDataChannel*, is tied to that of the *webRtcPeer*: when the *webRtcPeer.dispose()* method is invoked, the data channel will be closed and released too.

16.5 Reference documentation

16.5.1 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 invoked when an error in the data channel is produced. If none is provided, an error trace message will be logged in the browser console.
- *simulcast*: Indicates whether simulcast is going to be used. Value is *true|false*
- *configuration*: It is a JSON object where ICE Servers are defined using
 - * *iceServers*: The format for this variable is like:

```
[{"urls": "turn:turn.example.org", "username": "user", "credential": "myPassword"}]
[{"urls": "stun:stun1.example.net"}, {"urls": "stun:stun2.example.net"}]
```

- **callback**: It is a callback function which indicate, if all worked right or not

Also there are 3 specific methods for creating `WebRtcPeer` objects without using *mode* parameter:

- **WebRtcPeerRecvonly(options, callback)**: Create a `WebRtcPeer` as receive only.
- **WebRtcPeerSendonly(options, callback)**: Create a `WebRtcPeer` as send only.
- **WebRtcPeerSendrecv(options, callback)**: Create a `WebRtcPeer` as send and receive.

MediaConstraints

Constraints provide a general control surface that allows applications to both select an appropriate source for a track and, once selected, to influence how a source operates. `getUserMedia()` uses constraints to help select an 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.

16.5.2 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.

16.6 Souce code

The code is at [github](#).

Be sure to have *Node.js* and *Bower* installed in your system:

```
curl -sSL https://deb.nodesource.com/setup_18.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.git
cd kurento/browser/kurento-utils-js/
npm install
```

16.7 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:

```
git clone https://github.com/Kurento/kurento.git
cd kurento/browser/kurento-utils-js/
npm install
node_modules/.bin/grunt
```


SECURING KURENTO APPLICATIONS

Table of Contents

- *Securing Kurento Applications*
 - *Securing Application Servers*
 - * *Configure a Java server to use HTTPS*
 - * *Configure a Node.js server to use HTTPS*
 - * *Configure JavaScript applications to use HTTPS*
 - *Securing Kurento Media Server*
 - * *Signaling Plane authorization*
 - * *Signaling Plane security (WebSocket)*
 - *Connecting to Secure WebSocket*
 - * *Media Plane security (DTLS)*

17.1 Securing Application Servers

17.1.1 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: *Self-Signed Certificates*.
- 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.

17.1.2 Configure a Node.js 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: *Self-Signed Certificates*.
- Add the following changes to your *server.js*, in order to enable HTTPS:

```
...
var express = require('express');
var ws      = require('ws');
var fs      = require('fs');
var https   = require('https');
...

var options =
{
  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
```

17.1.3 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 -sSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install 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: [Self-Signed Certificates](#).
- Start the HTTPS web server, using the SSL certificate:

```
http-server -p 8443 --ssl --cert cert.pem --key key.pem
```

17.2 Securing Kurento Media Server

17.2.1 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 [Client API Reference](#) to access features of [Kurento Modules](#).
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.

17.2.2 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 Protocol*.

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 uncomment 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: *Self-Signed Certificates*. 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.js applications.** Take a look at this page: [Painless Self Signed Certificates in node.js \(archive\)](#).

For a faster but *INSECURE* alternative, configure Node.js 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.js 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.js:** 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";
```

17.2.3 Media Plane security (DTLS)

WebRTC uses [DTLS](#) for media data authentication. By default, if no certificate is provided for this, Kurento Media Server will auto-generate a new self-signed certificate for every `WebRtcEndpoint` instance.

Alternatively, an already existing certificate can be provided to be used for all endpoints. For this, edit the file `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini` and set either `pemCertificateRSA` or `pemCertificateECDSA` with a file containing both your certificate (chain) file(s) and the private key. You can generate such file with the `cat` command:

```
# Make a single file to be used with Kurento Media Server.
cat cert.pem key.pem >cert+key.pem
```

Then, *configure* the path to `cert+key.pem`.

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.

ENDPOINT EVENTS

This is a list of all events that can be emitted by an instance of *WebRtcEndpoint*. This class belongs to a chain of inherited classes, so this list includes events from all of them, starting from the topmost class in the inheritance tree:

Table of Contents

- *Endpoint Events*
 - *MediaObject events*
 - * *Error*
 - *MediaElement events*
 - * *ElementConnected*
 - * *ElementDisconnected*
 - * *MediaFlowInStateChanged*
 - * *MediaFlowOutStateChanged*
 - * *MediaTranscodingStateChanged*
 - *BaseRtpEndpoint events*
 - * *ConnectionStateChanged*
 - * *MediaStateChanged*
 - *WebRtcEndpoint events*
 - * *DataChannelClosed*
 - * *DataChannelOpened*
 - * *IceCandidateFound*
 - * *IceComponentStateChanged*
 - * *IceGatheringDone*
 - * *NewCandidatePairSelected*
 - *Sample sequence of events: WebRtcEndpoint*

18.1 MediaObject events

This is the base interface used to manage capabilities common to all Kurento elements, including both *MediaElement* and *MediaPipeline*.

18.1.1 Error

Kurento Client API docs: [Java](#), [JavaScript](#).

Some error has occurred on a Kurento *MediaObject* instance. Check the event parameters (such as *description*, *errorCode*, and *type*) to get information about what happened.

The `ErrorEvent` can be emitted from any child of the Kurento (see [Kurento Modules](#) for more details). This Event has a **type** string field that contains an error identifier. **Applications should subscribe to the Error event from all Kurento objects**, this way you'll know when something goes wrong.

Only a handful of errors have their type encoded into a well defined string:

- "RESOURCE_ERROR_OPEN": Indicates that there was a problem when trying to open a local file or resource. This will typically happen when, for example, the *PlayerEndpoint* tries to open a file for which it does not have read permissions from the filesystem, or when *RecorderEndpoint* doesn't have write permissions for the destination path.
- "RESOURCE_ERROR_WRITE": Indicates an storage error while a write operation was ongoing. This error might be seen if the disk fails while *RecorderEndpoint* is writing an output file.
- "RESOURCE_ERROR_NO_SPACE_LEFT": This error will mostly happen when the disk becomes full while a *RecorderEndpoint* is doing its job. This is a common thing to happen if you don't have free space monitoring on your servers, so you should listen for this error from the *RecorderEndpoint* if you use it in any of your applications.
- "STREAM_ERROR_DECODE": This error tends to happen when the sending side has transmitted an invalid encoded stream, and Kurento Media Server is trying to decode it but the underlying GStreamer library is unable to do so. This could happen, for example, when using a *PlayerEndpoint* (which by default decodes the input stream), or when *Transcoding* has been enabled due to incompatible codecs negotiated by different *WebRtcEndpoints*. When getting this error, you should review the settings of the sender, because there might be something wrong with its encoder configuration.
- "STREAM_ERROR_FAILED": A generic error that is originated from the underlying GStreamer library when any data flow issue occurs. KMS debug logs should be checked because chances are that more descriptive information has been printed in there.
- "UNEXPECTED_ELEMENT_ERROR": Unclassified error.

On top of this, some errors are actually not being handled by the *MediaObject* where they occurred, so they will end up in the error handler of *MediaPipeline*, with the *type* field set to "UNEXPECTED_PIPELINE_ERROR".

18.2 MediaElement events

These events indicate some low level information about the state of `GStreamer`, the underlying multimedia framework.

18.2.1 ElementConnected

[TODO - add contents]

18.2.2 ElementDisconnected

[TODO - add contents]

18.2.3 MediaFlowInStateChanged

- 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.

18.2.4 MediaFlowOutStateChanged

- 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.

18.2.5 MediaTranscodingStateChanged

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.

18.3 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.

18.3.1 ConnectionStateChanged

- State = *Connected*: All of the *KmsIRtpConnection* objects have been created [TODO: explain what this means].
- State = *Disconnected*: At least one of the *KmsIRtpConnection* objects is not created yet.

Call sequence:

```
signal KmsIRtpConnection::"connected"
-> signal KmsSdpSession::"connection-state-changed"
-> signal KmsBaseRtpEndpoint::"connection-state-changed"
-> BaseRtpEndpointImpl::updateConnectionState
```

18.3.2 MediaStateChanged

- State = *Connected*: At least *one* of the audio or video RTP streams in the session is still alive (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. *MediaFlowOutStateChanged* (State: *Flowing*).
-

18.4 WebRtcEndpoint events

These events provide information about the state of `libnice`, the underlying library in charge of the ICE Gathering process. The ICE Gathering is typically done before attempting any WebRTC call.

For further reference, see the `libnice`'s [Agent documentation](#) and [source code](#).

18.4.1 DataChannelClosed

[TODO - add contents]

18.4.2 DataChannelOpened

[TODO - add contents]

18.4.3 IceCandidateFound

A new local candidate has been found, after the ICE Gathering process was started. Equivalent to the signal `NiceAgent::"new-candidate-full"`.

18.4.4 IceComponentStateChanged

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.

18.4.5 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"`.

18.4.6 NewCandidatePairSelected

During the connectivity checks one of the pairs happened to provide a successful connection, and the pair had a higher preference than the previously selected one (or there was no previously selected pair yet). Equivalent to the signal `NiceAgent::"new-selected-pair"`.

18.5 Sample sequence of events: WebRtcEndpoint

Once an instance of *WebRtcEndpoint* is created inside a Media Pipeline, an event handler should be added for each one of the events that can be emitted by the endpoint. Later, the endpoint should be instructed to do one of either:

- Generate an SDP Offer, when KMS is the caller. Later, the remote peer will generate an SDP Answer as a reply, which must be provided to the endpoint.
- Process an SDP Offer generated by the remote peer, when KMS is the callee. This will in turn generate an SDP Answer, which should be provided to the remote peer.

As a last step, the *WebRtcEndpoint* should be instructed to start the ICE Gathering process.

You can see a working example of this in [Kurento Java Tutorial - Hello World](#). This example code shows the typical usage for the *WebRtcEndpoint*:

```
KurentoClient kurento;
MediaPipeline pipeline = kurento.createMediaPipeline();
WebRtcEndpoint webRtcEp = new WebRtcEndpoint.Builder(pipeline).build();
webRtcEp.addIceCandidateFoundListener(...);
webRtcEp.addIceComponentStateChangedListener(...);
webRtcEp.addIceGatheringDoneListener(...);
webRtcEp.addNewCandidatePairSelectedListener(...);

// Receive an SDP Offer, via the application's custom signaling mechanism
String sdpOffer = recvMessage();
```

(continues on next page)

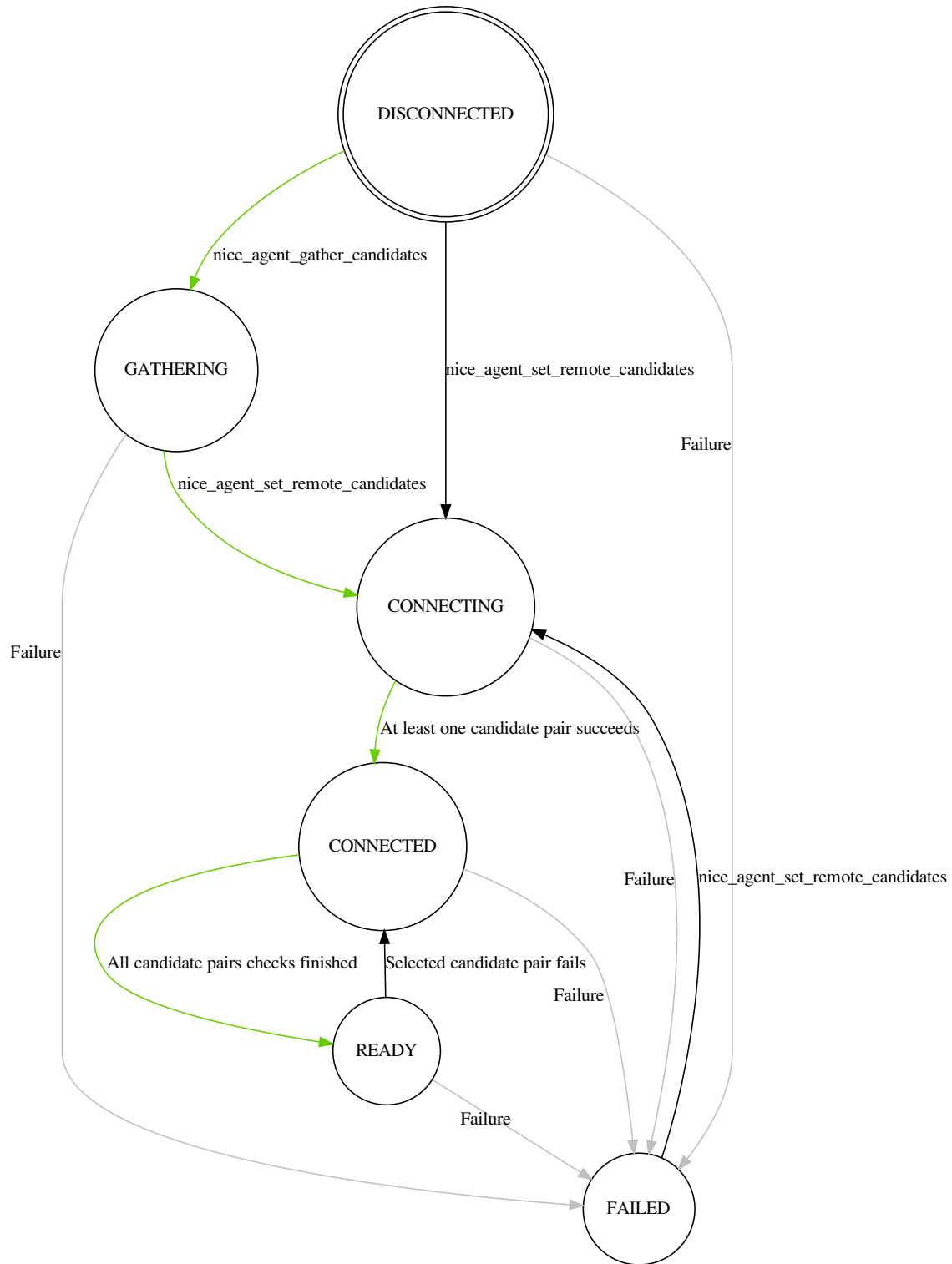


Fig. 1: libnice state transition diagram for NiceComponentState

(continued from previous page)

```
// 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; typically, candidates of type *host* (corresponding to the LAN, local network) are almost immediately found after starting the ICE gathering, and this event can arrive even before the event *IceComponentStateChanged* is emitted.

2. *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*.

19.1 WebRTC with ICE

ICE is the standard method used by *WebRTC* to solve the issue of *NAT Traversal*. Kurento supports ICE by means of a 3rd-party library: *libnice*, *The GLib ICE implementation*.

Refer to the *logging documentation* if you need to enable the debug logging for this library.

19.2 RTP without ICE

KMS is able to automatically infer what is the public IP and port of any remote peer which is communicating with it through an RTP connection. This removes the need to use ICE in some specific situations, where that complicated mechanism is not desired. This new automatic port discovery was inspired by the **Connection-Oriented Media Transport** (COMEDIA) as presented by the early Drafts of what finally would become the **RFC 4145**.

TCP-Based Media Transport in the Session Description Protocol (SDP) (**IETF RFC 4145**) defines an SDP extension which adds TCP connections and procedures, such as how a passive machine would wait for connections from a remote active machine and be able to obtain connection information from the active one, upon reception of an initial connection.

Early Drafts of **RFC 4145** (up to **Draft 05**) also contemplated the usage of this same concept of “Connection-Oriented Media Transport in SDP” with UDP connections, as a way of aiding *NAT Traversal*. This is what has been used as a basis for the implementation of automatic port discovery in KMS.

It works as follows:

1. The machine behind a *NAT* router acts as the active peer. It sends an SDP Offer to the other machine, the passive peer.
 - A. Sending an SDP Offer from behind a NAT means that the IP and port specified in the SDP message are actually just the private IP and port of that machine, instead of the public ones. The passive peer won't be able to use these to communicate back to the active peer. Due to this, the SDP Offer states the port 9 (*Discard port*) instead of whatever port the active machine will be using.
 - B. The SDP Offer includes the media-level attribute `a=direction:active`, so the passive peer is able to acknowledge that the Connection-Oriented Media Transport is being used for that media, and it writes `a=direction:passive` in its SDP Answer.

2. The passive peer receives the SDP Offer and answers it as usual, indicating the public IP and port where it will be listening for incoming packets. Besides that, it must ignore the IP and port indicated in the received SDP Offer. Instead, it must enter a wait state, until the active peer starts sending some packets.
3. When the active peer sends the first RTP/RTCP packets to the IP and port specified in the SDP Answer, the passive peer will be able to analyze them on reception and extract the public IP and reception port of the active peer.
4. The passive peer is now able to send RTP/RTCP packets to the discovered IP and port values of the active peer.

This mechanism has the following requisites and/or limitations:

- Only the active peer can be behind a NAT router. The passive peer must have a publicly accessible IP and port for RTP.
- The active peer must be able to receive RTP/RTCP packets at the same ports that are used to send RTP/RTCP packets. In other words, the active peer must be compatible with *Symmetric RTP and RTCP* as defined in [IETF RFC 4961](#).
- The active peer must actually do send some RTP/RTCP packets before the passive peer is able to send any data back. In other words, it is not possible to establish a one-way stream where only the passive peer sends data to the active peer.

This is how to enable the Connection-Oriented Media Transport mode:

- The SDP Offer must be sent from the active peer to the passive peer.
- The IP stated in the SDP Offer can be anything (as it will be ignored), so *0.0.0.0* can be used.
- The Port stated in the SDP Offer should be *9* (*Discard port*).
- The active peer must include the media-level attribute `a=direction:active` in the SDP Offer, for each media that requires automatic port discovery.
- The passive peer must acknowledge that it supports the automatic port discovery mode, by including the media-level attribute `a=direction:passive` in its SDP Answer. As per normal rules of the SDP Offer/Answer Model ([IETF RFC 3264](#)), if this attribute is not present in the SDP Answer, then the active peer must assume that the passive peer is not compatible with this functionality and should react to this fact as whatever is deemed appropriate by the application developer.

19.2.1 Example

This is a minimal example of an *SDP Offer/Answer* negotiation that a machine would perform with KMS from behind a *NAT* router. The highlighted lines are those relevant to NAT Traversal:

Listing 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
```

(continues on next page)

(continued from previous page)

```
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 H264/900000
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.

WEBRTC STATISTICS

[TODO full review]

20.1 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.

20.2 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){

      // ...
```

(continues on next page)

(continued from previous page)

```

    });
  });
};

```

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.
- *slrCount*: 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.
- *E2ELatency*: Array of average latencies (`MediaLatencyStat[]`) for each media (audio, video), in nanoseconds.

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*:

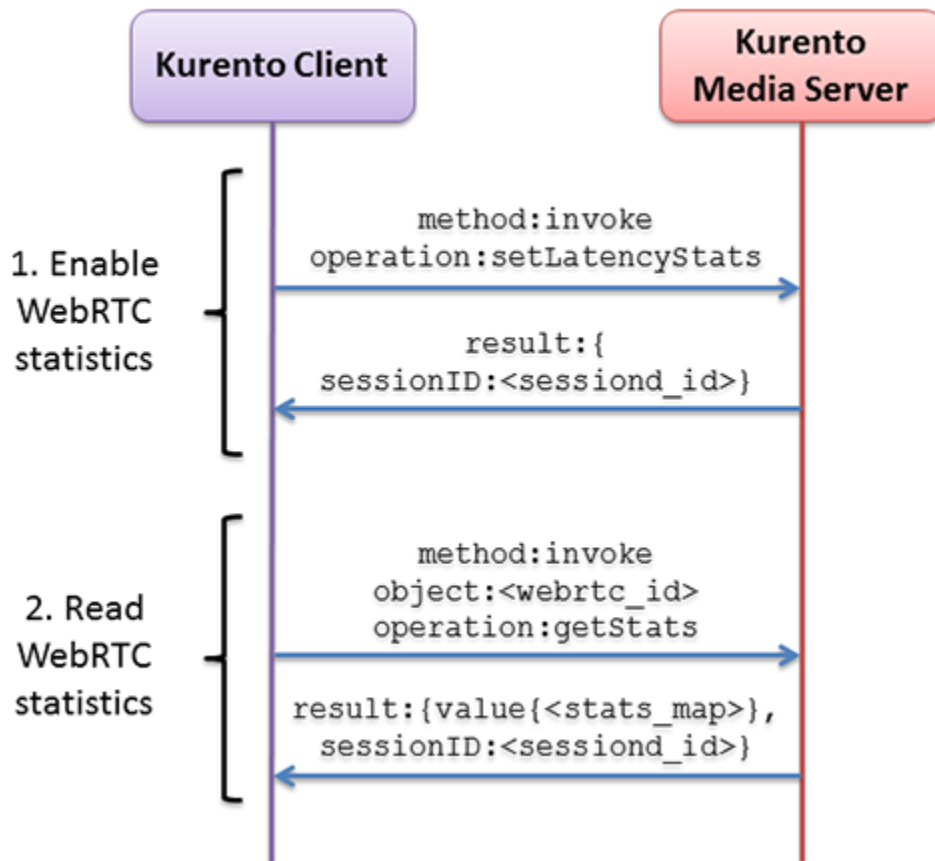


Fig. 1: Sequence diagram for gathering WebRTC statistics in KMS

20.3 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

KMS e2e latency: 0.278746 seconds

Stat	Browser	KMS
SSRC	842464803	842464803
Bytes received (browser)	12631812	12423332
Packets received (browser)	10424	10424
PLIs sent (browser)	0	0
FIRs sent (browser)	0	0
NACKs sent (browser)	0	0
Jitter	89	--
RTT	--	0.0006256103515625
Packet lost (browser)	0	--
REMB	--	823624

Fig. 2: *Statistics results in the kurento-loopback-stats demo GUI*

DEBUG LOGGING

Kurento Media Server prints log messages by using the [GStreamer logging library](#). This is a very flexible library that allows users to fine-tune the amount of verbosity that they want to get from the media server.

Logging verbosity is controlled by setting the `GST_DEBUG` environment variable with an appropriate string. In this section we'll show some useful examples, and then provide complete technical documentation about the logging features available for Kurento.

Table of Contents

- *Debug Logging*
 - *Default levels*
 - *Verbose logging*
 - * *Flowing of media*
 - * *Transcoding of media*
 - * *WebRtcEndpoint and RtpEndpoint*
 - * *PlayerEndpoint*
 - * *RecorderEndpoint*
 - * *Other components*
 - * *3rd-Party libraries*
 - *libnice*
 - *libsoup*
 - *Logs Location*
 - * *Logs Rotation*
 - *Log Contents*
 - * *Log colors*
 - *Logging levels and components*

21.1 Default levels

This is the default value for the `GST_DEBUG` environment variable, as found after installing Kurento Media Server for the first time:

```
export GST_DEBUG="2,Kurento*:4,kms*:4,sdp*:4,webrtc*:4,*rtppoint:4,rtp*handler:4,
↪rtpsynchronizer:4,agnosticbin:4"
```

Local installations will have this value set in the service settings file, `/etc/default/kurento-media-server` (for Debian/Ubuntu packages). On the other hand, the official *Docker images* come with this value already defined by default.

21.2 Verbose logging

While KMS is able to log a lot of technical details, most of them are disabled by default because otherwise the logging output would be huge. Here is a list of some strings that can be **added** to the default value of `GST_DEBUG`, to help with *Troubleshooting Issues*:

21.2.1 Flowing of media

```
export GST_DEBUG="${GST_DEBUG:-2},KurentoMediaElementImpl:5"
```

- “KurentoMediaElementImpl:5” shows *MediaFlowIn* and *MediaFlowOut* state changes, showing if media is actually flowing between endpoints (see *MediaElement events*).

21.2.2 Transcoding of media

```
export GST_DEBUG="${GST_DEBUG:-2},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.

21.2.3 WebRtcEndpoint and RtpEndpoint

- To show all state changes (*MediaFlowIn*, *MediaFlowOut*, *MediaTranscoding*, etc):

```
export GST_DEBUG="${GST_DEBUG:-2},Kurento*:5,KurentoWebSocket*:4"
```

- To show high-level debug messages of SDP processing in KMS (SDP Offer/Answer negotiation). In most situations this is the level you want to enable for troubleshooting issues with SDP:

```
export GST_DEBUG="${GST_DEBUG:-2},kmssdp session:5"
```

- For more verbose, low-level information about all SDP handling. Usually this is not needed except for the most advanced analysis of SDP issues:

```
export GST_DEBUG="${GST_DEBUG:-2},sdp*:5,basedp endpoint:5"
```

- To show the logic that governs ICE gathering and ICE candidate selection for WebRTC:

```
export GST_DEBUG="${GST_DEBUG:-2},webrtcendpoint:5,kmswebrtcsession:5,
↪kmsiceniceagent:5"
```

Note: See also *libnice* to enable advanced *ICE* logging for WebRTC.

- Lastly, to see messages about the *REMB* congestion control algorithm (adaptive video bitrate) for WebRTC. 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:-2},KurentoBaseRtpEndpointImpl:5,basertpendpoint:5,
↪kmsremb:5"
```

21.2.4 PlayerEndpoint

```
export GST_DEBUG="${GST_DEBUG:-2},KurentoUriEndpointImpl:5,uriendpoint:5,
↪playerendpoint:5,kmselement:5,appsrc:4,agnosticbin*:5,uridecodebin:6,rtspsrc:6,
↪soughhttpsrc:5,GST_URI:6,*CAPS*:3"
```

21.2.5 RecorderEndpoint

```
export GST_DEBUG="${GST_DEBUG:-2},KurentoUriEndpointImpl:5,uriendpoint:5,GST_URI:6,
↪KurentoRecorderEndpointImpl:5,recorderendpoint:5,basemediamuxer:5,qtmux:5,curl*:6"
```

21.2.6 Other components

Other less commonly used logging levels are:

- **imageoverlay, logooverlay** (as used, for example, in some *Kurento Tutorials*):

```
export GST_DEBUG="${GST_DEBUG:-2},imageoverlay:5,logooverlay:5"
```

- **RTP Synchronization:**

```
export GST_DEBUG="${GST_DEBUG:-2},kmsutils:5,rtpsynchroizer:5,rtpsyncontext:5,
↪basertpendpoint:5"
```

- **JSON-RPC API server:**

```
export GST_DEBUG="${GST_DEBUG:-2},KurentoServerMethods:5,KurentoWebSocket*:5"
```

- “KurentoServerMethods:5” shows WebSocket Ping/Pong messages. Use “KurentoServerMethods:6” for even more details about server session management such as caching of requests.
- “KurentoWebSocket*:5” shows all JSON-RPC messages that are sent and received, including Client/Server Keep-Alives.

- **Unit tests:**

```
export GST_DEBUG="${GST_DEBUG:-2},check:5,test_base:5"
```

21.2.7 3rd-Party libraries

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:-2},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*.

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
```

21.3 Logs Location

KMS prints by default all its log messages to standard output (*stdout*). This happens when the media server is run directly with `/usr/bin/kurento-media-server`, or when running from the official [Docker images](#).

Saving logs to file is enabled whenever the environment variable `KURENTO_LOGS_PATH` is set, or the `--logs-path` command-line flag is used. The KMS native packages take advantage of this, placing logs in a conventional location for the platform: `/var/log/kurento-media-server/`. This path can be customized by exporting the mentioned variable, or editing the service settings file located at `/etc/default/kurento-media-server` (from Debian/Ubuntu packages).

Log files are named as follows:

```
{DateTime}.{LogNumber}.pid{PID}.log
```

- *{DateTime}*: Logging file creation date and time, in [Wikipedia: ISO 8601](#) 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:

```
2018-06-14T194426.000000.pid13006.log
```

Besides normal log files, an *errors.log* file stores error messages and stack traces, in case KMS crashes.

21.3.1 Logs Rotation

When saving logs to file (due to either the environment variable `KURENTO_LOGS_PATH` or the `--logs-path` command-line flag), log files will be 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.

To configure this behavior:

- The `KURENTO_LOG_FILE_SIZE` env var or `--log-file-size` command-line flag control the maximum file size for rotating log files, in MB (default: 100 MB).
- The `KURENTO_NUMBER_LOG_FILES` env var or `--number-log-files` command-line flag set the maximum number of rotating log files to keep (default: 10 files).

21.4 Log Contents

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 [Wikipedia: ISO 8601](#) 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: `0x0000111122223333`.

- *{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 0x000007f59401f5880 info KurentoMediaServer main.  
→cpp:255 main() Kurento Media Server started
```

21.4.1 Log colors

Logs will be colored by default, but colors can be explicitly disabled: either with `--gst-debug-no-color` or with `export GST_DEBUG_NO_COLOR=1`.

When running KMS as a system service, the default settings will disable colors. This is done to write clean log files, otherwise the logs would end up filled with strange escape sequences (ANSI color codes).

21.5 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=2 \  
--gst-debug="Kurento*:4,kms*:4"
```

- You can also replace the command-line arguments with the *GST_DEBUG* environment variable. This command is equivalent to the previous one:

```
export GST_DEBUG="2,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="2,Kurento*:4,kms*:4"
```


KURENTO TEAM

The Kurento development team is part of 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 [NaevaTec](#).

- [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.
- [Juan Navarro](#) is our 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.
- [Pablo Fuente](#) works in [OpenVidu](#) and also in Kurento's Java and JavaScript client libraries.
- [Patxi Gortázar](#) is the original DevOps of Kurento platform and he's still working across several areas of the project.

CONTRIBUTION GUIDE

Table of Contents

- *Contribution Guide*
 - *Did you find a bug?*
 - *Did you fix a bug?*
 - *Did you fix whitespace, format code, or make a purely cosmetic patch?*
 - *Do you intend to add a new feature or change an existing one?*
 - *Thanks for helping*

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:

23.1 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](#).

23.2 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.

23.3 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.

23.4 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.

23.5 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

25.1 7.0.1 (UNRELEASED)

This is a template for changes that are being added to the next release.

To install Kurento Media Server: *Installation Guide*.

25.1.1 Added

- **Small addition.** Description.

Big addition name

Full description, with images, code samples, external links, etc. Some useful syntax examples:

- Documentation section: *Tutorials*.
- Glossary term: *SDP*.
- Inline link: [How to avoid Data Channel breaking](#).
- Blocks:

```
System.out.println("Some example Java code");
```

```
Some literal command output
```

Note: Something to keep in mind.

Thanks to @Username (Full Name, if available) for Kurento/kurento#{issue_id} (*Issue title*). Thanks to @Username (Full Name, if available) for Kurento/kurento#{pr_id} (*Pull Request title*).

25.1.2 Changed

- Description.

25.1.3 Deprecated

- Description.

25.1.4 Fixed

- Description.

25.1.5 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:

RepoName 1

- “Username for PR” ...

RepoName 2

- “Username for PR” ...

25.2 7.0.0 (March 2023)

A new Major version release of Kurento. This brings the opportunity to make breaking changes, which is used to clean the API a bit and fix some long standing issues with small details of the media server, such as the encoding format of audio recordings.

To install Kurento Media Server: *Installation Guide*.

Table of Contents

- *7.0.0 (March 2023)*
 - *Kurento 6.x to 7.0 Upgrade Guide*
 - * *timestamp -> timestampMillis*
 - * *MediaObject and MediaElement*
 - *Media Events*
 - *childs -> children*
 - *OuputBitrate, OutputBitrate -> EncoderBitrate*
 - * *WebRtcEndpoint*
 - *ICE Events*
 - *externalAddress -> externalIPv4, externalIPv6*
 - * *IceCandidatePair*

- * *Stats*
 - *inputAudioLatency, inputVideoLatency -> inputLatency*
 - *audioE2ELatency, videoE2ELatency -> E2ELatency*
- *Added*
 - * *requestKeyframe()*
- *Changed*
- *Removed*
- *Fixed*

25.2.1 Kurento 6.x to 7.0 Upgrade Guide

This section details all API changes that occur between Kurento versions 6 and 7. Following the method or member renames detailed here, you should be able to make the jump to newer versions of Kurento without requiring any rewrites at the logic level.

timestamp -> timestampMillis

Several object classes contained a `timestamp` field, which wasn't fine-grained enough, so the `timestampMillis` field was introduced to replace the former.

These classes are `Stats` (common parent of all `Stats` classes), and `RaiseBase` (common parent of all `Event` classes).

- Old: `timestamp` - Seconds elapsed since the UNIX Epoch (Jan 1, 1970, UTC)
- New: `timestampMillis` - Milliseconds elapsed since the UNIX Epoch (Jan 1, 1970, UTC)

MediaObject and MediaElement

These changes are located in the parent classes of all Kurento elements, so all Kurento classes are affected, such as `RtpEndpoint`, `WebRtcEndpoint`, `PlayerEndpoint`, `RecorderEndpoint`, etc.

Media Events

A series of deprecations and renamings that normalize all events into the same naming convention.

- Old: `MediaFlowOutStateChange` event
- New: `MediaFlowOutStateChanged` event
- Old: `MediaFlowInStateChange` event
- New: `MediaFlowInStateChanged` event
- Old: `MediaTranscodingStateChange` event
- New: `MediaTranscodingStateChanged` event

childs -> children

- Old: `MediaObject.getChilds()`
New: `MediaObject.getChildren()`

OutputBitrate, OutputBitrate -> EncoderBitrate

All `MediaElement`-derived classes had a `setOutputBitrate()` method that could be used to control the resulting bitrate of elements that perform encoding. This method was broken and didn't actually work as intended, see [Kurento/kms-core#30](#) for more details.

Instead of the *OutputBitrate* methods, use the new *EncoderBitrate* ones:

- Old: `setOutputBitrate()`
- New: `setEncoderBitrate()`
- Old: `setMinOutputBitrate(), setMaxOutputBitrate()`
- New: `setMinEncoderBitrate(), setMaxEncoderBitrate()`

WebRtcEndpoint

ICE Events

A series of deprecations and renamings that normalize all events into the same naming convention.

- Old: `OnIceCandidate` event
New: `IceCandidateFound` event
- Old: `OnIceGatheringDone` event
New: `IceGatheringDone` event
- Old: `OnIceComponentStateChanged`, `IceComponentStateChange` events
New: `IceComponentStateChanged` event
- Old: `OnDataChannelOpened`, `DataChannelOpen` events
New: `DataChannelOpened` event
- Old: `OnDataChannelClosed`, `DataChannelClose` event
New: `DataChannelClosed` event

externalAddress -> externalIPv4, externalIPv6

- Old: `externalAddress` setting
New: `externalIPv4`, `externalIPv6` settings
- Old: `getExternalAddress()`
New: `getExternalIPv4()`, `getExternalIPv6()`
- Old: `setExternalAddress()`
New: `setExternalIPv4()`, `setExternalIPv6()`

IceCandidatePair

Unifies all Kurento “Id” members under the same naming convention.

- Old: `streamID`
New: `streamId`
- Old: `componentID`
New: `componentId`

Stats

inputAudioLatency, inputVideoLatency -> inputLatency

- Old: `ElementStats.inputAudioLatency`, `ElementStats.inputVideoLatency` - Average latency, in nanoseconds.
New: `ElementStats.inputLatency` - Array of average latencies (`MediaLatencyStat[]`), in nanoseconds.

audioE2ELatency, videoE2ELatency -> E2ELatency

- Old: `EndpointStats.audioE2ELatency`, `EndpointStats.videoE2ELatency` - End-to-end latency, in nanoseconds.
New: `EndpointStats.E2ELatency` - Array of average latencies (`MediaLatencyStat[]`), in nanoseconds.

25.2.2 Added

requestKeyframe()

This method has been added to RTP-based elements (*RtpEndpoint*, *WebRtcEndpoint*) in order to allow requesting new keyframes from subscribing elements. This can be useful for streaming applications that want to force a new video keyframe on specific points in time.

Kurento Client API docs: [Java](#), [JavaScript](#).

25.2.3 Changed

- Change MP4 recorder audio codec from MP3 to AAC. This was a bad decision taken during the first stages of Kurento development, and couldn't be changed until now as it was considered a breaking change for user's media processing pipelines.

Thanks to [@Vijay-mRoads](#) for [Kurento/kms-core#11](#) (*Change MP4 recorder audio codec from MP3 to AAC*).

25.2.4 Removed

Kurento has always included several Computer Vision plugins and extension modules, *for demonstration purposes*. These were used to showcase the powerful, dynamic plug-and-play capabilities of Kurento Pipelines, providing a somewhat fancy way to convey how easy it is to manipulate video images in real time, with Kurento and OpenCV.

However, Kurento's OpenCV modules had been written against the old, C-based API of OpenCV 2.0. This was supported until OpenCV 4.0, which **marks the point where the code doesn't compile and must be disabled** (for now). List of disabled plugins:

- kms-crowddetector
- kms-markerdetector
- kms-platedetector
- kms-pointerdetector

These won't be available for installation. In future releases they might be brought up-to-date with the OpenCV C++ API, but no promises are made. If you'd like to see these plugins alive again, please make a Pull Request and/or contact us!.

25.2.5 Fixed

- Frame skipping when using `PlayerEndpoint.setPosition()`. This was caused by the incorrect usage of `GST_SEEK_FLAG_TRICKMODE`.

Thanks to @slabajo (Saúl Labajo) for [Kurento/kms-elements#44](#) (*Remove seek flags trickmode*).

- Duplicated element IDs with high loads. Two MediaPipelines or MediaElements could end up with the same element ID (which is supposed to always be unique) due to missing thread-safety mechanisms around the UUID library calls.

Thanks to @slabajo (Saúl Labajo) for [Kurento/kurento#4](#) (*Update UUIDGenerator.cpp*).

- Couldn't use special characters (/ ? @) in *PlayerEndpoint* and *RecorderEndpoint* URIs. This mainly affected users wanting to play RTSP sources, and it was caused by limitations in both sides of Kurento and the underlying GStreamer library. It should now be possible to use special characters in either of the username or password, which must be URL-encoded fields.

Kurento Client API docs: [Java](#), [JavaScript](#).

25.3 6.18.0 (September 2022)

One of the latest (if not the last) releases of the 6.x branch of Kurento; this one brings several deprecations that pave the way for introduction of the upcoming Kurento 7.0.

To install Kurento Media Server: [Installation Guide](#).

Table of Contents

- *6.18.0 (September 2022)*
 - *Added*
 - * *FLV Recording Profile for RTMP*
 - * *Explicit network interface for WebSocket*

- * *Differentiated Services Code Point (DSCP) for WebRTC QoS*
- *Changed*
 - * *WebRTC DTLS Quick Connection*
- *Deprecated: OpenCV extra modules*
- *Deprecated: Renamed API methods*
 - * *timestamp -> timestampMillis*
- * *MediaObject and MediaElement*
 - *Media Events*
 - *childs -> children*
 - *setOutputBitrate -> minOutputBitrate, maxOutputBitrate*
 - *minOutputBitrate, maxOutputBitrate -> minOutputBitrate, maxOutputBitrate*
- * *WebRtcEndpoint*
 - *ICE Events*
 - *externalAddress -> externalIPv4, externalIPv6*
- * *IceCandidatePair*
- * *Stats*
 - *inputAudioLatency, inputVideoLatency -> inputLatency*
 - *audioE2ELatency, videoE2ELatency -> E2ELatency*
- *Fixed*
- *Other changes*

25.3.1 Added

FLV Recording Profile for RTMP

The *RecorderEndpoint* gained a new **FLV** recording profile, which means that the resulting files can be used directly for RTMP streaming.

Thanks to @alex1712 (Alex) for [Kurento/kms-core#24](#) and [Kurento/kms-elements#30](#) (*FLV media profile*).

Explicit network interface for WebSocket

Up until now, the media server would open a WebSocket listening on port 8888 on all network interfaces. Any application wanting to control the media server, by means of the *Kurento Protocol*, would just connect to this port and start sending RPC requests.

However, some users need to be able to configure the exact interface where this control port is opened; now this can be done in the static configuration files.

Thanks to @Craeckie for [Kurento/kurento-media-server#21](#) (*WebSocketTransport: allow to set listen address in config*).

Differentiated Services Code Point (DSCP) for WebRTC QoS

Networks can provide different favorable routings for individual IPv4 and IPv6 packets, based on the *Differentiated Services* (DS) field of the IP header. This kind of *Quality of Service* (QoS) enhancement is enabled by setting the DS field to one of the *Differentiated Services Code Point* (DSCP) values that have been specified for usage with WebRTC, as defined by [RFC 8837](#).

At least some web browsers (i.e. Chrome) are able to set the DS field on its outbound packets, and now Kurento is also able to do the same thanks to this addition.

You can set a global DSCP value for all WebRtcEndpoints in the static configuration file, `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`. Alternatively, it is also possible to set the DSCP value separately for each endpoint, with the `qosDscp()` method of the WebRtcEndpoint Builder class.

Thanks to [@slabajo](#) (Saúl Labajo) for [Kurento/kms-elements#41](#) (*Feature/webrtc qos dscp*).

25.3.2 Changed

WebRTC DTLS Quick Connection

There was a design issue in the way Kurento established the WebRTC communications channel, that led it to mistakenly send the initial DTLS handshake packets before ICE had established a working network socket, leading to the dropping of such handshake packets.

Thankfully, DTLS doesn't give up as soon as some packets are dropped; instead, it just follows a progressively larger timeout scheme. Eventually the ICE protocol would establish a working socket, and one of the DTLS reattempts would finally be able to travel through it. However, this whole delay meant that WebRTC connections were taking much more time than what they really should.

Thanks to this change, now the DTLS handshake will be put on hold until ICE has finished its work and the network socket is well established and ready for comms; this way, the very first DTLS packets will already reach their destination, thus speeding up the whole process:

Comparison of connection speed (top: before; bottom: after)

Thanks to [@slabajo](#) (Saúl Labajo) for [Kurento/kms-elements#37](#) (*WebRTC DTLS handshake quick connection*) and [Kurento/kms-elements#38](#) (*dtls server quick connection fix*).

25.3.3 Deprecated: OpenCV extra modules

Kurento has always included several Computer Vision plugins and extension modules, for demonstration purposes. These were used to showcase the powerful, dynamic plug-and-play capabilities of Kurento Pipelines, providing a very visual and somewhat fancy way to convey how easy it is to manipulate video images in real time, with Kurento and OpenCV.

However, Kurento's OpenCV modules had been written against the old, C-based API of OpenCV 2.0. Over time, OpenCV 3.0 evolved into a more modern C++ based API, while keeping some backwards-compatibility with older C code. This was enough for us to keep publishing the Kurento demonstration plugins on Ubuntu 16.04 "Xenial" and Ubuntu 18.04 "Bionic". However, on 2020 the release of Ubuntu 20.04 "Focal", **OpenCV 4.0 marks the point where the Kurento plugin code doesn't compile and must be retired**.

The list of modules marked for retirement is as follows:

- Plugins that came included in *kms-filters* module:
 - facedetector
 - faceoverlay

- imageoverlay
- logooverlay
- movementdetector
- Plugins that were offered as additional installable modules:
 - kms-chroma
 - kms-crowddetector
 - kms-datachannelexample
 - kms-markerdetector
 - kms-platedetector
 - kms-pointerdetector

Starting with support for OpenCV 4.0, the old OpenCV 2.0 based plugin code cannot be compiled any more. Kurento project maintainers do not have the time, knowledge, or scheduling bandwidth to migrate these plugins into modern OpenCV 4.0 style code, so they will get discontinued until/unless some open-source community members can offer some help with porting them. If you'd like to see these plugins alive, and would be able to give us a hand, please contact us! :-)

These removals will be effective starting from the next major release, Kurento 7.0.

25.3.4 Deprecated: Renamed API methods

This section details all API deprecations that occur with the intention of paving the way for a cleaned up API in Kurento 7.0.

By following the renames detailed here, you should be able to make the jump to newer versions of Kurento without requiring any rewrites at the logic level.

timestamp -> timestampMillis

Several object classes contained a `timestamp` field, which wasn't fine-grained enough, so the `timestampMillis` field was introduced to replace the former.

These classes are `Stats` (common parent of all `Stats` classes), and `RaiseBase` (common parent of all `Event` classes).

- Old: `timestamp` - Seconds elapsed since the UNIX Epoch (Jan 1, 1970, UTC)
- New: `timestampMillis` - Milliseconds elapsed since the UNIX Epoch (Jan 1, 1970, UTC)

MediaObject and MediaElement

These changes are located in the parent classes of all Kurento elements, so all Kurento classes are affected, such as `RtpEndpoint`, `WebRtcEndpoint`, `PlayerEndpoint`, `RecorderEndpoint`, etc.

Media Events

A series of deprecations and renamings that normalize all events into the same naming convention.

- Old: `MediaFlowOutStateChange` event
New: `MediaFlowOutStateChanged` event
- Old: `MediaFlowInStateChange` event
New: `MediaFlowInStateChanged` event
- Old: `MediaTranscodingStateChange` event
New: `MediaTranscodingStateChanged` event

childs -> children

- Old: `MediaObject.getChilds()`
New: `MediaObject.getChildren()`

setOutputBitrate -> minOutputBitrate, maxOutputBitrate

All `MediaElement`-derived classes had a `setOutputBitrate()` method that could be used to set a specific target bitrate for the video stream. Instead, use the setters to specify a minimum and maximum desired target. To replicate the same behavior that `setOutputBitrate()` had, just provide the same value as both min and max.

- Old: `setOutputBitrate()`
New: `setMinOutputBitrate(), setMaxOutputBitrate()`

minOuputBitrate, maxOuputBitrate -> minOutputBitrate, maxOutputBitrate

These changes fix a typo in the original property names.

- Old: `getMinOuputBitrate(), setMinOuputBitrate()`
New: `getMinOutputBitrate(), setMinOutputBitrate()`
- Old: `getMaxOuputBitrate(), setMaxOuputBitrate()`
New: `setMaxOutputBitrate(), setMaxOutputBitrate()`

WebRtcEndpoint

ICE Events

A series of deprecations and renamings that normalize all events into the same naming convention.

- Old: `OnIceCandidate` event
New: `IceCandidateFound` event
- Old: `OnIceGatheringDone` event
New: `IceGatheringDone` event

- Old: `OnIceComponentStateChanged`, `IceComponentStateChange` events
New: `IceComponentStateChanged` event
- Old: `OnDataChannelOpened`, `DataChannelOpen` events
New: `DataChannelOpened` event
- Old: `OnDataChannelClosed`, `DataChannelClose` event
New: `DataChannelClosed` event

externalAddress -> externalIPv4, externalIPv6

- Old: `externalAddress` setting
New: `externalIPv4`, `externalIPv6` settings
- Old: `getExternalAddress()`
New: `getExternalIPv4()`, `getExternalIPv6()`
- Old: `setExternalAddress()`
New: `setExternalIPv4()`, `setExternalIPv6()`

IceCandidatePair

Unifies all Kurento “Id” members under the same naming convention.

- Old: `streamID`
New: `streamId`
- Old: `componentID`
New: `componentId`

Stats

inputAudioLatency, inputVideoLatency -> inputLatency

- Old: `ElementStats.inputAudioLatency`, `ElementStats.inputVideoLatency` - Average latency, in nanoseconds.
New: `ElementStats.inputLatency` - Array of average latencies (`MediaLatencyStat[]`), in nanoseconds.

audioE2ELatency, videoE2ELatency -> E2ELatency

- Old: `EndpointStats.audioE2ELatency`, `EndpointStats.videoE2ELatency` - End-to-end latency, in nanoseconds.
New: `EndpointStats.E2ELatency` - Array of average latencies (`MediaLatencyStat[]`), in nanoseconds.

25.3.5 Fixed

- [#289](#) (*Kurento scaffolder produces not compilable code if using a wrong name*).
- [#470](#) (*kmsaudiomixer outputs silence after running for 2 hours*).
- [#616](#) (*Kurento Media Server not sending relay candidates, although configured, unless ANSWER is received or OFFER is processed*).
- [#622](#) (*externalIpv4 and externalIpv6 affect all candidates instead of just host*).
- [#631](#) (*Kurento overwrites PPID of data channel packets as type “String” even when sent as “Binary”*).

25.3.6 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:

- [@dpocock](#) (Daniel Pocock) for [Kurento/kms-elements#35](#) (*CMakeLists.txt: explicitly list the Boost libraries required for linking*).
- [@DorianScholz](#) (Dorian Scholz) for [Kurento/kms-elements#42](#) (*data channel: do not overwrite ppid if it was successfully parsed from received data*).
- [@slabajo](#) (Saúl Labajo) for [Kurento/kurento-client-js#13](#) (*Fix memory leak in JavaScript Client*).

25.4 6.17.0 (March 2022)

This is a very small release, made to incorporate the ability to inherit new modules from *RtpEndpoint*.

To install Kurento Media Server: [Installation Guide](#).

25.4.1 Changed

Inheritable RtpEndpoint

The Kurento *RtpEndpoint* classes were *final*, meaning that it was not possible to inherit from them in order to create new, specialized versions of the endpoint. This was a problem for the fellow contributors at [Naeva Tec](#), who were trying to write *SipRtpEndpoint*, a new endpoint that contains specific behaviors to make it work well with SIP communications.

This limitation got solved in this release. With the reorganization of several classes and introduction of a new internal library for the *RtpEndpoint*, it can now be inherited to create new classes based on it.

Thanks to [@slabajo](#) (Saúl Labajo) for [Kurento/kms-elements#32](#) (*prepare for siprtp_module*) and [Kurento/kms-elements#33](#) (*RtpEndpoint library*).

25.4.2 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:

- [@pabs3](#) (Paul Wise) for [Kurento/kurento-module-creator#3](#) (*Add KurentoModuleCreatorConfig with path to FindKurentoModuleCreator*).

25.5 6.16.0 (March 2021)

To install Kurento Media Server: *Installation Guide*.

25.5.1 Added

ICE-TCP setting

ICE-TCP is what allows WebRTC endpoints to exchange ICE candidates that use the TCP protocol; in other words, the feature of using TCP instead of UDP for WebRTC communications.

Kurento had this setting enabled and it was hardcoded, so users were not able to easily change whether TCP should be used or not for ICE candidate exchange (the part of WebRTC that finds connectivity between peers). Thanks to this addition, it is now possible to use the **IceTcp** setting and disable ICE-TCP when desired.

So when should you use this setting? Well, for the most majority of cases the previous default was the best choice, but if you have a well known scenario and you are 100% sure that UDP will work, then disabling TCP will provide for slightly faster times when establishing WebRTC sessions. I.e., with ICE-TCP disabled, the time between joining a call and actually seeing the video will be smaller.

Of course, if you cannot guarantee that UDP will work in your network, then **you should leave this setting enabled**; otherwise, UDP might fail and there would be no TCP fallback for WebRTC to work.

Thanks to [@prlanzarin](#) (Paulo Lanzarin) for [Kurento/kms-elements#26](#) (*Add niceAgentIceTcp configuration option and API methods for it*).

Local install

- Set value `iceTcp` to 1 (ON) or 0 (OFF) in `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`.

Docker

- Set environment variable `KMS_ICE_TCP` to 1 (ON) or 0 (OFF).

Client API

- Java: `setIceTcp`.
- JavaScript: `setIceTcp`.

Packet loss correction in Recorder

RecorderEndpoint has gained a **new configuration file**: `/etc/kurento/modules/kurento/RecorderEndpoint.conf.ini`, where static settings can be written in the same manner than for other modules, such as *PlayerEndpoint* or *WebRtcEndpoint*.

For now, this file contains a single parameter: `gapsFix`, allowing users to decide which of the packet loss correction techniques they want to use for the recordings. Packet loss causes gaps in the input streams, and this can happen for example when an RTP or WebRTC media flow suffers from network congestion and some packets don't arrive at the media server.

Currently there are two of such techniques implemented:

- **NONE**: Do not fix gaps.

Leave the stream as-is, and store it with any gaps that the stream might have. Some players are clever enough to adapt to this during playback, so that the gaps are reduced to a minimum and no problems are perceived by the user; other players are not so sophisticated, and will struggle trying to decode a file that contains gaps. For example, trying to play such a file directly with Chrome will cause lipsync issues (audio and video will fall out of sync).

This is the best choice if you need consistent durations across multiple simultaneous recordings, or if you are anyway going to post-process the recordings (e.g. with an extra FFmpeg step).

- **GENPTS**: Adjust timestamps to generate a smooth progression over all frames.

This technique rewrites the timestamp of all frames, so that gaps are suppressed. It provides the best playback experience for recordings that need to be played as-is (i.e. they won't be post-processed). However, fixing timestamps might cause a change in the total duration of a file. So different recordings from the same session might end up with slightly different durations.

See the [extended description of GapsFixMethod](#) for more details about these settings.

Also have a look at [Introduction to Kurento](#) for an intro to all of the available Kurento modules.

25.5.2 Changed

ErrorEvent Type

Kurento Client API docs: [Java](#), [JavaScript](#).

The `ErrorEvent` can be emitted from any of the Kurento *MediaElement* objects (see [Kurento Modules](#) for more details). This Event has a **Type** string field that contains an error identifier, but until now it was mostly unused and all errors were identified as `UNEXPECTED_ELEMENT_ERROR`. On top of this, most errors were actually not being handled at all by the *MediaElement* where they occurred, so they would end up in the general handler of the *MediaPipeline*, with the *Type* identifier set to `UNEXPECTED_PIPELINE_ERROR`.

This now changes to provide better and more informational errors, so when possible the *ErrorEvent* will be emitted from the actual *MediaElement* where the error is taking place. Existing applications will continue to work, but the general recommendation holds that **Applications should subscribe to the Error event from all of Kurento objects**.

Some new *Type* identifiers have been added to the Error event:

- **RESOURCE_ERROR_OPEN**: Indicates that there was a problem when trying to open a local file or resource. This will typically happen when, for example, the *PlayerEndpoint* tries to open a file for which it does not have read permissions from the filesystem.
- **RESOURCE_ERROR_WRITE**: Similar to the previous one, this identifier marks an error writing to some file. This error could be seen when the *RecorderEndpoint* in Kurento lacks write permissions to the target path.

- **RESOURCE_ERROR_NO_SPACE_LEFT**: This error will mostly happen when a *RecorderEndpoint* is writing a recording but the disk becomes full. This is a common thing to happen if you don't have additional free space monitoring on your servers, so you should listen for this error from the *RecorderEndpoint* if you use it in any of your applications.
- **STREAM_ERROR_DECODE**: This error tends to happen when the sending side has transmitted an invalid encoded stream, and Kurento Media Server is trying to decode it but the underlying GStreamer library is unable to do so. This could happen, for example, when using a *PlayerEndpoint* (which by default decodes the input stream), or when *Transcoding* has been enabled due to incompatible codecs negotiated by different *WebRtcEndpoints*. When getting this error, you should review the settings of the sender, because there might be something wrong with its encoder configuration.
- **STREAM_ERROR_FAILED**: A generic error that is originated from the underlying GStreamer library when any data flow issue occurs. KMS debug logs should be checked because chances are that more descriptive information has been printed in there.

25.5.3 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

- @heirecka (Heiko Becker) for [Kurento/kms-core#25](#) (Include `<string>` for `std::string`).

kurento-client-js

- @stasee for [Kurento/kurento-client-js#3](#) (* Fix for invalid subscriptions*).

25.6 6.15.0 (November 2020)

To install Kurento Media Server: [Installation Guide](#).

25.6.1 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:

```
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**.

25.6.2 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.

25.6.3 Deprecated

- `externalAddress` should be replaced by `externalIPv4` and/or `externalIPv6`.

25.6.4 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-scanner*).

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.

25.6.5 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*).
- @tuttiee (Yuichiro Tsuchiya) for [Kurento/kurento-client-js#11](#) (*Fix register() calls inside lib/index.js*).

kurento-tutorial-js

- @tuttiee (Yuichiro Tsuchiya) for [Kurento/kurento-tutorial-js#9](#) (*Use IceCandidateFound event instead of deprecated OnIceCandidate event*).

25.7 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](#).

25.7.1 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.

25.7.2 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), where 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.

25.7.3 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*).
- [@piyushwadhvani](#) for [Kurento/doc-kurento#5](#) (*adapting to newer version of spring boot*).

kms-core

- [@pmlocek](#) for [Kurento/kms-core@2f144b5](#) (*fix: segfault in KmsIRtpSessionManager*).

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*).

25.8 6.13.2 (May 2020)

To install Kurento Media Server: [Installation Guide](#).

25.8.1 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*.

25.8.2 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.

25.8.3 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.

25.8.4 libnice 0.1.16

The [libnice](#) library is a core part of our WebRTC implementation, because it handles 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 keep-alives](#), [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.

25.8.5 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

- [@tuttiieee](#) 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*).

25.9 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](#).

25.9.1 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 payloaders (`rtppvp8pay`, `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:

```
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](#).

25.9.2 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.

25.9.3 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.

25.10 6.12.0 (October 2019)

Kurento Media Server **6.12** has been released!

To install it: [Installation Guide](#).

25.10.1 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](#).

25.10.2 Changed

- *libsrt* 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.

25.10.3 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.

25.10.4 Fixed

- Fix the **MediaFlowInStateChange crash**. [Kurento/bugtracker#393](#) (*MediaFlowInStateChange Seg Fault*).
- Fix leaks and possible 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.

25.11 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](#).

25.11.1 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*.

25.11.2 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).

25.11.3 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.

25.11.4 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.

25.11.5 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 Terasys Plugin).
- @kindritskiyMax (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

- @xenyoun (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).

25.12 6.10.0 (Apr 2019)

Kurento Media Server **6.10** is seeing the light with some important news!

To install it: *Installation Guide*.

25.12.1 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 hkp://keyserver.ubuntu.com:80 \
  --recv-keys 234821A61B67740F89BFD669FC8A16625AFA7A83

sudo tee "/etc/apt/sources.list.d/kurento.list" >/dev/null <<EOF
# Kurento Media Server - Release packages
deb [arch=amd64] http://ubuntu.openvidu.io/7.0.0 $DISTRO kms6
EOF

sudo apt-get update ; sudo apt-get install 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 resources, 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 **facetedetector** 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 rewrite 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.

25.12.2 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 1.2¹. 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.

25.12.3 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.

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 1.2¹.

¹ DTLS 1.2 was added in OpenSSL 1.0.2: [Major changes between OpenSSL 1.0.1f and OpenSSL 1.0.2](#) [22 Jan 2015].

25.12.4 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 `usrctp` 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.

25.12.5 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*”).

25.12.6 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 at the new knowledge section: [H.264 video codec](#).

25.12.7 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" : "VP8/900000"
  },
  {
    "name" : "H264/900000"
  }
]
```

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" : "VP8/900000"
// },
  {
    "name" : "H264/900000"
  }
]
```

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.

25.12.8 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.

25.12.9 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](#).
 - *rtph264[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](#).

25.13 6.9.0 (Dec 2018)

25.13.1 (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.

Remove global lock

!17 · opened 1 month ago by Olivier Crête 0.1.15 Internal enhancement

WIP: Use per-agent locks and GWeakRefs in callbacks from timeout sources

!13 · opened 3 months ago by Juan Navarro

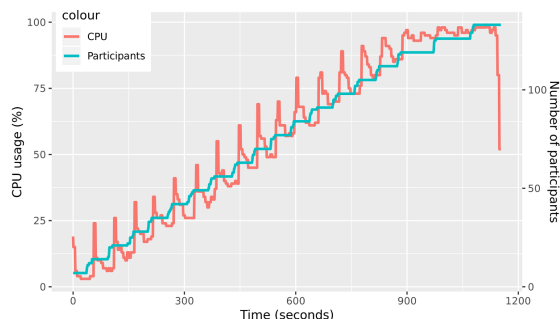
WIP: Use per-agent locks instead of a global lock

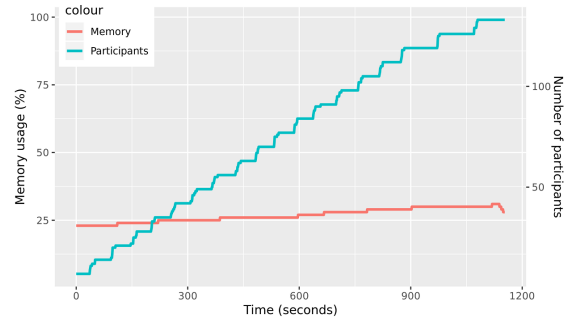
!12 · opened 4 months ago by Lorenzo Miniero

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:

25.13.2 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](#)!

25.13.3 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.

25.13.4 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*.

25.14 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.

25.14.1 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.

25.14.2 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:

```
2018-06-14T194426.000000.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.

25.14.3 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.

25.14.4 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!

25.15 6.7.2 (May 2018)

25.15.1 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.

25.15.2 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 (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.

25.15.3 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*.

Table of Contents

- *Developer Guide*
 - *Introduction*
 - *Code repositories*
 - *Development 101*
 - * *Libraries*
 - * *Debian packages*
 - * *Build tools*
 - *Build from sources*
 - * *Install required tools*
 - * *Install build dependencies*
 - * *Download the source code*
 - * *Build and run Kurento Media Server*
 - * *Clean up your system*
 - *Install debug symbols*
 - * *Why are debug symbols useful?*
 - *Run and debug with GDB*
 - * *GDB from sources*
 - * *GDB from installation*
 - * *GDB commands*
 - *Work on a forked library*
 - * *Full cycle*
 - * *In-place linking*

- *Create Deb packages*
 - * *kurento-buildpackage script*
 - * *kurento-buildpackage Docker image*
- *Unit Tests*
 - * *How to disable tests*
- *How-To*
 - * *How to add or update external libraries*
 - * *How to add new fork libraries*
 - * *How to work with API changes*
 - * *Known problems*

26.1 Introduction

This is an overview of the tools and technologies used by Kurento:

- Officially supported platform(s): **Ubuntu 20.04 (Focal)** (64-bits).
- 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 GStreamer multimedia framework sits at the heart of Kurento Media Server.
- In addition to GStreamer, Kurento uses lots of other libraries, such as Boost, jsoncpp, libsrtp, or libnice.

26.2 Code repositories

Kurento source code belongs to the Kurento organization at <https://github.com/Kurento>. Additionally, several 3rd-party libraries are also forked under the same organization; 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 modules of Kurento Media Server:

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:

Kurento 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 Kurento; 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.

- [libsrtp](#)
- [openh264](#)

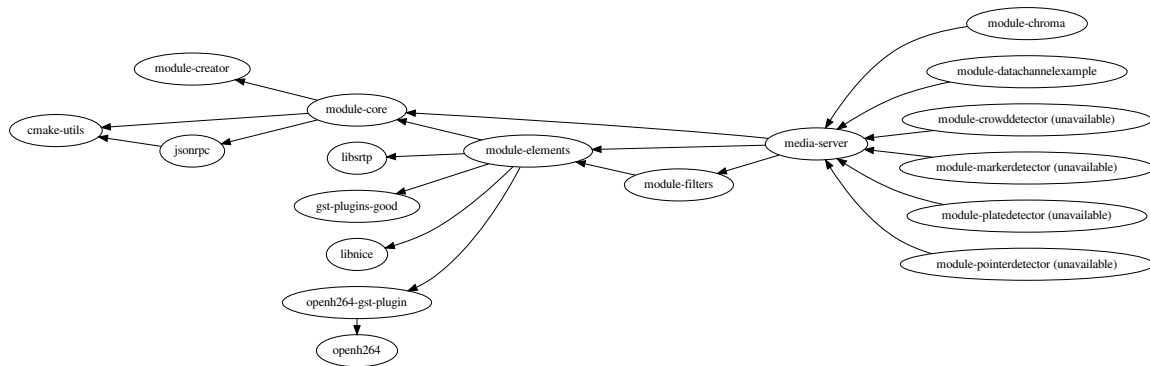


Fig. 1: Media Server dependency graph

- `openh264-gst-plugin`
- `gst-plugins-good`
- `libnice` (produces gstreamer1.0-nice)

Kurento monorepo

The bulk of source code resides in the Kurento monorepo: <https://github.com/Kurento/kurento>. The media server itself exists under the `server/` subdir, and contains these modules:

- `server/module-creator`: A code generation tool for generating code scaffolding for plugins. This code includes Kurento Media Server code, and Kurento client code.
- `server/cmake-utils`: Contains a set of utilities for building the media server with CMake.
- `server/jsonrpc`: The Kurento protocol is based on JsonRpc, and makes use of a JsonRpc library contained in this module.
- `server/module-core`: Core GStreamer code. This is the base plugin library that is needed for other plugins.
- `server/module-elements`: Main elements offering pipeline capabilities like WebRTC, RTP, media player, media recorder, etc.
- `server/module-filters`: Basic video filters included with the media server.
- `server/media-server`: Main entry point of the media server. That is, the `main()` function for the server executable code.

Example plugins

Kurento Media Server is distributed with some basic GStreamer pipeline elements, but other elements are available in form of example plugins. These showcase the kind of third party modules that could be written and integrated with Kurento, and are just for instructional purposes. Don't use them in production:

Note: These plugins were available for installation with Kurento 6.x; however, they are currently unavailable for Kurento 7.x due to breaking changes in OpenCV 4.0.

- `server/module-examples/chroma`
- `server/module-examples/crowddetector`

- `server/module-examples/datachannelexample`
- `server/module-examples/markerdetector`
- `server/module-examples/platedetector`
- `server/module-examples/pointerdetector`

There are also a couple minimal samples of what can be achieved with the default scaffolding done by Kurento Module Creator (see [Writing Kurento Modules](#)):

- `server/module-examples/gstreamer-example`
- `server/module-examples/opencv-example`

Clients

Application Servers can be developed in Java, JavaScript with Node.js, or JavaScript directly in the browser. Each of these languages have their respective client SDK:

- `clients/java`: For Application Servers written with Java technologies.
- `clients/javascript`: For Application Servers written with Node.js, or directly with browser JavaScript (not recommended).

This is an overview of the dependency graph for Java packages:

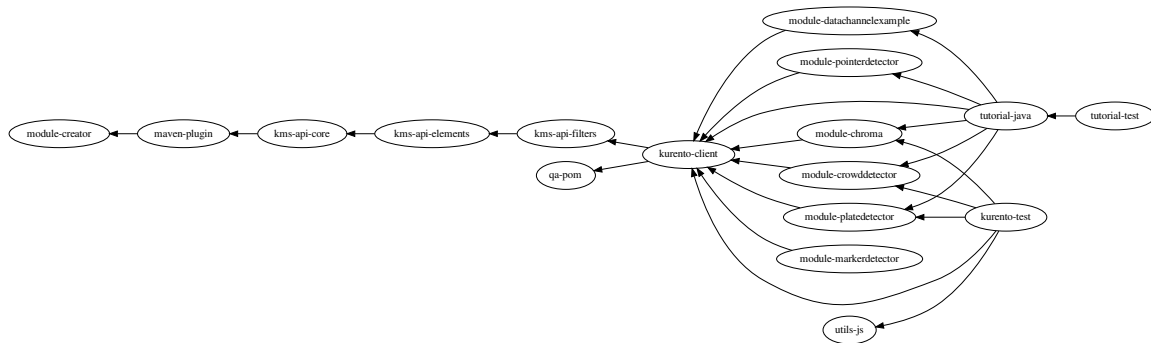


Fig. 2: Java dependency graph

Tutorials and examples

There are several repositories that contain sample applications for Kurento. Currently these are:

- `tutorials/java`
- `tutorials/javascript-node`
- `tutorials/javascript-browser`

A developer intending to work on Kurento itself must know how to work with the fork and server modules, and understand that each of these have a different development life cycle. Most of the development occurs at the server, while it's unusual to make changes in forks except for updating their upstream versions.

26.3 Development 101

Kurento 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.

26.3.1 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)
```

26.3.2 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`.

26.3.3 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. Kurento 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.

26.4 Build from sources

To build the source code of Kurento Media Server, you have 2 options:

- Build absolutely everything from scratch. Keeping in mind the dependency graph from *Code repositories*, you will need to start from the leftmost part and progress towards the right, building all projects one by one.
- Start from an intermediate point. For example if you only want to build Kurento Media Server itself, and not its dependencies, you can leverage the packages that are already built in the **Kurento packages repository** (see instructions for either the *Release repo* or *Development repo*).

In all cases, the workflow is the same. Follow these steps to end up with an environment that is appropriate for hacking on the Kurento source code:

1. Install required tools.
2. Install build dependencies.
3. Download the source code.
4. Build and run Kurento Media Server.
5. Build and run Kurento tests.

26.4.1 Install required tools

This command installs the basic set of tools that are needed for the next steps:

```
sudo apt-get update ; sudo apt-get install --no-install-recommends \  
build-essential \  
ca-certificates \  
cmake \  
git \  
gnupg \  
pkg-config
```

26.4.2 Install build dependencies

Option 1: Quick setup

If you install the `kurento-media-server-dev` package, all build dependencies will get installed too. This is a quick and easy way to get all the dependencies, if you don't care about building them from scratch.

First add the Kurento repos to your system, by following either of *release install* or *development install*. Then, run this command:

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

If you *do care* about building everything from scratch, keep reading.

Option 2: Build everything

All repositories that form the Kurento Media Server codebase are prepared to be packaged with Debian packaging tools. For every one of the fork libraries and modules of Kurento, you should have a look at its `debian/control` file, and make sure the dependencies listed in `Build-Depends` are installed in your system. This can be automated with `mk-build-deps` (which is part of the `devscripts` package).

For example, to build the Kurento *core* module:

```
sudo apt-get update ; sudo apt-get install --no-install-recommends \
    devscripts equivs

cd server/module-core/

# Get DISTRIB_* env vars.
source /etc/upstream-release/lsb-release 2>/dev/null || source /etc/lsb-release

sudo apt-get update ; sudo mk-build-deps --install --remove \
    --tool="apt-get -o Debug::pkgProblemResolver=yes --target-release 'a=${DISTRIB_
    ↪CODENAME}-backports' --no-install-recommends --no-remove" \
    ./debian/control
```

26.4.3 Download the source code

Run:

```
git clone https://github.com/Kurento/kurento.git
cd kurento/

git submodule update --init --recursive
```

26.4.4 Build and run Kurento Media Server

Change into the `server/` directory, then run this command:

```
export MAKEFLAGS="-j$(nproc)"

bin/build-run.sh
```

By default, the script `build-run.sh` will set up the environment and settings to make a Debug build of Kurento Media Server. 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 also set the logging level of specific categories by exporting the environment variable `GST_DEBUG` before running this script (see [Debug Logging](#)).

After the build has been completed, you can change into the build directory and run the unit tests. For more info, see [Unit Tests](#).

26.4.5 Clean up your system

To leave the system in a clean state, remove all Kurento packages and related development libraries. All Kurento packages contain the word “*kurento*” in the version number, so Aptitude makes it very easy to uninstall them all:

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

Use `purge` instead of `remove` to also delete any leftover configuration files in `/etc/`.

26.5 Install debug symbols

To work with Kurento source code itself or analyze a 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:

```
# Get DISTRIB_* env vars.
source /etc/upsream-release/lsb-release 2>/dev/null || source /etc/lsb-release

# Add Ubuntu debug repository key for apt-get.
apt-get update ; apt-get install --yes ubuntu-dbg-sym-keyring \
|| apt-key adv \
--keyserver hkp://keyserver.ubuntu.com:80 \
--recv-keys F2EDC64DC5AEE1F6B9C621F0C8CAB6595FDFF622

# Add Ubuntu debug repository line for apt-get.
sudo tee "/etc/apt/sources.list.d/ddebs.list" >/dev/null <<EOF
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 packages that are relevant to Kurento:

```
# Install debug packages.
# The debug packages repository fails very often due to bad server state.
# Try to update, and only if it works install debug symbols.
sudo apt-get update && sudo apt-get install --no-install-recommends --yes \
kurento-dbg
```

26.5.1 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::mediaFlowInStateChanged(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
```

(continues on next page)

(continued from previous page)

```
[g_hook_list_marshall]
/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::mediaFlowInStateChanged(int, char*, KmsElementPadType)]
/home/kurento/server/module-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/server/module-core/src/gst-plugins/commons/kmselement.c:578
[g_hook_list_marshall]
/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.

26.6 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 Kurento Media Server from sources and then use GDB to execute and debug it. Alternatively, you can also use GDB with an already installed version of Kurento.

26.6.1 GDB 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 Kurento with GDB.

For this step, the easiest method is to use our launch script, *build-run.sh*. It builds all sources, configures the environment, and starts up the debugger:

```
bin/build-run.sh --gdb
# [... wait for build ...]
(gdb)
```

4. Run GDB commands to *start Kurento Media Server* and then get a *backtrace* (see indications in next section).

26.6.2 GDB from installation

You don't *have* to build Kurento 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 Kurento 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 Kurento 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 Kurento is not already running as a service.

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

5. Run Kurento with GDB.

```
gdb /usr/bin/kurento-media-server
# [ ... GDB starts up ... ]
(gdb)
```

6. Run GDB commands to *start Kurento Media Server* and then get a *backtrace* (see indications in next section).

Running Kurento with Docker

If you are running Kurento from the Docker image, you can also follow the steps above, however a couple extra things must be done:

- Launch the Kurento Docker container with these additional arguments:

```
docker run -ti --cap-add SYS_PTRACE --security-opt seccomp=unconfined --entrypoint /
↪bin/bash [...]
```

- Skip steps 4 and 5 from above.

26.6.3 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 Kurento Media Server process
(gdb) run

# At this point, Kurento is running; now try to make the crash happen,
```

(continues on next page)

(continued from previous page)

```
# 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.

26.7 Work on a forked library

These are the two typical workflows used to work with fork libraries:

26.7.1 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 Kurento 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.

26.7.2 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 Kurento Media Server 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.

[TODO: Add concrete instructions for every forked library]

26.8 Create Deb packages

You can easily create Debian packages (*.deb* files) for Kurento 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](#).

26.8.1 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 development branch of *kurento-module-core* against Kurento 7.0.0. Run these commands:

```
git clone https://github.com/Kurento/kurento.git
cd kurento/server/module-core/
../../ci-scripts/kurento-buildpackage.sh \
  --install-kurento 7.0.0 \
  --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.

26.8.2 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 all Kurento packages for *Ubuntu 20.04 (Focal)*, from scratch (i.e. without jump-starting from the *apt-get* repositories), you've been saving them into `$HOME/packages/`, and now it's the turn of *kurento-module-core*. Run these commands:

```
git clone https://github.com/Kurento/kurento.git

cd kurento/

docker run --rm \
  --mount type=bind,src=server/module-core,dst=/hostdir \
  --mount type=bind,src=ci-scripts,dst=/ci-scripts \
  --mount type=bind,src="$HOME/packages",dst=/packages \
  kurento/kurento-buildpackage:focal \
  --install-files /packages \
  --dstdir /packages
```

26.9 Unit Tests

Kurento uses the Check unit testing framework for C (<https://libcheck.github.io/check/>). If you are working on the source code and *building from sources*, you can build and run unit tests manually: just `cd` to the build directory and run `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 will 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 could look like this:

```
export GST_DEBUG_NO_COLOR=1
export GST_DEBUG="3,check:5,test_base: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 test_<TestName>.check`. For example:

```
make test_agnosticbin.check
```

If you had Valgrind installed (to analyze memory usage), a `.valgrind` target will have been generated too. For example:

```
make test_agnosticbin.valgrind
```

26.9.1 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`.

Some tests require an IPv6-enabled network. These can be disabled temporarily in order to run all remaining tests on a machine that only has an IPv4 interface. Just use `-DDISABLE_IPV6_TESTS=TRUE` in either your CMake call, or in the Debian files where this property is mentioned, if you want to create `.deb` packages (currently, only `server/module-elements/debian/rules`).

26.10 How-To

26.10.1 How to add or update external libraries

Add or change it in these files:

- *debian/control*.
- *CMakeLists.txt*.

26.10.2 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.

26.10.3 How to work with API changes

What to do when you are developing a new feature that spans across the media server and the public API? This is a summary of the actions done in CI by `ci-scripts/kurento_generate_java_module.sh` and `ci-scripts/kurento_maven_deploy.sh`:

1. Work on your changes, which may include changing files where the Kurento API is defined.
2. Generate client SDK dependencies:

```
cd server/<module>/ # E.g. server/module-filters/
mkdir build ; cd build/
cmake -DGENERATE_JAVA_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_GENERATION=TRUE ..
cd java/
mvn -DskipTests=true clean install
```

3. Generate client SDK:

```
cd clients/java/
mvn -DskipTests=true clean install
```

4. At this point, the new Java packages have been generated and installed *in the local Maven cache*. Your Java application can now make use of any changes that were introduced in the API.

26.10.4 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`.

CONTINUOUS INTEGRATION

We have two types of repositories containing Debian packages: either Release or Nightly builds of the Kurento Main Repositories and Kurento Fork Repositories.

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/<KurentoVersion> <UbuntuCodename> main`
- Nightly: `deb http://ubuntu.openvidu.io/dev <UbuntuCodename> main`

Here, *<KurentoVersion>* is any of the released versions (such as 7.0.0) and *<UbuntuCodename>* is the name of each supported Ubuntu version (e.g. “focal”).

We also have several Continuous-Integration (CI) jobs such that new nightly packages can be built from each Git repository’s *main* branch, to be then uploaded to the nightly repositories.

All scripts used by CI are stored in the `ci-scripts/` subdir of the Kurento git repo: <https://github.com/Kurento/kurento>.

RELEASE PROCEDURES

Table of Contents

- *Release Procedures*
 - *Introduction*
 - * *General considerations*
 - *Release order*
 - *FIRST: Open a new Release Process*
 - *Kurento Media Server*
 - * *Preparation: kurento-module-creator*
 - * *Preparation: kurento-maven-plugin*
 - * *Preparation: API modules*
 - * *Release steps*
 - *Kurento JavaScript client*
 - * *Release steps*
 - * *Post-Release*
 - *Kurento Java client*
 - * *Preparation: kurento-java*
 - * *Release steps*
 - * *Post-Release*
 - *Docker images*
 - *Kurento documentation*
 - *LAST: Close the Release Process*
 - * *Kurento Media Server*
 - * *Kurento JavaScript client*
 - * *Kurento Java client*
 - * *Kurento documentation*

28.1 Introduction

This document describes all release procedures that apply to each one of the modules that are part of the Kurento project. The main form of categorization is by technology type: C/C++ based modules, Java modules, JavaScript modules, and others.

28.1.1 General considerations

- Lists of projects in this document are sorted according to the repository lists given in [Code repositories](#).
- During development, Kurento projects will have the future version number, followed by a development suffix:
 - -SNAPSHOT in Java (Maven) projects. Example: 1.0.0-SNAPSHOT.
 - -dev in C/C++ (CMake) and JavaScript projects. Example: 1.0.0-dev.

These suffixes must be removed for release, and then added again to start a new development cycle.

- The release version number doesn't need to match the one that had been in use during development. For example, after 1.0.1-dev, maybe enough features had been added that it gets released as 1.1.0 instead of 1.0.1. This is something that will be decided at the time of each release.
- Git tags are created and named with the version number of each release. Example: 1.0.0. No superfluous prefix is used, such as v.
- If hotfix branches are needed, they will use x as placeholder for the unspecified number. For example:
 - A support branch for the 1.1 minor release would be called 1.1.x.
 - A support branch for the whole 1 major release would be called 1.x.
- Contrary to the project version, Debian package versions don't contain development suffixes, and should always be of the form 1.0.0-1kurento1:
 - The first part (1.0.0) is the project's **base version number**.
 - The second part (1kurento1) is the **Debian package revision**:
 - * The number prefix indicates the version relative to other same-name packages provided by the base system.
 - * The number suffix means the amount of times that the package itself has been repackaged and republished. 1 means that this is the *first* time a given project version was packaged. If it was 1kurento3, it would mean that's the third time the same version has been repackaged.

Example:

A new version 1.0.0 is released for the first time. The full Debian package version will be: 1.0.0-1kurento1. Later, you realize the package doesn't install correctly in some machines, because of a bug in the package's post-install script. It's time to fix the mistake and republish! The software's source code itself has not changed at all, only the packaging files (in /debian/ dir); thus, the *base version* will remain 1.0.0, and only the *Debian revision* needs to change. The new package's full version will be 1.0.0-1kurento2.

Please check the [Debian Policy Manual](#) and [this Ask Ubuntu answer](#) for more information about 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 making backwards-incompatible or breaking API changes.

(continues on next page)

(continued from previous page)

2. MINOR version when adding new functionality in a backwards-compatible manner.
3. PATCH version when making backwards-compatible bug fixes.

Please refer to <https://semver.org/> for more information.

Example:

If the last release was **1.0.0**, then the next development version would be **1.0.1-dev** (or *1.0.1-SNAPSHOT* for Java components).

Some weeks later, the time comes to release this new development. If the new code only includes bug fixes and patches, then the version number *1.0.1* is already good. However, if some new features had been added, this new code should be released with version number *1.1.0*. The Debian package version is reset accordingly, so the full version is **1.1.0-1kurento1**.

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>
```

Note that the **main** branch in GitHub is a protected branch. This means force-pushing is disallowed, to avoid breaking the git trees of anyone who has this repository cloned.

28.2 Release order

First, the C/C++ parts of the code are built, Debian packages are created, and everything is left ready for deployment in an Apt repository (for *apt-get*) managed by [Aptly](#).

Before Kurento Media Server itself, all required forks and libraries must be built and installed:

- [libsrt](#)
- [openh264](#)
- [openh264-gst-plugin](#)
- [gst-plugins-good](#)
- [libnice](#)

The main *Kurento Media Server* modules should be built in this order:

- `server/module-creator`
- `server/cmake-utils`
- `server/jsonrpc`
- `server/module-core`
- `server/module-elements`
- `server/module-filters`
- `server/media-server`

And the example Kurento modules, which depend on Kurento's *core*, *elements*, and *filters*, can be built now:

- `server/module-examples/chroma`

(NOTE: Build disabled on Ubuntu >= 20.04 due to breaking changes in OpenCV 4.0)

- `server/module-examples/crowddetector`
- `server/module-examples/datachannelexample`
- `server/module-examples/markerdetector`
- `server/module-examples/platedetector`
- `server/module-examples/pointerdetector`

With this, the Media Server part of Kurento is built and ready for use. This includes an JSON-RPC server that listens for connections and speaks the *Kurento Protocol*.

To make life easier for application developers, there is a Java and a JavaScript client SDK that implements the RPC protocol. These are libraries that get auto-generated from each of the Kurento modules. See *[Kurento Java client](#)* and *[Kurento JavaScript client](#)*.

Java release order:

- `server/module-creator` ([org.kurento.kurento-module-creator](#))
- `clients/java/maven-plugin` ([org.kurento.kurento-maven-plugin](#))
- `clients/java/qa-pom` ([org.kurento.kurento-qa-pom](#))
- `server/module-core` ([org.kurento.kms-api-core](#))
- `server/module-elements` ([org.kurento.kms-api-elements](#))
- `server/module-filters` ([org.kurento.kms-api-filters](#))

- `clients/java` ([org.kurento.kurento-java](#), including [org.kurento.kurento-client](#))

After *kurento-client* is done, the client code for example Kurento modules can be built:

- `server/module-examples/chroma` ([org.kurento.module.chroma](#))
- `server/module-examples/crowddetector` ([org.kurento.module.crowddetector](#))
- `server/module-examples/datachannelexample` ([org.kurento.module.datachannelexample](#))
- `server/module-examples/markerdetector` ([org.kurento.module.markerdetector](#))
- `server/module-examples/platedetector` ([org.kurento.module.platedetector](#))
- `server/module-examples/pointerdetector` ([org.kurento.module.pointerdetector](#))

Now, the Kurento testing packages (which depend on some of the example modules). *kurento-utils-js* library must also be built at this stage, because it is a dependency of *kurento-test*:

- `browser/kurento-utils-js` ([kurento-utils](#))
- `test/integration` ([org.kurento.kurento-integration-tests](#), including [org.kurento.kurento-test](#))

And lastly, the tutorials (which depend on the example modules):

- `tutorials/java` ([org.kurento.tutorial.kurento-tutorial](#), including [org.kurento.tutorial.*](#))
- `test/tutorial`

JavaScript follows a similar ordering. Starting from *Kurento JavaScript client* for the main Kurento modules:

- `server/module-core` ([kurento-client-core](#))
- `server/module-elements` ([kurento-client-elements](#))
- `server/module-filters` ([kurento-client-filters](#))
- `clients/javascript/jsonrpc` ([kurento-jsonrpc](#))
- `clients/javascript/client` ([kurento-client](#))

Example Kurento modules:

- `server/module-examples/chroma` ([kurento-module-chroma](#))
- `server/module-examples/crowddetector` ([kurento-module-crowddetector](#))
- `server/module-examples/datachannelexample` ([kurento-module-datachannelexample](#))
- `server/module-examples/markerdetector` ([kurento-module-markerdetector](#))
- `server/module-examples/platedetector` ([kurento-module-platedetector](#))
- `server/module-examples/pointerdetector` ([kurento-module-pointerdetector](#))

And tutorials:

- `tutorials/javascript-node`
- `tutorials/javascript-browser`

Last, but not least, the project maintains a set of Docker images and documentation pages:

- [Docker images](#)
- [Kurento documentation](#)

28.3 FIRST: Open a new Release Process

To start with a new release, first of all create a new git branch that will contain all changes related to the release itself:

```
git switch --create release-1.0.0
git push --set-upstream origin HEAD
```

28.4 Kurento Media Server

All KMS projects:

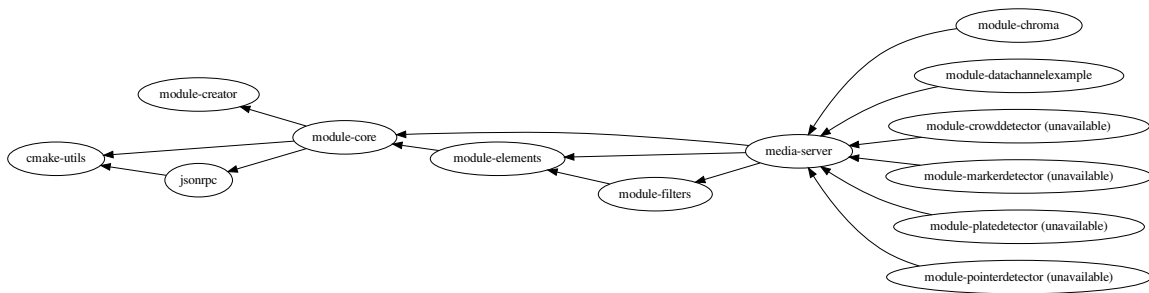


Fig. 1: Projects that are part of Kurento Media Server

Release order:

- server/module-creator
- server/cmake-utils
- server/jsonrpc
- server/module-core
- server/module-elements
- server/module-filters
- server/media-server
- server/module-examples/chroma
- server/module-examples/crowddetector
- server/module-examples/datachannelexample
- server/module-examples/markerdetector
- server/module-examples/platedetector
- server/module-examples/pointerdetector

28.4.1 Preparation: kurento-module-creator

- If *kurento-maven-plugin* is getting a new version, edit the file `server/module-creator/src/main/templates/maven/model_pom.xml.ftl` to update it:

```
<groupId>org.kurento</groupId>
<artifactId>kurento-maven-plugin</artifactId>
- <version>1.0.0</version>
+ <version>1.1.0</version>
```

Build the new version (if any), install it to the Maven cache, and set the PATH appropriately:

```
cd server/module-creator/
mvn -DskipTests=false clean install
export PATH="$PWD/scripts:$PATH"
```

28.4.2 Preparation: kurento-maven-plugin

Build the new version (if any) and install it to the Maven cache:

```
cd clients/java/maven-plugin/
mvn -DskipTests=false clean install
```

28.4.3 Preparation: API modules

Local check: Test that the KMS API module generation works.

Note that if the generation templates (*.ftl) have been changed, you'll probably need them to be in effect, and for that you'll need to use a local build of the Kurento Module Creator, instead of using the version that gets installed with the *kurento-module-creator* package.

This is the command to generate and build a Java module:

```
mkdir build/ && cd build/ \
&& cmake -DGENERATE_JAVA_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_GENERATION=TRUE .. \
&& cd java/ \
&& mvn -DskipTests=true clean install
```

For JavaScript modules, the command is very similar:

```
mkdir build/ && cd build/ \
&& cmake -DGENERATE_JS_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_GENERATION=TRUE .. \
&& cd js/ \
&& npm install
```

Complete code for Java and JavaScript modules:

```
sudo apt-get update ; sudo apt-get install --no-install-recommends \
kurento-module-creator \
kurento-cmake-utils \
kurento-jsonrpc-dev \
kurento-module-core-dev \
kurento-module-elements-dev \
```

(continues on next page)

(continued from previous page)

```

kurento-module-filters-dev

cd server/

function do_release {
    local PROJECTS=(
        module-core
        module-elements
        module-filters
        module-examples/chroma
        #module-examples/crowddetector
        module-examples/datachannelexample
        #module-examples/markerdetector
        #module-examples/platedetector
        #module-examples/pointerdetector
    )

    for PROJECT in "${PROJECTS[@]}; do
        pushd "$PROJECT" || { echo "ERROR: Command failed: pushd"; return 1; }

        mkdir -p build/ && cd build/

        cmake -DGENERATE_JAVA_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_GENERATION=TRUE .. \
        ↪ \
            && pushd java/ \
            && mvn -DskipTests=true clean install \
            && popd \
            || { echo "ERROR: Java code generation failed"; return 1; }

        cmake -DGENERATE_JS_CLIENT_PROJECT=TRUE -DDISABLE_LIBRARIES_GENERATION=TRUE .. \
            && pushd js/ \
            && npm install \
            && popd \
            || { echo "ERROR: JavaScript code generation failed"; return 1; }

        popd
    done

    echo "Done!"
}

# Run in a subshell where all commands are traced.
( set -o xtrace; do_release; )

```

28.4.4 Release steps

1. Choose the *final release version*, following the SemVer guidelines as explained in *General considerations*.
2. Set the new version. Most modules have a `bin/set-version.sh` script to make it easier with per-project specific commands.
3. Check there are no dangling development versions in any of the dependencies.
Search for the `-dev` suffix.
4. Commit changes (rebase and squash as needed).
5. Run the [Server Build All](#) job with parameters:
 - `jobGitName`: Release branch name (e.g. `release-1.0.0`).
 - `jobRelease`: **ENABLED**.
 - `jobOnlyKurento`: **DISABLED**.

All-In-One script:

```
# Change here.
NEW_VERSION="<ReleaseVersion>" # Eg.: 1.0.0
NEW_DEBIAN="<DebianRevision>" # Eg.: 1kurento1

cd server/

function do_release {
    # Set the new version.
    bin/set-versions.sh "$NEW_VERSION" --debian "$NEW_DEBIAN" \
        --release --commit \
    || { echo "ERROR: Command failed: set-versions"; return 1; }

    # Check for development versions.
    grep -Pr \
        --include CMakeLists.txt \
        --include '*.cmake' \
        --include '*.kmd.json' \
        --exclude-dir 'test*/' \
        -- '\d+\.\d+\.\d+-dev' \
    && { echo "ERROR: Development versions not allowed!"; return 1; }

    echo "Done!"
}

# Run in a subshell where all commands are traced.
( set -o xtrace; do_release; )

# Review committed changes. Amend as needed.
git log --max-count 1 --patch

# Push committed changes.
git push
```

28.5 Kurento JavaScript client

Release order:

- browser/kurento-utils-js (kurento-utils)
- server/module-core (kurento-client-core)
- server/module-elements (kurento-client-elements)
- server/module-filters (kurento-client-filters)
- clients/javascript/jsonrpc (kurento-jsonrpc)
- clients/javascript/client (kurento-client)

Example Kurento modules:

- server/module-examples/chroma (kurento-module-chroma)
- server/module-examples/crowddetector (kurento-module-crowddetector)
- server/module-examples/datachannelexample (kurento-module-datachannelexample)
- server/module-examples/markerdetector (kurento-module-markerdetector)
- server/module-examples/platedetector (kurento-module-platedetector)
- server/module-examples/pointerdetector (kurento-module-pointerdetector)

And tutorials:

- tutorials/javascript-node
- tutorials/javascript-browser

28.5.1 Release steps

1. Choose the *final release version*, following the SemVer guidelines as explained in [General considerations](#).
2. Set the new version. Most modules have a bin/set-version.sh script to make it easier with per-project specific commands.
3. Check there are no dangling development versions in any of the dependencies.
Search for the -dev suffix.
4. Commit changes (rebase and squash as needed).
5. Run the [Clients Build All JavaScript](#) job with parameters:
 - `jobRelease`: **ENABLED**.
 - `jobServerVersion`: Repository name of the release branch name (e.g. `dev-release-1.0.0`).

All-In-One script:

```
# Change here.
NEW_VERSION="<ReleaseVersion>" # Eg.: 1.0.0

function do_release {
  local PROJECTS=(
    browser/kurento-utils-js
    clients/javascript
```

(continues on next page)

(continued from previous page)

```

    tutorials/javascript-node
    tutorials/javascript-browser
)

for PROJECT in "${PROJECTS[@]"; do
    pushd "$PROJECT" \
    || { echo "ERROR: Command failed: pushd"; return 1; }

    # Set the new version.
    bin/set-versions.sh "$NEW_VERSION" --release --commit \
    || { echo "ERROR: Command failed: set-versions"; return 1; }

    # Check for development versions.
    grep -Pr --exclude-dir node_modules --include package.json -- '-dev|git\+http' \
    && { echo "ERROR: Development versions not allowed!"; return 1; }

    # Test the build.
    if [[ "$PROJECT" == "clients/javascript" ]]; then
        # kurento-client depends on kurento-jsonrpc, so install it
        # directly here to resolve the dependency.
        # Do not use `npm link`, because it is broken [1] and the link
        # will be lost with the `npm install` that comes afterwards.
        # [1]: https://github.com/npm/cli/issues/2372
        pushd jsonrpc/ && npm install && popd
        cd client/
        npm install \
            ../jsonrpc/ \
            ../../../../server/module-core/build/js/ \
            ../../../../server/module-elements/build/js/ \
            ../../../../server/module-filters/build/js/
    fi
    if [[ -f package.json ]]; then
        npm install || { echo "ERROR: Command failed: npm install"; return 1; }
    fi
    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 1; }
    fi

    popd
done

echo "Done!"
}

# Run in a subshell where all commands are traced.
( set -o xtrace; do_release; )

```

28.5.2 Post-Release

If CI jobs fail, the most common issue is that the code is not properly formatted. 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
```

When all CI jobs have finished successfully:

- Check that the auto-generated JavaScript client repos have been updated with the new version:
 - `kurento-client-core-js`
 - `kurento-client-elements-js`
 - `kurento-client-filters-js`
 - `kurento-module-chroma-js`
 - `kurento-module-crowddetector-js`
 - `kurento-module-datachannelexample-js`
 - `kurento-module-markerdetector-js`
 - `kurento-module-platedetector-js`
 - `kurento-module-pointerdetector-js`
- Check that the JavaScript packages have been published to NPM:
 - NPM: `kurento-client-core`
 - NPM: `kurento-client-elements`
 - NPM: `kurento-client-filters`
- Open the [Nexus Sonatype Staging Repositories](#) section.
- Select **kurento** repository.
- Inspect **Content** to ensure they are as expected:
 - `kurento-module-chroma-js`
 - `kurento-module-crowddetector-js`
 - `kurento-module-datachannelexample-js`
 - `kurento-module-markerdetector-js`
 - `kurento-module-platedetector-js`
 - `kurento-module-pointerdetector-js`
 - `kurento-utils-js`
 - `kurento-jsonrpc-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 within 30 minutes.

28.6 Kurento Java client

Release order:

- `server/module-creator` (`org.kurento.kurento-module-creator`)
- `clients/java/maven-plugin` (`org.kurento.kurento-maven-plugin`)
- `clients/java/qa-pom` (`org.kurento.kurento-qa-pom`)
- `server/module-core` (`org.kurento.kms-api-core`)
- `server/module-elements` (`org.kurento.kms-api-elements`)
- `server/module-filters` (`org.kurento.kms-api-filters`)
- `clients/java` (`org.kurento.kurento-java`, including `org.kurento.kurento-client`)

After *kurento-client* is done, the client code for example Kurento modules can be built:

- `server/module-examples/chroma` (`org.kurento.module.chroma`)
- `server/module-examples/crowddetector` (`org.kurento.module.crowddetector`)
- `server/module-examples/datachannelexample` (`org.kurento.module.datachannelexample`)
- `server/module-examples/markerdetector` (`org.kurento.module.markerdetector`)
- `server/module-examples/platedetector` (`org.kurento.module.platedetector`)
- `server/module-examples/pointerdetector` (`org.kurento.module.pointerdetector`)

Now, the Kurento testing packages (which depend on some of the example modules). *kurento-utils-js* library must also be built at this stage, because it is a dependency of *kurento-test*:

- `browser/kurento-utils-js` (`kurento-utils`)
- `test/integration` (`org.kurento.kurento-integration-tests`, including `org.kurento.kurento-test`)

And lastly, the tutorials (which depend on the example modules):

- `tutorials/java` (`org.kurento.tutorial.kurento-tutorial`, including `org.kurento.tutorial.*`)
- `test/tutorial`

Dependency graph:

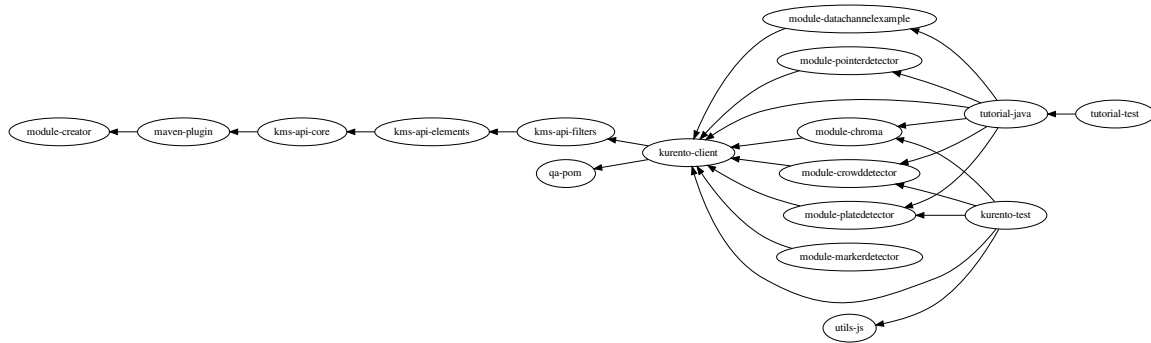


Fig. 2: Java dependency graph

28.6.1 Preparation: kurento-java

- If *kurento-maven-plugin* is getting a new version, edit the file `clients/java/parent-pom/pom.xml` to update it:

```

- <version.kurento-maven-plugin>1.0.0</version.kurento-maven-plugin>
+ <version.kurento-maven-plugin>1.1.0</version.kurento-maven-plugin>

```

- If *kurento-utils-js* is getting a new version, edit the file `clients/java/parent-pom/pom.xml` to update it:

```

- <version.kurento-utils-js>1.0.0</version.kurento-utils-js>
+ <version.kurento-utils-js>1.1.0</version.kurento-utils-js>

```

28.6.2 Release steps

1. Choose the *final release version*, following the SemVer guidelines as explained in [General considerations](#).
2. Set the new version. Most modules have a `bin/set-version.sh` script to make it easier with per-project specific commands.
3. Check there are no dangling development versions in any of the dependencies.
Search for the `-SNAPSHOT` suffix. Note that most versions are defined as properties in `clients/java/parent-pom/pom.xml`.
4. Commit changes (rebase and squash as needed).
5. Run the [Clients Build All Java](#) job with parameters:
 - *jobServerVersion*: Repository name of the release branch name (e.g. *dev-release-1.0.0*).

All-In-One script:

```

# Change here.
NEW_VERSION="<ReleaseVersion>" # Eg.: 1.0.0

function do_release {
    local PROJECTS=(

```

(continues on next page)

(continued from previous page)

```

clients/java/qa-pom
clients/java
tutorials/java
test/integration
test/tutorial
)

for PROJECT in "${PROJECTS[@]"; do
    pushd "$PROJECT" \
    || { echo "ERROR: Command failed: pushd"; return 1; }

    # Set the new version.
    bin/set-versions.sh "$NEW_VERSION" --kms-api "$NEW_VERSION" \
        --release --commit \
    || { echo "ERROR: Command failed: set-versions"; return 1; }

    # Check for development versions.
    grep -Fr --include pom.xml -- '-SNAPSHOT' \
    && { echo "ERROR: Development versions not allowed!"; return 1; }

    # Install the project.
    # * Build and run tests.
    # * Do not use -U because for each project we want Maven to find
    #   the locally cached artifacts from previous project.
    mvn -Pkurento-release -DskipTests=false clean install \
    || { echo "ERROR: Command failed: mvn"; return 1; }

    popd
done

echo "Done!"
}

# Run in a subshell where all commands are traced.
( set -o xtrace; do_release; )

```

28.6.3 Post-Release

When all CI jobs have finished successfully:

- Open the [Nexus Sonatype Staging Repositories](#) section.
- Select **kurento** repository.
- Inspect **Content** to ensure it is as expected:
 - org.kurento.kms-api-core
 - org.kurento.kms-api-elements
 - org.kurento.kms-api-filters
 - org.kurento.kurento-client
 - org.kurento.kurento-commons

- org.kurento.kurento-java
- org.kurento.kurento-jsonrpc
- org.kurento.kurento-jsonrpc-client
- org.kurento.kurento-jsonrpc-client-jetty
- org.kurento.kurento-jsonrpc-server
- org.kurento.kurento-maven-plugin
- org.kurento.kurento-module-creator
- org.kurento.kurento-parent-pom
- org.kurento.kurento-qa-config
- org.kurento.kurento-qa-pom
- org.kurento.module.chroma
- org.kurento.module.crowddetector (Unavailable since Kurento 7.0.0)
- org.kurento.module.datachannelexample
- org.kurento.module.markerdetector (Unavailable since Kurento 7.0.0)
- org.kurento.module.platedetector (Unavailable since Kurento 7.0.0)
- org.kurento.module.pointerdetector (Unavailable since Kurento 7.0.0)

All of them must appear in the correct version, \$NEW_VERSION.

- **Close** repository.
- Wait a bit.
- **Refresh**.
- **Release** repository.
- Maven artifacts will be available [within 30 minutes](#).

28.7 Docker images

A new set of development images is deployed to [Kurento Docker Hub](#) on each build. Besides, a release version will be published as part of the CI jobs chain when the [Server Build All](#) job is triggered.

28.8 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. Edit `doc-kurento/VERSIONS.env` 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 modules to verify what is the latest version of each one, and put it in the corresponding variable:

- [VERSION_DOC]: The docs version shown to readers. Normally same as [VERSION_KMS].
 - [VERSION_KMS]: Version of the Kurento Media Server
 - [VERSION_CLIENT_JAVA]: Version of the Java client SDK
 - [VERSION_CLIENT_JS]: Version of the JavaScript client SDK
 - [VERSION_UTILS_JS]: Version of *kurento-utils-js*
 - [VERSION_TUTORIAL_JAVA]: Version of the Java tutorials package.
 - [VERSION_TUTORIAL_NODE]: Version of the Node.js tutorials package.
 - [VERSION_TUTORIAL_JS]: Version of the Browser JavaScript tutorials package.
3. In `doc-kurento/VERSIONS.env`, set `VERSION_RELEASE` to **true**. Remember to set it again to *false* after the release, when starting a new development iteration.
 4. Test the build locally, check everything works.

```
python3 -m venv python_modules
source python_modules/bin/activate
python3 -m pip install --upgrade -r requirements.txt
make html
```

JavaDoc and JsDoc pages can be generated separately with `make langdoc`.

5. Commit changes (rebase and squash as needed).
6. Run the [Documentation build](#) job with parameters:
 - `jobRelease: ENABLED`.
7. CI automatically tags Release versions in ReadTheDocs generated repo `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 the `doc-kurento-readthedocs` repo. ReadTheDocs reads it to determine the documentation sources and release tags.

8. Check for errors in [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 detect that there is a new tag in the `doc-kurento-readthedocs` repo.

All-In-One script:

```
# Change here.
NEW_VERSION="ReleaseVersion" # Eg.: 1.0.0

cd doc-kurento/

function do_release {
    local COMMIT_MSG="Prepare documentation release $NEW_VERSION"

    # Set [VERSION_RELEASE]="true".
```

(continues on next page)

(continued from previous page)

```

sed -r -i 's/\[VERSION_RELEASE\]=.*\[VERSION_RELEASE]="true"/' VERSIONS.env \
|| { echo "ERROR: Command failed: sed"; return 1; }

# Set [VERSION_DOC].
local VERSION_DOC="$NEW_VERSION"
sed -r -i "s/\[VERSION_DOC\]=.*\[VERSION_DOC]=\"$VERSION_DOC\"/" VERSIONS.env \
|| { echo "ERROR: Command failed: sed"; return 2; }

# `--all` to include possibly deleted files.
git add --all \
    VERSIONS.env \
    source/project/relnotes/ \
&& git commit -m "$COMMIT_MSG" \
|| { echo "ERROR: Command failed: git"; return 4; }

echo "Done!"
}

# Run in a subshell where all commands are traced
( set -o xtrace; do_release; )

```

28.9 LAST: Close the Release Process

To finish the release, go to GitHub and create a new Pull Request from the release branch. Review all the changes, and accept the PR with a **merge commit**. Do not use the other options (squash or rebase), because we want all changes to get separately recorded with author and date information.

After merging the PR, fetch the new commit:

```

git switch main
git fetch --all --tags --prune --prune-tags
git pull --autostash

```

Tag the commit with the new version number (change 1.0.0 to the correct one):

```

git tag --force --annotate -m "1.0.0" "1.0.0"
git push --force origin "1.0.0"

```

And now you can optionally do some cleanup in your local clone:

```

git branch --force --delete release-1.0.0

```

Then, proceed to start a new development iteration for all of the Kurento components.

28.9.1 Kurento Media Server

1. Bump to a new development version. Do this by incrementing the *.PATCH* number and resetting the **Debian revision** to 1.
2. Run the [Server Build All](#) job with parameters:
 - *jobGitName*: (Default value).
 - *jobRelease*: **DISABLED**.
 - *jobOnlyKurento*: **DISABLED**.

All-In-One script:

```
# Change here.
NEW_VERSION="<NextVersion>" # Eg.: 1.0.1
NEW_DEBIAN="<DebianRevision>" # Eg.: 1kurento1

cd server/

# Set the new version.
bin/set-versions.sh "$NEW_VERSION" --debian "$NEW_DEBIAN" \
  --new-development --commit
```

28.9.2 Kurento JavaScript client

1. Bump to a new development version. Do this by incrementing the *.PATCH* number.
2. Run the [Clients Build All JavaScript](#) job with parameters:
 - *jobRelease*: **DISABLED**.
 - *jobServerVersion*: (Default value).

All-In-One script:

```
# Change here.
NEW_VERSION="<NextVersion>" # Eg.: 1.0.1

function do_release {
  local PROJECTS=(
    browser/kurento-utils-js
    clients/javascript
    tutorials/javascript-node
    tutorials/javascript-browser
  )

  for PROJECT in "${PROJECTS[@]"; do
    pushd "$PROJECT" \
    || { echo "ERROR: Command failed: pushd"; return 1; }

    # Set the new version.
    bin/set-versions.sh "$NEW_VERSION" --new-development --commit \
    || { echo "ERROR: Command failed: set-versions"; return 1; }

    popd
  done
}
```

(continues on next page)

(continued from previous page)

```
done

echo "Done!"
}

# Run in a subshell where all commands are traced.
( set -o xtrace; do_release; )
```

28.9.3 Kurento Java client

1. Bump to a new development version. Do this by incrementing the *.PATCH* number.
2. Run the [Clients Build All Java](#) job with parameters:
 - *jobServerVersion*: (Default value).

All-In-One script:

```
# Change here.
NEW_VERSION="<NextVersion>" # Eg.: 1.0.1

function do_release {
    local PROJECTS=(
        clients/java/qa-pom
        clients/java
        tutorials/java
        test/integration
        test/tutorial
    )

    for PROJECT in "${PROJECTS[@]"; do
        pushd "$PROJECT" \
        || { echo "ERROR: Command failed: pushd"; return 1; }

        # Set the new version.
        bin/set-versions.sh "$NEW_VERSION" --kms-api "${NEW_VERSION}-SNAPSHOT" \
            --new-development --commit \
        || { echo "ERROR: Command failed: set-versions"; return 1; }

        popd
    done

    echo "Done!"
}

# Run in a subshell where all commands are traced.
( set -o xtrace; do_release; )
```

28.9.4 Kurento documentation

1. Set `VERSION_RELEASE` to **false**.
2. Create a Release Notes document template where to write changes that will accumulate for the next release.
3. Run the [Documentation build](#) job with parameters:

- `jobRelease: DISABLED`.

All-In-One script:

```
# Change here.
NEW_VERSION="<NextVersion>" # Eg.: 1.0.1

cd doc-kurento/

function do_release {
    # Set [VERSION_RELEASE]="false"
    sed -r -i 's/\[VERSION_RELEASE\]=.*\[VERSION_RELEASE]="false"/' VERSIONS.env \
    || { echo "ERROR: Command failed: sed"; return 1; }

    # Set [VERSION_DOC]
    local VERSION_DOC="$NEW_VERSION-dev"
    sed -r -i "s/\[VERSION_DOC\]=.*\[VERSION_DOC]=\"$VERSION_DOC\"/" VERSIONS.env \
    || { echo "ERROR: Command failed: sed"; return 2; }

    # 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 3; }

    # Amend last commit with these changes.
    # This assumes that previous modules have been committed already,
    # otherwise just replace `--amend --no-edit`
    # with `-m "Prepare for next development iteration"`.
    git add \
        VERSIONS.env \
        source/project/relnotes/ \
    && git commit --amend --no-edit \
    || { echo "ERROR: Command failed: git"; return 4; }

    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 enabled¹. Ubuntu has also followed the same policy regarding their own build procedures².

Kurento Media Server had been lagging in this respect, and old releases only implemented the standard [Debian hardening options](#) that are applied by the *dpkg-buildflags* tool by default:

- **Format string checks** (`-Wformat -Werror=format-security`)³. These options instruct the compiler to make sure that the arguments supplied to string functions such as *printf* and *scanf* have types appropriate to the format string specified, and that the conversions specified in the format string make sense.
- **Fortify Source** (`-D_FORTIFY_SOURCE=2`)⁴. When this macro is defined at compilation time, several compile-time and run-time protections are enabled around unsafe use of some glibc string and memory functions such as *memcpy* and *strcpy*, with get replaced by their safer counterparts. This feature can prevent some buffer overflow attacks, but it requires optimization level `-O1` or higher so it is not enabled in Debug builds (which use `-O0`).
- **Stack protector** (`-fstack-protector-strong`)⁵. This compiler option provides a randomized stack canary that protects against *stack smashing* attacks that could lead to buffer overflows, and reduces the chances of arbitrary code execution via controlling return address destinations.
- **Read-Only Relocations** (RELRO) (`-Wl,-z,relro`). This linker option marks any regions of the relocation table as “read-only” if they were resolved before execution begins. This reduces the possible areas of memory in a program that can be used by an attacker that performs a successful *GOT-overwrite* memory corruption exploit. This option works best with the linker’s *Immediate Binding* mode, which forces *all* regions of the relocation table to be resolved before execution begins. However, immediate binding is disabled by default.

Starting from version **6.7**, KMS also implements these extra hardening measurements:

- **Position Independent Code** (`-fPIC`) / **Position Independent Executable** (`-fPIE -pie`)⁶. Allows taking advantage of the [Address Space Layout Randomization \(ASLR\)](#) protection offered by the Kernel. This protects against [Return-Oriented Programming \(ROP\)](#) 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.

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 https://man7.org/linux/man-pages/man7/feature_test_macros.7.html

⁵ <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

⁶ <https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html>

- **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.

29.1 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
```

29.2 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 `cmake-utils/CMake/CommonBuildFlags.cmake`, which is included by all modules of KMS.

29.3 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.
- 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*.

29.4 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

Table of Contents

- *Writing this documentation*
 - *Building locally*
 - * *Check spelling*
 - * *Install dependencies*
 - * *Build docs*
 - *Kurento documentation Style Guide*
 - * *Paragraph conventions*
 - * *Inline markup*
 - * *Header conventions*
 - * *Comments*
 - *Sphinx documentation generator*
 - *Read the Docs builds*
 - * *robots.txt*

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. This is done in a 3 steps process:

1. Sources are written in a markup language called **reStructuredText** (or **reST**, for short). This format is less known than the popular **Markdown**, but it is much more powerful and adequate for long-form documentation writing.
To learn the *reST* language have a look at the [Sphinx guide to reStructuredText](#) and the official [Quick reStructuredText](#) reference.
2. [Sphinx](#) is used to convert the *reST* files into the final output format, such as HTML or PDF.
3. The resulting files are hosted in [Read the Docs](#).

The core *reST* language is extended by Sphinx by means of *extensions*, and we use some of them:

- `sphinx.ext.extlinks`, to easily provide links to external sites (currently for the [English Wikipedia](#)).
- `sphinx.ext.graphviz`, to embed [Graphviz](#) DOT diagrams.
- `sphinx.ext.ifconfig`, to conditionally build different blocks into the documentation.

30.1 Building locally

30.1.1 Check spelling

Please 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:

```
code --install-extension streetsidesoftware.code-spell-checker
code --install-extension lextudio.restructuredtext
```

30.1.2 Install dependencies

You'll need these tools:

- Python 3, including:
 - The *PIP* package installer, used to install Sphinx.
 - Optionally, but strongly recommended, the Python's virtual env tool.
- Graphviz, to convert *.dot* files into graphs.

```
sudo apt-get update ; sudo apt-get install --no-install-recommends \
python3 python3-pip python3-venv \
graphviz
```

Optionally, make a bit of cleanup in case old Sphinx versions were installed:

```
# Ensure that old versions of Sphinx are not installed
sudo apt-get purge --auto-remove \
'^python-sphinx.*' \
'^python3-sphinx.*' \
'^sphinx.*'

python3 -m pip freeze | grep -i '^sphinx' | xargs sudo -H python3 -m pip uninstall
```

And finally, install Sphinx:

```
# Create and load the Python virtual environment
python3 -m venv python_modules
source python_modules/bin/activate

# Install Sphinx and the Read the Docs theme
python3 -m pip install --upgrade -r requirements.txt
```

30.1.3 Build docs

Run `make html` inside the documentation directory, and open the newly built files with a web browser:

```
# Load the Python virtual environment
source python_modules/bin/activate

# Build and open the documentation files
make html
firefox build/html/index.html
```

30.2 Kurento documentation Style Guide

30.2.1 Paragraph conventions

- **Line breaks:** *Don't* break the lines. Documentation is prose text, and not source code, so the typical code line length limit rules don't make any sense and don't apply here.

30.2.2 Inline markup

- Names, acronyms, and in general any kind of referential name should be emphasized with single asterisks (as in **word**).
- File names, full paths, URLs, package names, variable names, class and event names, code samples, commands, and in general any machine-oriented keywords, must be written inside double back-quotes (as in ``word``). This formatting *prevents line breaking*, which tends to be desirable for these kinds of technical words.

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.

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.

Important differences between *reST* and *Markdown*:

- *reST* uses **two back-quotes** for inline code, not one. It is ``word``, not ``word``.
- *reST* renders *single asterisks* (**word**) and *single back-quotes* (``word``) as *italic text*. For this reason, it's better to always use asterisks for emphasizing, to avoid confusing people who come from Markdown.
- *reST* does *not* render underscores (as in `_word_`), so don't use them to emphasize text.

30.2.3 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  
""""""
```

30.2.4 Comments

It is possible to include hidden comments, which work just like commented-out lines in any programming language. For this, use two dots in a single line, followed by indented text. For example:

```
..  
    These lines are commented out, and won't appear in the final output.  
    You can put here some notes about the text itself.
```

30.3 Sphinx documentation generator

Our Sphinx-based project is hosted in the `doc-kurento/` subdir within <https://github.com/Kurento/kurento>. 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: `7.0.0`.

Note: Sphinx already includes a substitutions feature by itself, for the keywords `version` and `release`. 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 during a pre-processing step, if we want reliable results.

The way this works is that the *source* folder gets 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 *Client API Reference*. 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 *main* branch if no such version tag exists. Then, the client stubs of the *Kurento Modules* 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/` subdir.
- **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.

30.4 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.

30.4.1 robots.txt

Read the Docs allows setting up a custom **robots.txt**, which we can use to prevent search engines from scrapping old and deprecated versions of the documentation, giving instead full priority to the `/latest/` and `/stable/` subdirectories in search engines:

- [How can I avoid search results having a deprecated version of my docs?](#).
- [Custom robots.txt Pages](#).

This is exactly the behavior we want, because without it, searches like “kurento webrtc” would show results from old 6.9 or 6.10 pages, while we’d rather have the latest or stable versions appearing.

Table of Contents

- *Testing*
 - *E2E Tests*
 - * *Functional*
 - * *Stability*
 - * *Tutorials*
 - * *API*
 - *Running Java tests*
 - * *Kurento Media Server*
 - * *Browsers*
 - * *Web server*
 - * *Fake clients*
 - * *Other test features*
 - *Kurento Testing Framework explained*
 - * *Test scenario in JSON*
 - * *TO-DO*

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.

31.1 E2E Tests

This section introduces the different types of E2E implemented to assess different parts of Kurento, namely **functional**, **stability**, **tutorials**, and **API**.

31.1.1 Functional

Functional tests are aimed to evaluate a given capability provided by Kurento. These tests have been created in Java. You can find the source code in the `test/integration/` subdir within <https://github.com/Kurento/kurento>. In order to run functional tests, Maven should be used as follows:

```
git clone https://github.com/Kurento/kurento.git
cd kurento/test/integration/
mvn \
  --projects 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 have 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 assess the result of this media mixing.

31.1.2 Stability

Stability tests verify 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 `test/integration/`. 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.git
cd kurento/test/integration/
mvn \
  --projects kurento-test --also-make \
  -Pintegration \
  -Dgroups=org.kurento.common.testing.SystemStabilityTests \
  clean verify
```

31.1.3 Tutorials

The documentation of Kurento includes a number of *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 `test/tutorial/`. In order to run these tests, Maven should be used:

```
git clone https://github.com/Kurento/kurento.git
cd kurento/test/tutorial/
mvn clean verify
```

31.1.4 API

Kurento provides *Java and JavaScript clients* that implement the *Kurento Protocol*. 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.git
cd kurento/test/integration/
mvn \
  --projects client-test --also-make \
  -Pintegration \
  -Dgroups=org.kurento.common.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.git
cd kurento/clients/javascript/client/
npm install
rm -f node_modules/kurento-client && ln -s .. node_modules/kurento-client
npm test
```

31.2 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 way which E2E Kurento are executed. Therefore, in order to run E2E tests, first we need to 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 `test/integration/` subdirectory within <https://github.com/Kurento/kurento>. Inside this project, we need to invoke Maven to execute tests, for example as follows:

```
git clone https://github.com/Kurento/kurento.git
cd kurento/test/integration/
mvn \
  --projects kurento-test --also-make \
  -Pintegration \
  -Dgroups=org.kurento.commons.testing.IntegrationTests \
  -Dtest=WebRtcOneLoopbackTest \
  clean verify
```

Let's take a closer look at 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-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.
- `-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 client-test`.
- `-Dtest=WebRtcOneLoopbackTest`: Although not mandatory, it is highly recommended, to select a test or group of tests 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).

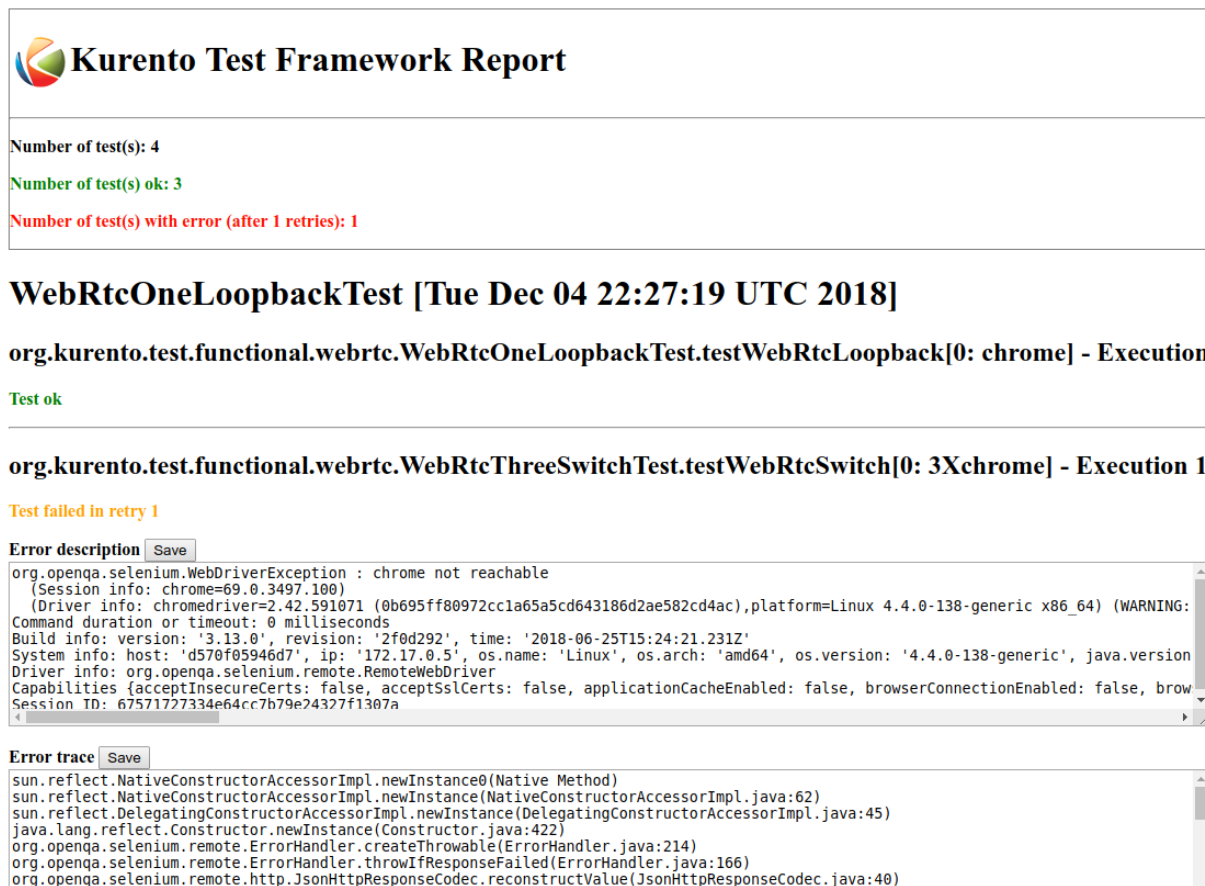
Note that the method name is given with a wildcard; this is because for most tests, the actual method name includes information about the browser which is used. Using a wildcard would run this test with both Chrome and Firefox; to choose specifically between those, specify it in the method name:

- `-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.



Kurento Test Framework Report

Number of test(s): 4
 Number of test(s) ok: 3
 Number of test(s) with error (after 1 retries): 1

WebRtcOneLoopbackTest [Tue Dec 04 22:27:19 UTC 2018]

org.kurento.test.functional.webrtc.WebRtcOneLoopbackTest.testWebRtcLoopback[0: chrome] - Execution 1/1

Test ok

org.kurento.test.functional.webrtc.WebRtcThreeSwitchTest.testWebRtcSwitch[0: 3Xchrome] - Execution 1/1

Test failed in retry 1

Error description

```
org.openqa.selenium.WebDriverException: chrome not reachable
(Session info: chrome=69.0.3497.100)
(Driver info: chromedriver=2.42.591071 (0b695ff80972cc1a65a5cd643186d2ae582cd4ac),platform=Linux 4.4.0-138-generic x86_64) (WARNING:
Command duration or timeout: 0 milliseconds
Build info: version: '3.13.0', revision: '2f0d292', time: '2018-06-25T15:24:21.231Z'
System info: host: 'd570f05946d7', ip: '172.17.0.5', os.name: 'Linux', os.arch: 'amd64', os.version: '4.4.0-138-generic', java.version
Driver info: org.openqa.selenium.remote.RemoteWebDriver
Capabilities {acceptInsecureCerts: false, acceptSslCerts: false, applicationCacheEnabled: false, browserConnectionEnabled: false, brow
Session ID: 67571727334e64cc7b79e24327f1307a
```

Error trace

```
sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
java.lang.reflect.Constructor.newInstance(Constructor.java:422)
org.openqa.selenium.remote.ErrorHandler.createThrowable(ErrorHandler.java:214)
org.openqa.selenium.remote.ErrorHandler.throwIfResponseFailed(ErrorHandler.java:166)
org.openqa.selenium.remote.http.JsonHttpStatusCode.reconstructValue(JsonHttpStatusCode.java:40)
```

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 summarize the main test parameters and their default values organized in different categories.

31.2.1 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
<i>test.kms.autostart</i>	Specifies if tests must start Kurento Media Server by themselves (with the method set by <i>test.kms.scope</i>), or if an external KMS service should be used instead: <ul style="list-style-type: none"> • <i>false</i>: Test must use an external KMS service, located at the URL provided by property <i>kms.ws.uri</i> • <i>test</i>: A KMS instance is automatically started before each test execution, and stopped afterwards. • <i>testsuite</i>: 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. 	<i>test</i>
<i>test.kms.scope</i>	Specifies how to start KMS when it is internally managed by the test itself (<code>-Dtest.kms.autostart != false</code>): <ul style="list-style-type: none"> • <i>local</i>: Try to use local KMS installation. Test will fail if no local KMS is found. • <i>remote</i>: KMS is a remote host (use <i>kms.login</i> and <i>kms.passwd</i>, or <i>kms.pem</i>, to access using SSH to the remote machine). • <i>docker</i>: Request the docker daemon to start a KMS container based in the image specified by <i>test.kms.docker.image.name</i>. 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). 	<i>local</i>
<i>test.kms.docker.image.name</i>	KMS docker image used to start a new docker container when KMS service is internally managed by test (<code>-Dtest.kms.autostart=test</code> or <code>testsuite</code>) with <i>docker</i> scope	<i>kurento/kurento-media-server:latest</i>
31.2. Running Java tests	(<code>-Dtest.kms.scope=docker</code>). Ignored if <i>test.kms.autostart</i> is <i>false</i> . See available Docker images for KMS in Docker Hub .	517

For example, in order to run the complete WebRTC functional test suite against a local instance KMS, the Maven command would be as follows:

```
cd kurento/test/integration/

mvn \
  --projects 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`).

31.2.2 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
<i>test.selenium.scope</i>	Specifies the scope used for browsers in Selenium test scenarios: <ul style="list-style-type: none"> • <i>local</i>: browser installed in the local machine. • <i>docker</i>: browser in Docker container (Chrome or Firefox). • <i>saucelabs</i>: browser in SauceLabs cloud. 	<i>local</i>
<i>docker.node.chrome.image</i>	Docker image identifier for Chrome when browser scope is <i>docker</i> .	<i>elastestbrowsers/chrome:latest</i>
<i>docker.node.firefox.image</i>	Docker image identifier for Firefox when browser scope is <i>docker</i> .	<i>elastestbrowsers/firefox:latest</i>
<i>test.selenium.record</i>	Allow recording the browser while executing a test, and generate a video with the completely test. This feature can be activated (<i>true</i>) only if the scope for browsers is <i>docker</i> .	<i>false</i>
<i>test.config.file</i>	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.	<i>test.conf.json</i>
<i>test.timezone</i>	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 <i>GMT</i> , <i>CET</i> , etc.	<i>none</i>
<i>saucelab.user</i>	User for SauceLabs	<i>none</i>
<i>saucelab.key</i>	Key path for SauceLabs	<i>none</i>
<i>saucelab.idle.timeout</i>	Idle time in seconds for SauceLabs requests	<i>120</i>
<i>saucelab.command.timeout</i>	Command timeout for SauceLabs requests	<i>300</i>
<i>saucelab.max.duration</i>	Maximum duration for a given SauceLabs session (in seconds)	<i>1800</i>

For example, in order to run the complete WebRTC functional test suite using *dockerized* browsers and recordings, the command would be as follows:

```
cd kurento/test/integration/

mvn \
  --projects kurento-test --also-make \
  -Pintegration \
  -Dgroups=org.kurento.common.testing.SystemFunctionalTests \
  -Dtest=WebRtc* \
```

(continues on next page)

(continued from previous page)

```
-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`).

31.2.3 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
<i>test.app.autostart</i>	Specifies whether test application where Selenium browsers connect must be started by test or if it is externally managed: <ul style="list-style-type: none"> • <i>false</i> : 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: <i>test.host</i>, <i>test.port</i>, <i>test.path</i> and <i>test.protocol</i>. • <i>test</i> : test application is started before each test execution. • <i>testsuite</i>: Test application is started at the beginning of test execution. 	<i>testsuite</i>
<i>test.host</i>	IP address or host name of the URL where Selenium browsers will connect when test application is externally managed (<i>-Dtest.app.autostart=false</i>). Notice this address must be reachable by Selenium browsers and hence network topology between browser and test application must be taken into account.	<i>127.0.0.1</i>
<i>test.port</i>	Specifies port number where test application must bind in order to listen for browser requests.	<i>7779</i>
<i>test.path</i>	Path of the URL where Selenium connects when test application is externally managed (<i>-Dtest.app.autostart=false</i>).	<i>/</i>
<i>test.protocol</i>	Protocol of the URL where Selenium browsers will connect when test application is externally managed (<i>-Dtest.app.autostart=false</i>).	<i>http</i>
<i>test.url.timeout</i>	Timeout (in seconds) to wait that web under test is available.	<i>500</i>

31.2.4 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
<i>fake.kms.scope</i>	This property is similar to <i>test.kms.scope</i> , except that it affects the KMS used by fake client sessions.	<i>local</i>
<i>fake.kms.ws.uri</i>	URL of a KMS service used by WebRTC clients. This property is used when service is externally managed (<code>-Dfake.kms.autostart=false</code>) and ignored otherwise. If not specified, <i>kms.ws.uri</i> is first looked at before using the default value.	<code>ws://localhost:8888/kurento</code>
<i>fake.kms.autostart</i>	<p>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):</p> <ul style="list-style-type: none"> • <i>false</i>: Test must use an external KMS service whose URL is provided by the property <i>fake.kms.ws.uri</i> (with <i>kms.ws.uri</i> as fallback). Test will fail if neither properties are provided. • <i>test</i>: KMS instance is started for before each test execution. KMS is destroyed after test execution. • <i>testsuite</i>: KMS service is started at the beginning of test suite execution. KMS service is stopped after test suite execution. <p>Following properties are honored when KMS is managed by test: <i>fake.kms.scope</i>, <i>test.kms.docker.image.name</i>, <i>test.kms.debug</i></p>	<i>false</i>

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](#).

31.2.5 Other test features

Kurento tests can be configured in many different ways. The following table summarizes these miscellaneous features for tests.

Parameter	Description	Default value
<code>test.num.retries</code>	Number of retries for failed tests	<code>1</code>
<code>test.report</code>	Path for HTML report	<code>target/report.html</code>
<code>test.project.path</code>	Path for test file output (e.g. log files, screen captures, and video recordings).	<code>target/surefire-reports/</code>
<code>test.workspace</code>	Absolute path of working directory used by tests as temporary storage. Make sure test user has full access to this folder.	<code>/tmp</code>
<code>test.workspace.absolute</code>	Absolute path, seen by docker agent, where directory <code>test.workspace</code> 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.	<code>none</code>
<code>test.docker.forcepulling</code>	Force pulling docker pull to always get the latest Docker images.	<code>true</code>
<code>test.files.disk</code>	Absolute path where test files (videos) are located.	<code>/var/lib/jenkins/test-files</code>
<code>test.files.http</code>	Hostname (without “ <code>http://</code> ”) of a web server where test files (videos) are located.	<code>files.openvidu.io</code>
<code>test.player.url</code>	URL used for playback tests. It can be anything supported by PlayerEndpoint: <code>file://...</code> , <code>http://...</code> , <code>rtsp://...</code> , etc.	<code>http://{test.files.http}</code>
<code>project.path</code>	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 <code>/</code> .	<code>.</code>
<code>kms.generatePath</code>	Path where mp3/pst statistics will be stored	<code>file://WORKSPACE/testClassName</code>
<code>bower.kurento-client</code>	Tag used by Bower to download kurento-client	<code>none</code>
<code>bower.kurento-utils</code>	Tag used by Bower to download kurento-utils.	<code>none</code>
<code>bower.release</code>	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.	<code>none</code>
<code>test.seek.repetitions</code>	Number of times the tests with seek feature will be executed	<code>100</code>
<code>test.num.sessions</code>	Number of total sessions executed in stability tests	<code>50</code>
<code>test.screenshots</code>	Size of the window to be shared automatically from tests	<code>Screen 1</code>

31.3 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:

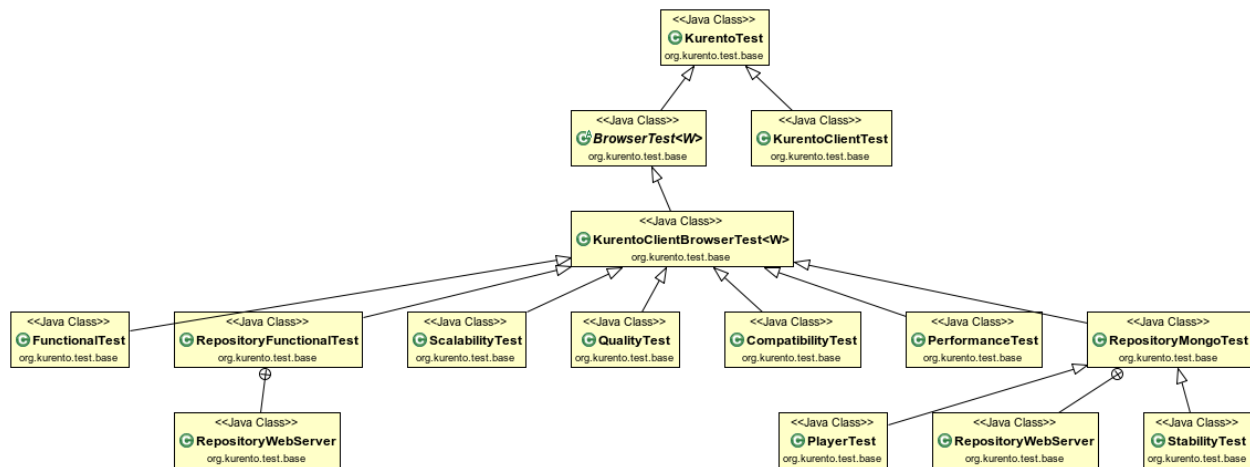


Fig. 2: Kurento Testing Framework class hierarchy

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:
 - Improved test lifecycle: KTF enhances the lifecycle 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 an 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.

31.3.1 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.

31.3.2 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*

BROWSER DETAILS

This page is a compendium of information that can be useful to work with or configure different web browsers, for tasks that are common to WebRTC development.

Example commands are written for a Linux shell, because that's what Kurento developers use in their day to day. But most if not all of these commands should be easily converted for use on Windows or Mac systems.

Table of Contents

- *Browser Details*
 - *Firefox*
 - * *Security sandboxes*
 - * *Test instance*
 - * *Debug logging*
 - *Examples*
 - *Safari*
 - *Chrome*
 - * *Test instance*
 - * *Debug logging*
 - * *Packet Loss*
 - * *H.264 codec*
 - *Command-line*
 - * *Chrome*
 - * *Firefox*
 - *WebRTC JavaScript API*
 - *Browser MTU*
 - *Bandwidth Estimation*
 - *Video Encoding*
 - * *Video Bitrate*
 - * *H.264 profile*

32.1 Firefox

32.1.1 Security sandboxes

Firefox has several sandboxes that can affect the logging output. For troubleshooting and development, it is recommended that you learn which sandbox might be getting in the way of the logs you need, and disable it:

For example:

- To get logs from `MOZ_LOG="signaling:5"`, first set `security.sandbox.content.level` to `0`.
- To inspect audio issues, disable the audio sandbox by setting `media.cubeb.sandbox` to `false`.

32.1.2 Test instance

To run a new Firefox instance with a clean profile:

```
/usr/bin/firefox -no-remote -profile "$(mktemp --directory)"
```

Other options:

- `-jsconsole`: Start Firefox with the [Browser Console](#).
- `[-url] <URL>`: Open URL in a new tab or window.

32.1.3 Debug logging

Sources:

- <https://wiki.mozilla.org/Media/WebRTC/Logging>
- https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/HTTP_logging
- https://developer.mozilla.org/en-US/docs/Mozilla/Command_Line_Options
- https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Gecko_Logging

Debug logging can be enabled with the parameters `MOZ_LOG` and `MOZ_LOG_FILE`. These are controlled either with environment variables, or command-line flags.

In Firefox `>= 54`, you can use `about:networking`, and select the Logging option, to change `MOZ_LOG` / `MOZ_LOG_FILE` options on the fly (without restarting the browser).

You can also use `about:config` and set any log option into the profile preferences, by adding (right-click -> New) a variable named `logging.<ModuleName>`, and setting it to an integer value of 0-5. For example, setting `logging.foo` to 3 will set the module `foo` to start logging at level 3 ("*Info*").

The special pref `logging.config.LOG_FILE` can be set at runtime to change the log file being output to, and the special booleans `logging.config.sync` and `logging.config.add_timestamp` can be used to control the `sync` and `timestamp` properties:

- **sync**: Print each log synchronously, this is useful to check behavior in real time or get logs immediately before crash.
- **timestamp**: Insert timestamp at start of each log line.

Logging Levels:

- **(0) DISABLED**: Indicates logging is disabled. This should not be used directly in code.

- **(1) ERROR:** An error occurred, generally something you would consider asserting in a debug build.
- **(2) WARNING:** A warning often indicates an unexpected state.
- **(3) INFO:** An informational message, often indicates the current program state. and rare enough to be logged at this level.
- **(4) DEBUG:** A debug message, useful for debugging but too verbose to be turned on normally.
- **(5) VERBOSE:** A message that will be printed a lot, useful for debugging program flow and will probably impact performance.

Log categories:

- Multimedia:
 - AudioStream:5
 - MediaCapabilities:5
 - MediaControl:5
 - MediaEncoder:5
 - MediaManager:5
 - MediaRecorder:5
 - MediaStream:5
 - MediaStreamTrack:5
 - MediaTimer:5
 - MediaTrackGraph:5
 - Muxer:5
 - PlatformDecoderModule:5
 - PlatformEncoderModule:5
 - TrackEncoder:5
 - VP8TrackEncoder:5
 - VideoEngine:5
 - VideoFrameConverter:5
 - cubeb:5
- WebRTC:
 - Autoplay:5
 - GetUserMedia:5
 - webrtc_trace:5
 - signaling:5
 - MediaPipeline:5
 - RtpLogger:5
 - RTCRtpReceiver:5
 - sdp:5

Examples

Linux:

```
export MOZ_LOG=timestamp,rotate:200,nsHttp:5,cache2:5,nsSocketTransport:5,  
↪nsHostResolver:5  
export MOZ_LOG_FILE=/tmp/firefox.log  
  
/usr/bin/firefox
```

Linux with *MOZ_LOG* passed as command line arguments:

```
/usr/bin/firefox \  
-MOZ_LOG=timestamp,rotate:200,nsHttp:5,cache2:5,nsSocketTransport:5,nsHostResolver:5  
↪ \  
-MOZ_LOG_FILE=/tmp/firefox.log
```

Mac:

```
export MOZ_LOG=timestamp,rotate:200,nsHttp:5,cache2:5,nsSocketTransport:5,  
↪nsHostResolver:5  
export MOZ_LOG_FILE=/tmp/firefox.log  
  
/Applications/Firefox.app/Contents/MacOS/firefox-bin
```

Windows:

```
set MOZ_LOG=timestamp,rotate:200,nsHttp:5,cache2:5,nsSocketTransport:5,nsHostResolver:5  
set MOZ_LOG_FILE=%TEMP%\firefox.log  
  
"C:\Program Files\Mozilla Firefox\firefox.exe"
```

ICE candidates / *STUN* / *TURN*:

```
export R_LOG_DESTINATION=stderr  
export R_LOG_LEVEL=7  
export R_LOG_VERBOSE=1  
  
/usr/bin/firefox -no-remote -profile "$(mktemp --directory)" \  
"https://localhost:8443/"
```

WebRTC dump example (see <https://blog.mozilla.org/webrtc/debugging-encrypted-rtp-is-more-fun-than-it-used-to-be/>):

```
export MOZ_LOG=timestamp,signaling:5,jsep:5,RtpLogger:5  
export MOZ_LOG_FILE="$PWD/firefox"  
  
/usr/bin/firefox -no-remote -profile "$(mktemp --directory)" \  
"https://localhost:8443/"  
  
grep -E '(RTP_PACKET|RTCP_PACKET)' firefox.*.moz_log \  
| cut -d '|' -f 2 \  
| cut -d ' ' -f 5- \  
| text2pcap -D -n -l 1 -i 17 -u 1234,1235 -t '%H:%M:%S.' - firefox-rtp.pcap
```

Media decoding (audio sandbox can be enabled or disabled with the user preference `media.cubeb.sandbox`):

```
export MOZ_LOG=timestamp,sync,MediaPipeline:5,MediaStream:5,MediaStreamTrack:5,webrtc_
↪trace:5

/usr/bin/firefox -no-remote -profile "$(mktemp --directory)" \
    "https://localhost:8443/"
```

32.2 Safari

To enable the Debug menu in Safari, run this command in a terminal:

```
defaults write com.apple.Safari IncludeInternalDebugMenu 1
```

32.3 Chrome

32.3.1 Test instance

To run a new Chrome instance with a clean profile:

```
/usr/bin/google-chrome --user-data-dir="$(mktemp --directory)"
```

32.3.2 Debug logging

Sources:

- <https://www.chromium.org/for-testers/enable-logging>
- <https://www.chromium.org/developers/how-tos/run-chromium-with-flags>
- <https://peter.sh/experiments/chromium-command-line-switches/>
- <https://webrtc.org/web-apis/chrome/>

Linux:

```
TEST_BROWSER="/usr/bin/chromium"
TEST_BROWSER="/usr/bin/google-chrome"
#
TEST_PROFILE="/tmp/chrome-profile"
#
{
    "$TEST_BROWSER" \
        --user-data-dir="$TEST_PROFILE" \
        --use-fake-ui-for-media-stream \
        --use-fake-device-for-media-stream \
        --enable-logging=stderr \
        --log-level=0 \
        --vmodule='*/webrtc/*=2,*/libjingle/*=2,*=-2' \
        --v=0 \
        "https://localhost:8443/" \
        >chrome_debug.log 2>&1 &
```

(continues on next page)

(continued from previous page)

```
# Other flags:
# --use-file-for-fake-audio-capture="/path/to/audio.wav" \
# --use-file-for-fake-video-capture="/path/to/video.y4m" \

tail -f chrome_debug.log
}
```

MacOS:

```
"/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" \
--enable-logging=stderr \
--vmodule=*/webrtc/*=2,*/libjingle/*=2,*=-2
```

32.3.3 Packet Loss

A command line for 3% sent packet loss and 5% received packet loss is:

```
--force-fieldtrials=WebRTCFakeNetworkSendLossPercent/3/
↪WebRTCFakeNetworkReceiveLossPercent/5/
```

32.3.4 H.264 codec

Chrome uses OpenH264 (same lib as Firefox uses) for encoding, and FFmpeg (which is already used elsewhere in Chrome) for decoding.

- Feature page: <https://www.chromestatus.com/feature/6417796455989248>
- Since Chrome 52.
- Bug tracker: <https://bugs.chromium.org/p/chromium/issues/detail?id=500605>

Autoplay:

- <https://developers.google.com/web/updates/2017/09/autoplay-policy-changes#best-practices>
- <https://www.chromium.org/audio-video/autoplay>

32.4 Command-line

32.4.1 Chrome

```
export WEB_APP_HOST_PORT="198.51.100.1:8443"

/usr/bin/google-chrome \
--user-data-dir="$(mktemp --directory)" \
--enable-logging=stderr \
--no-first-run \
--allow-insecure-localhost \
--allow-running-insecure-content \
```

(continues on next page)

(continued from previous page)

```
--disable-web-security \
--unsafely-treat-insecure-origin-as-secure="https://${WEB_APP_HOST_PORT}" \
"https://${WEB_APP_HOST_PORT}"
```

32.4.2 Firefox

```
export SERVER_PUBLIC_IP="198.51.100.1"

/usr/bin/firefox \
  -profile "$(mktemp --directory)" \
  -no-remote \
  "https://${SERVER_PUBLIC_IP}:4443/" \
  "http://${SERVER_PUBLIC_IP}:4200/#/test-sessions"
```

32.5 WebRTC JavaScript API

Generate an SDP Offer.

```
let pc1 = new RTCPeerConnection();
navigator.mediaDevices.getUserMedia({ video: true, audio: true })
.then((stream) => {
  stream.getTracks().forEach((track) => {
    console.log("Local track available: " + track.kind);
    pc1.addTrack(track, stream);
  });
  pc1.createOffer().then((offer) => {
    console.log(JSON.stringify(offer).replace(/\r\n/g, '\n'));
  });
});
```

32.6 Browser MTU

The default **Maximum Transmission Unit (MTU)** in the official [libwebrtc](#) implementation is **1200 Bytes** ([source code](#)). All browsers base their WebRTC implementation on *libwebrtc*, so this means that all use the same MTU:

- [Chrome source code](#).
- [Firefox source code](#).
- Safari: No public source code, but Safari uses Webkit, and [Webkit uses libwebrtc](#), so probably same MTU as the others.

32.7 Bandwidth Estimation

WebRTC **bandwidth estimation (BWE)** was implemented first with *Google REMB*, and later with *Transport-CC*. Clients need to start “somewhere” with their estimations, and the official *libwebrtc* implementation chose to do so at 300 kbps (kilobits per second) ([source code](#)). All browsers base their WebRTC implementation on *libwebrtc*, so this means that all use the same initial BWE:

- [Chrome source code](#).
- [Firefox source code](#).

32.8 Video Encoding

32.8.1 Video Bitrate

Web browsers will try to estimate the real performance of the network, and with this information they adapt their video output quality. Most browsers are able to adjust the **video bitrate**; in addition, Chrome also dynamically adapts the **resolution** and **framerate** of its video output.

The **maximum video bitrate** is calculated for WebRTC by following a simple rule based on the dimensions of the video source:

- 600 kbps if `width * height <= 320 * 240`.
- 1700 kbps if `width * height <= 640 * 480`.
- 2000 kbps (2 Mbps) if `width * height <= 960 * 540`.
- 2500 kbps (2.5 Mbps) for bigger video sizes.
- Never less than 1200 kbps, if the video is a screen capture.

Source: The `GetMaxDefaultVideoBitrateKbps()` function in [libwebrtc source code](#).

To verify what is exactly being sent by your web browser, check its internal WebRTC stats. For example, to check the outbound stats in Chrome:

1. Open this URL: `chrome://webrtc-internals/`.
2. Look for the stat name “*Stats graphs for RTCTransceiver (outbound-rtp)*”.
3. You will find the effective output bitrate in `[bytesSent_in_bits/s]`, and the output resolution in `frameWidth` and `frameHeight`.

You can also check what is the network bandwidth estimation in Chrome:

1. Look for the stat name “*Stats graphs for RTCTransceiver (candidate-pair)*”. Note that there might be several of these, but only one will be active.
2. Find the output network bandwidth estimation in `availableOutgoingBitrate`. Chrome will try to slowly increase its effective output bitrate, until it reaches this estimation.

32.8.2 H.264 profile

By default, Chrome uses this line in the SDP Offer for an H.264 media:

```
a=fmtp:100 level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=42e01f
```

profile-level-id is an SDP attribute, defined in [RFC 6184](#) as the hexadecimal representation of the *Sequence Parameter Set* (SPS) from the H.264 Specification. The value **42e01f** decomposes as the following parameters:

- *profile_idc* = 0x42 = 66
- *profile_iop* = 0xE0 = 1110_0000
- *level_idc* = 0x1F = 31

These values translate into the **Constrained Baseline Profile, Level 3.1**.

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*.

33.1 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.

33.2 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**.

Table of Contents

- *H.264 video codec*
 - *Profiles and Levels*
 - * *Profiles*
 - * *Levels*
 - *NAL Units (NALU)*
 - *SPS, PPS*
 - *GStreamer caps*
 - *GStreamer “codec_data”*
 - * *Example mapping*

Sources:

- [H.264 Specification](#)
- [Wikipedia: Advanced Video Coding](#)

34.1 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](#).

34.1.1 Profiles

These are the first five, most commonly used Profiles:

Profile name	Parameters
Baseline	<i>profile_idc</i> = 66 = 0x42, <i>constraint_set1_flag</i> = 0
Constrained Baseline	<i>profile_idc</i> = 66 = 0x42, <i>constraint_set1_flag</i> = 1
Main	<i>profile_idc</i> = 77 = 0x4D
Extended	<i>profile_idc</i> = 88 = 0x58
High	<i>profile_idc</i> = 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.

34.1.2 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 constraints 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.

34.2 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.

34.3 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.

34.4 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*.

34.5 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;
```

(continues on next page)

(continued from previous page)

```

    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.

34.5.1 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 = 1
  ↳ 0x01 = 1
    42                                     -> AVCProfileIndication = 66
  ↳ 0x42 = 66
      c0                                     -> profile_compatibility = 0
  ↳ 0xc0
        1f                                     -> AVCLevelIndication = 31
  ↳ = 31
          ff                                     -> lengthSizeMinusOne = 3
  ↳ = 3
            e1                                     -> numOfSequenceParameterSets = 1
  ↳ = 0b00001 = 1
              000e                                     -> sequenceParameterSetLength = 14
  ↳ = 0x000E = 14
                6742c01f8c8d40501e900f08846a
  ↳ sequenceParameterSetNALUnit
                    01                                     -> numOfPictureParameterSets = 1
  ↳ = 0x01 = 1
                        0004                                     -> pictureParameterSetLength = 4
  ↳ = 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**.

MEMORY FRAGMENTATION

Table of Contents

- *Memory Fragmentation*
 - *Problem background*
 - *Solution*
 - *Using Jemalloc*
 - *Other suggestions*

35.1 Problem background

Not all memory problems are related to memory leaks, and an application without leaks still could raise out-of-memory errors. One of the possible reasons for this is **Memory Fragmentation**, a problem that is conceptually similar to the well known disk fragmentation that affected some popular file systems such as FAT or NTFS on Windows systems.

There are numerous online resources where a description of the Memory Fragmentation problem can be found; two such resources are [Memory Leak Caused by Fragmentation \(archive\)](#) and [Preventing Memory Fragmentation \(archive\)](#).

In the presence of small and big memory allocations in a lineal memory space it may happen that, as memory is allocated, the free spaces between already allocated blocks may not be big enough to allocate a newly requested amount of memory, and this causes the process to request more memory from the Operating System. Later, before that big memory area is released, some other smaller blocks could have been allocated near it. This would cause that the big memory area cannot be given back to the OS. when the application releases it. This memory area would then be marked as “ready to be reclaimed”, but the Kernel won’t get an immediate handle of it.

Forward some hundreds or thousands of memory allocations later, and this kind of memory fragmentation can end up causing an out-of-memory error, even though technically there are lots of released memory areas, which the Kernel hasn’t been able to reclaim.

This issue is more likely to happen in systems that make heavy use of dynamic memory and with different sizes for allocated memory, which is really the case with Kurento Media Server.

35.2 Solution

The best option we know about is replacing `malloc`, the standard memory allocator that comes by default with the system, with a specific-purpose allocator, written with this issue in mind in order to avoid or mitigate it as much as possible.

Two of the most known alternative allocators are [Jemalloc \(code repository\)](#) and Google's [TCMalloc \(code repository\)](#).

Jemalloc has been tested with Kurento Media Server, and found to give pretty good results. It is important to note that internal fragmentation cannot be reduced to zero, but this alternative allocator was able to reduce memory fragmentation issues to a minimum.

35.3 Using Jemalloc

First install it on your system. For the versions of Ubuntu that are explicitly supported by Kurento, you can run this command:

```
sudo apt-get update ; sudo apt-get install libjemalloc1
```

Jemalloc is installed as a standalone library. To actually use it, you need to preload it when launching KMS:

```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so.1 /usr/bin/kurento-media-server
```

This will use Jemalloc with its default configuration, which should be good enough for normal operation.

If you know how to fine-tune the allocator with better settings than the default ones, you can do so with the `MALLOC_CONF` environment variable. For example:

```
export MALLOC_CONF=stats_print:true
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so.1 /usr/bin/kurento-media-server
```

This would cause KMS to dump memory usage statistics when exiting. Those statistics could then be used to tune Jemalloc and define the configuration to use.

Note: To use Jemalloc from inside a Docker container, you'll want to make a custom image that is derived from the official one, where the required package is installed and the Docker Entrypoint script has been modified to add the library preloading step.

There is some additional information about how to start making a customized image in [Kurento in Docker](#).

Note: Since Ubuntu 20.04, the package is named `libjemalloc2` and the library file is `/usr/lib/x86_64-linux-gnu/libjemalloc.so.2`.

35.4 Other suggestions

It is a good idea to maintain health checks on servers that are running Kurento Media Server, to automatically detect and react against memory exhaustion issues. We suggest maintaining some health probes on KMS, that at least should take care of memory usage and behave as follows:

1. Maintain a probe on memory usage of the Kurento Media Server process.
2. As soon as that usage grows over a threshold value in a sustained manner (i.e. it does not get back when sessions finish), that server instance should be recycled:
 - 2.1. It should not accept further sessions, and
 - 2.2. As soon as the last session is finished, the Kurento Media Server instance should be stopped (and probably restarted).

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

Table of Contents

- *NAT Types and NAT Traversal*
 - *Basic Concepts*
 - * *Transport Address*
 - * *Packet transmission*
 - * *Inbound NAT transmission*
 - *Types of NAT*
 - * *Full Cone NAT*
 - * *(Address-)Restricted Cone NAT*
 - * *Port-Restricted Cone NAT*
 - * *Symmetric NAT*
 - *Types of NAT in the Real World*
 - *NAT Traversal*
 - * *Do-It-Yourself hole punching*
 - * *PyNAT*

Sources:

- [Wikipedia: Network address translation](#)
- [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 \(archive\)](#)
- [What type of NAT combinations requires a TURN server?](#)
- [Under what scenarios Server- and Peer-Reflexive candidates differ?](#)

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:

$(\text{SRC_ADDR}, \text{SRC_PORT}) \rightarrow (\text{DST_ADDR}, \text{DST_PORT})$

Note:

- \rightarrow denotes the direction of the transmission.
 - $(\text{ADDR}, \text{PORT})$ denotes a transport address.
 - $(\text{SRC_ADDR}, \text{SRC_PORT})$ is the transport address of the host that sends packets.
 - $(\text{DST_ADDR}, \text{DST_PORT})$ 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 [Wikipedia](#):

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 [Wikipedia: Hairpinning](#) 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 machines:

- One outside the NAT, e.g. by directly connecting it to the internet, with no firewall. We'll call this the **[Server]**.
- One sitting behind a NAT. This is the typical situation for consumer-grade home networks, so this one will be the **[Client]**.

Set some helper variables: the *public* IP address of each host, and their listening ports:

```
SERVER_IP="203.0.113.2" # Public IP address of the Server
SERVER_PORT="1111"      # Listening port of the Server

CLIENT_IP="198.51.100.1" # Public IP address of the NAT that hides the Client
CLIENT_PORT="2222"      # Listening port of the Client
```

1. **[Client]** starts listening for data. Leave this running in **[Client]**:

```
nc -vnul "$CLIENT_PORT"
```

2. **[Server]** tries to send data, but the NAT in front of **[Client]** will discard the packets. Run in **[Server]**:

```
echo "TEST" | nc -vnu -p "$SERVER_PORT" "$CLIENT_IP" "$CLIENT_PORT"
```

3. **[Client]** performs a hole punch, forcing its NAT to create a new inbound rule. **[Server]** awaits for the UDP packet, for verification purposes.

Run in **[Server]**:

```
sudo tcpdump -n -i eth0 "src host $CLIENT_IP and udp dst port $SERVER_PORT"
```

Run in **[Client]**:

```
sudo hping3 --count 1 --udp --baseport "$CLIENT_PORT" --keep --destport "$SERVER_
↪PORT" "$SERVER_IP"
```

As an alternative to *hping3*, it's also possible to use plain *netcat*:

```
echo "TEST" | nc -vnu -p "$CLIENT_PORT" "$SERVER_IP" "$SERVER_PORT"
```

4. **[Server]** tries to send data again. Run in **[Server]**:

```
echo "TEST" | nc -vnu -p "$SERVER_PORT" "$CLIENT_IP" "$CLIENT_PORT"
```

After this command, you should see the “TEST” string appearing on the Client.

Note: The difference between a Cone NAT and a Symmetric NAT can be detected during step 3:

- If the *tcpdump* command on **[Server]** shows a source port equal to *\$CLIENT_PORT*, then the NAT is respecting the source port chosen by the application, which means that it is one of the Cone NAT types.

In this case, the data sent from **[Server]** should arrive correctly at **[Client]** after step 4.

- However, if *tcpdump* shows that the source port is different from *\$CLIENT_PORT*, then the NAT is changing the source port during outbound mapping, which means that it is a Symmetric NAT.

When this happens, the data sent from **[Server]** won't arrive at **[Client]** after step 4, because *\$CLIENT_PORT* is the wrong destination port. If you write the correct port (as discovered in step 3) instead of *\$CLIENT_PORT*, then the data should arrive at **[Client]**.

37.4.2 PyNAT

PyNAT is a tool that uses STUN servers in order to try and detect what is the type of the NAT, when running from a host behind it. To install and run:

```
sudo apt-get update ; sudo apt-get install --no-install-recommends \
python3 python3-pip

sudo -H python3 -m pip install --upgrade pynat

pynat
```

You will see an output similar to this:

```
$ pynat
Network type: Restricted-port NAT
Internal address: 192.168.1.2:54320
External address: 203.0.113.9:54320
```

RTP STREAMING COMMANDS

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 \  
  gstreamer1.0-{tools,libav} \  
  gstreamer1.0-plugins-{base,good,bad,ugly}
```

2. [Optional] Enable debug logging:

```
export GST_DEBUG="2"
```

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.

Table of Contents

- *RTP Streaming Commands*
 - *RTP sender examples*
 - * *Simplest RTP sender*
 - * *Example 2*

- * *Example 3*
- * *Example 4*
- * *Example 5*
- * *Example 6*
- * *Full-featured RTP sender*
- *RTP receiver examples*
 - * *Example 1*
 - * *Example 2*
- *SRTP examples*
 - * *SRTP simple sender*
 - * *SRTP simple receiver*
 - * *SRTP complete sender*
 - * *SRTP complete receiver*
- *Additional Notes*
 - * *About 'sync=false'*
 - * *About the SRTP Master Key*
 - * *Using FFmpeg*
 - * *SDP Offer examples*

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 \
(continues on next page)
```

(continued from previous page)

```

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 \

```

(continues on next page)

(continued from previous page)

```

    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 \
        ! 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" \

```

(continues on next page)

(continued from previous page)

```

        ! 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 \

```

(continues on next page)

(continued from previous page)

```

! 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 \
gstreamer1.0-{tools,libav} \
gstreamer1.0-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="$PWD/video.mp4" \
SELF_VSSRC=112233 \
SELF_KEY="4142434445464748494A4B4C4D4E4F505152535455565758595A31323334" \
bash -c 'gst-launch-1.0 -e \
uridecodebin uri="file://$SELF_PATH" \
! videoconvert \
! x264enc tune=zerolatency \
! rtph264pay \
! "application/x-rtp,payload=(int)103,ssrc=(uint)$SELF_VSSRC" \
! srtppenc key="$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="4142434445464748494A4B4C4D4E4F505152535455565758595A31323334" \
SELF_V=9004 \
SRTP_CAPS="payload=(int)103,ssrc=(uint)$PEER_VSSRC,roc=(uint)0, \
srtpp-key=(buffer)$PEER_KEY, \

```

(continues on next page)

(continued from previous page)

```

    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-1.0 -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="$PWD/video.mp4" \
SELF_V=5004 SELF_VSSRC=112233 \
SELF_KEY="4142434445464748494A4B4C4D4E4F505152535455565758595A31323334" \
SRTP_CAPS="payload=(int)103,ssrc=(uint)$PEER_VSSRC,roc=(uint)0, \
    srtp-key=(buffer)$PEER_KEY, \
    srtp-cipher=(string)aes-128-icm,srtp-auth=(string)hmac-sha1-80, \
    srtcp-cipher=(string)aes-128-icm,srtcp-auth=(string)hmac-sha1-80" \
bash -c 'gst-launch-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\\"user\\@example.
com\\" \
    srtpenc name=e key="$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)) \

```

(continues on next page)

(continued from previous page)

```

! "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="4142434445464748494A4B4C4D4E4F505152535455565758595A31323334" \
SELF_V=9004 SELF_VSSRC=332211 \
SELF_KEY="343332315A595857565554535251504F4E4D4C4B4A494847464544434241" \
SRTP_CAPS="payload=(int)103,ssrc=(uint)$PEER_VSSRC,roc=(uint)0, \
    srtcp-key=(buffer)$PEER_KEY, \
    srtcp-cipher=(string)aes-128-icm,srtcp-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-1.0 -e \
    rtpsession name=r sdes="application/x-rtp-source-sdes,cname=(string)\`recv\`@example.
↪com\`"" \
    srtppenc name=e key="$SELF_KEY" \
        rtp-cipher="aes-128-icm" rtp-auth="hmac-sha1-80" \
        rtcp-cipher="aes-128-icm" rtcp-auth="hmac-sha1-80" \
    srtppdec name=d \
    udpsrc port=$SELF_V \
        ! "application/x-srtcp,$SRTP_CAPS" \
        ! d.rtcp_sink \
    d.rtcp_src \
        ! "application/x-rtp,$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 \

```

(continues on next page)

(continued from previous page)

```
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: https://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: NDMyMVpZWfZWVVRTU1FQT05NTEtKSUHRkVEQ0JB.

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
```

(continues on next page)

(continued from previous page)

```
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
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 endpoints⁰, 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>
```

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:

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](#).

```
<video id="myVideo" autoplay muted playsinline></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" style="width: 320px"></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" style="width: 320px" playsinline></video>
```

With this, videos will play in *iOS Safari* as they do in any other browser, as inline videos inside their corresponding area.

SELF-SIGNED CERTIFICATES

Table of Contents

- *Self-Signed Certificates*
 - *Using a local domain*
 - * *With the Hosts file*
 - * *With Zeroconf*
 - *Trusting a self-signed certificate*
 - * *On desktop browsers*
 - * *On mobile devices*
 - * *On Node.js applications*

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 **trusted certificate** signed by a Certification Authority (CA). This should be your primary choice for final production deployments of the software.
- Make a custom, **untrusted self-signed certificate**. This can ease operations during the phase of software development and make testing easier.

Web browsers show a big security warning that must be accepted by the user. Other non-browser applications will also need to be configured to bypass security checks. This should not be a problem, given that it will only happen during development and testing.

Warning: iOS Safari is the big exception to the above comment. It will outright reject untrusted self-signed certs, instead of showing a security warning.

To test your app with iOS Safari and a self-signed cert, the cert's Root CA needs to be installed in the device itself: *Trusting a self-signed certificate*.

We strongly recommend **using a certificate generation tool** such as `mkcert`. While it is perfectly fine to issue OpenSSL commands directly, the web is full of *outdated* tutorials and you'll probably end up running into lots of pitfalls, such as newer strict browser policies, or technicalities like which optional fields should be used. A tool such as `mkcert` already takes into account all these aspects, so that you don't need to.

To generate new certificate files with `mkcert`, first install the program:

```
sudo apt-get update ; sudo apt-get install --yes \
    ca-certificates libnss3-tools wget

sudo wget -O /usr/local/bin/mkcert 'https://github.com/FiloSottile/mkcert/releases/
↳download/v1.4.1/mkcert-v1.4.1-linux-amd64'
sudo chmod +x /usr/local/bin/mkcert
```

Then run it:

```
# Generate new untrusted self-signed certificate files.
CAROOT="$PWD" mkcert -cert-file cert.pem -key-file key.pem \
    "127.0.0.1" \
    "::1" \
    "localhost" \
    "*.home.arpa"

# Set correct permissions.
chmod 440 *.pem
```

This command already includes some useful things:

- It will create a new Root CA if none existed already.
- It will use the Root CA to spawn a new server certificate from it.
- The new certificate allows accessing the server from localhost in its IPv4, IPv6, and hostname forms.
- The cert also allows accessing the server at the **.home.arpa domain wildcard*, regardless of the IP address in your network, so other machines in your internal LAN can be used for testing. Check the section below for examples of how to easily assign a domain name to your server.

Note: **.home.arpa* is the IETF recommended domain name for use in private networks. You can check more info in [What domain name to use for your home network](#) and [RFC 8375](#).

40.1 Using a local domain

40.1.1 With the Hosts file

You can take advantage of a domain wildcard such as **.home.arpa*, by adding a new entry to the *Hosts file* in client machines that will connect to your test server.

This is a quick and dirty technique, but has the drawback of having to do the change separately on each client. Also, doing this is easy in desktop computers, but not so easy (or outright impossible) on mobile devices.

On Linux and macOS, add a line like this to your */etc/hosts* file (but with the correct IP address of your server):

```
192.168.1.50 dev.home.arpa
```

Now, opening *dev.home.arpa* on a client's web browser will access your test server at 192.168.1.50.

On Windows you can do the same; the Hosts file is located at *%SystemRoot%\System32\drivers\etc\hosts*. Different systems have this file in different locations, so check here for a more complete list: [Wikipedia: Hosts_\(file\)#Location_in_the_file_system](#).

40.1.2 With Zeroconf

You can publish your server IP address as a temporary domain name in your LAN. This is a more flexible solution than editing Hosts files in every client machine, as it only needs to be done once, in the server itself.

This could be done with a full-fledged DNS server, but a simpler solution is to assign your machine a **discoverable Zeroconf address**.

For example, if your test server uses Ubuntu, run this:

```
# Get and publish the IP address to the default network gateway.
IP_ADDRESS="$(ip -4 -oneline route get 1.0.0.0 | grep -Po 'src \K([\d.]+)')"
avahi-publish --address --no-reverse -v "dev.home.arpa" "$IP_ADDRESS"
```

This technique is very handy, because practically all modern platforms include an mDNS client to discover Zeroconf addresses in the LAN.

Note: As of this writing, Android seems to be the only major platform unable to resolve Zeroconf addresses. All other systems support them in one way or another:

- Windows: [mDNS and DNS-SD slowly making their way into Windows 10](#).
- Mac and iOS include mDNS natively.
- Linux systems support mDNS if the appropriate Avahi packages are installed.

You can vote for adding mDNS to Android by adding a star (top, next to the title) on this issue: [#140786115 Add .local mDNS resolving to Android](#) (requires login; any Google account will do). **Please refrain from commenting “+1”**, which sends a useless email to all other users who follow the issue.

40.2 Trusting a self-signed certificate

Most if not all clients of any kind, will not trust a self-signed certificate when they connect to a server that uses one. What the client will do is to block the connection with an error message (this is what iOS Safari does, also Node.js apps); or show a security warning page (Chrome and Firefox web browsers).

Normally, there is some way to override this behavior. Either by installing your Root CA in the device’s root storage, or by setting some configuration. Then, the self-signed certificate will be trusted just like if it had been issued by a reputable Authority.

40.2.1 On desktop browsers

Installing the Root CA is easy because *mkcert* does it for you. In the terminal, go to the dir where your *rootCA.pem* file is located, and run:

```
CAROOT="$PWD" mkcert -install
```

40.2.2 On mobile devices

Installing the Root CA is a bit more difficult:

- With iOS, you can either email the `rootCA.pem` file to yourself, use AirDrop, or serve it from an HTTP server. Normally, a dialog should pop up asking if you want to install the new certificate; afterwards, you must [enable full trust in it](#). When finished, your self-signed certs will be trusted by the system, and iOS Safari will allow accessing pages on the `*.home.arpa` subdomain.

Note: Only AirDrop, Apple Mail, or Safari are allowed to download and install certificates on iOS. Other applications will not work for this.

- With Android, you'll have to install the Root CA and then enable user roots in the development build of your app. See [this StackOverflow answer](#).

40.2.3 On Node.js applications

Node.js does not use the system root store, so it won't accept mkcert certificates automatically. Instead, you will have to set the `[NODE_EXTRA_CA_CERTS]`(https://nodejs.org/api/cli.html#cli_node_extra_ca_certs_file) environment variable:

```
export NODE_EXTRA_CA_CERTS="/path/to/rootCA.pem"
```

One good way to do this is by adding the export to the file `~/.profile`, so it will get automatically set on every system startup.

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:

[Wikipedia: Audio Video Interleave](#)

[Wikipedia: Resource Interchange File Format](#)

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:

[Wikipedia: Builder pattern](#)

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:

[Wikipedia: Cross-origin resource sharing](#)

[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:

[Wikipedia: H.264/MPEG-4 AVC](#)

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:

[Wikipedia: Hypertext Transfer Protocol](#)

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:

[Wikipedia: Interactive Connectivity Establishment](#)

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:

[Wikipedia: IP Multimedia Subsystem](#)

[Wikipedia: 3GPP](#)

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 *Web-Socket* 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:

[Wikipedia: IP Multimedia Subsystem](#)

[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:

[Wikipedia: MPEG-4 Part 14](#)

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 are 2 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:

[Wikipedia: Multimedia](#)

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:

[Multimedia container format](#)

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:

[Wikipedia: Network address translation](#)

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 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:

[Wikipedia: QR code](#)

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:

[Wikipedia: Representational state transfer](#)

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:

[Wikipedia: RTP Control Protocol](#)

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:

[Wikipedia: Real-time Transport Protocol](#)

[Wikipedia: RTP audio video profile](#)

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:

[Wikipedia: Same-origin policy](#)

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
```

(continues on next page)

(continued from previous page)

```
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:

[Wikipedia: Session Description Protocol](#)

[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:

[Wikipedia: Session Initiation Protocol](#)

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:

[Wikipedia: Secure Real-time Transport Protocol](#)

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:

[Wikipedia: Transport Layer Security](#)

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:

[Wikipedia: VP8](#)

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.

INDICES AND TABLES

- `genindex`
- `search`

A

Agnostic media, [579](#)
 AVI, [579](#)

B

Bower, [579](#)
 Builder Pattern, [579](#)

C

CORS, [579](#)

D

DOM, [579](#)

E

EOS, [580](#)

G

GStreamer, [580](#)

H

H.264, [580](#)
 HTTP, [580](#)

I

ICE, [580](#)
 IMS, [580](#)

J

jQuery, [580](#)
 JSON, [580](#)
 JSON-RPC, [580](#)

K

KMS, [581](#)
 Kurento, [580](#)
 Kurento API, [580](#)
 Kurento Client, [581](#)
 Kurento Media Server, [581](#)
 Kurento Protocol, [581](#)

M

Maven, [581](#)
 Media
 Pad, [582](#)
 Pipeline, [581](#)
 Sink, [584](#)
 Source, [585](#)
 Media Element, [581](#)
 Media Pipeline, [581](#)
 Media Plane, [581](#)
 MP4, [581](#)
 Multimedia, [581](#)
 Multimedia container format, [581](#)

N

NAT, [581](#)
 NAT Traversal, [582](#)
 Network Address Translation, [582](#)
 Node.js, [582](#)
 NPM, [582](#)

O

OpenCV, [582](#)

P

Pad, Media, [582](#)
 Plane
 Media, [581](#), [584](#)

Q

QR, [582](#)

R

REMB, [582](#)
 REST, [583](#)
 RFC
 RFC 2616, [580](#)
 RFC 3550, [461](#), [583](#)
 RFC 3574, [580](#)
 RFC 3605, [583](#)
 RFC 3711, [585](#)

- RFC 3984, [580](#)
- RFC 4145, [403](#)
- RFC 4566, [584](#)
- RFC 4568, [584](#)
- RFC 4787, [555](#)
- RFC 5245, [580](#)
- RFC 5246, [585](#)
- RFC 6184, [535](#), [580](#)
- RFC 6386, [573](#), [586](#)
- RFC 8375, [576](#)
- RFC 8656, [322](#), [324](#)
- RFC 8837, [432](#)

RTCP, [583](#)

RTP, [583](#)

S

Same-origin policy, [583](#)

SDP, [583](#)

SDP Offer/Answer, [583](#)

Semantic Versioning, [584](#)

Session Description Protocol, [583](#)

Signaling Plane, [584](#)

Single-Page Application, [585](#)

Sink, Media, [584](#)

SIP, [584](#)

Source, Media, [585](#)

SPA, [585](#)

Sphinx, [585](#)

Spring Boot, [585](#)

SRTCP, [585](#)

SRTP, [585](#)

SSL, [585](#)

STUN, [585](#)

T

TLS, [585](#)

Trickle ICE, [585](#)

TURN, [586](#)

V

VP8, [586](#)

W

WebM, [586](#)

WebRTC, [586](#)

WebSocket, [586](#)